Informatics

Diagram merging and diffing in the context of online IDEs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Victor-Gabriel Dulcă, BSc.
Matrikelnummer 1226990

Irina Avram, BSc.
Matrikelnummer 1103974

an der Fakultät für Informatik

der Technischen Universität Wien
Betreuung: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel
Mitwirkung: Dr. Philip Langer

Wien, 25. Februar 2022 [illegible] Avram

Victor-Gabriel Dulcă Irina Avram Gertrude Kappel

Informatics

Diagram merging and diffing in the context of cloud based IDEs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Victor-Gabriel Dulcă, BSc.
Registration Number 1226990

Irina Avram, BSc.
Registration Number 1103974

to the Faculty of Informatics

at the TU Wien
Advisor: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel
Assistance: Dr. Philip Langer

Vienna, 25th February, 2022

[illegible]	Avram	
Victor-Gabriel Dulcă	Irina Avram	Gertrude Kappel

Erklärung zur Verfassung der Arbeit

Victor-Gabriel Dulcă, BSc.

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, 25. Februar 2022

Dulcă

Victor-Gabriel Dulcă

Gertrude Kappel

Erklärung zur Verfassung der Arbeit

Irina Avram, BSc.

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, 25. Februar 2022

Avram

Irina Avram

Gertrude Kappel

Acknowledgements

Throughout the writing of this thesis we have received a great deal of support and guidance from a number of people and as such would like to take this opportunity to thank them for all the help they offered along the way.

Firstly we would like to thank O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gertrude Kappel for the guidance and valuable insights she had to offer.

A special thanks is warranted to Dr. Philip Langer for his continued support, technical expertise, and feedback offered at every step.

Additionally we would like to thank Ass. Prof. Dr. Stefan Sobernig for the guidance offered during the evaluation phase of this thesis.

Finally we would like to thank our families and friends for their unwavering support throughout the duration of our studies.

Kurzfassung

Das letzte Jahrzehnt hat unzählige Fortschritte auf dem Gebiet der Technologie gebracht. Wo die Softwareentwicklung früher ein komplexer und aufwändiger Prozess war, durchgeführt in einfachen Texteditoren ohne jede Art von Syntaxhervorhebung oder Validierung, sind jetzt IDEs und speziell entwickelte Code-Editoren in diesem Bereich unübertroffen. Längst vorbei sind die Zeiten, in denen das Bauen und Ausführen selbst der einfachsten Softwareteile vom Entwickler die Abarbeitung einer Reihe von Aufgaben erforderte. Jetzt lässt sich ein Großteil dieser Aufgaben per Knopfdruck in der IDE erledigen. Ob „Ausführen“ oder „Starten“, die allgemeine Funktionalität dieses Knopfes ist klar: Wandle den Code von dem Text aus einem Editor in ein laufendes Programm, das sein Ziel erreicht, um.

Abgesehen vom Usability-Aspekt gab es in den letzten Jahren auch in anderen Bereichen Verbesserungen, was dazu geführt hat, dass frühere Nischenmethoden mehr und mehr zum Mainstream wurden. Ein Beispiel dafür ist Model Driven Development, kurz MDD. Darüber hinaus bedeutet das Aufkommen von Cloud-basierten IDEs, dass Entwickler nicht länger an die Rechenkapazitäten der Maschinen, die ihnen physisch zur Verfügung stehen, gebunden sind. Die Client-Server-Architektur solcher IDEs ermöglicht Geräten mit bescheidenen Spezifikationen, die intensiveren Aufgaben an einen Server zu delegieren und sich lediglich mit dem Senden der Anweisungen und dem Anzeigen der Ergebnisse zu beschäftigen.

Es ist an der Schnittstelle zwischen den Bereichen Cloud-basierte IDEs und MDD, wo diese Diplomarbeit ihren Beitrag leistet. MDD ermöglicht Entwicklern, komplexe Anwendungen zu erstellen, währendem sie gleichzeitig ein hohes Maß an Abstraktion beibehalten und schnell Änderungen vornehmen können, ohne sich mit codespezifischen Problemen zu befassen. Da komplexe Anwendungen entsprechend komplexe Modelle hinter sich haben, ist es oft so, dass mehr als ein Entwickler an einem Modell und/oder Diagramm eines Systems arbeitet. Dies erhöht den Bedarf an kollaborativer Modellierungsunterstützung für diagrammbasierte Darstellungen, ähnlich der textbasierten Unterstützung, angeboten von Tools wie Git [git21a] oder SVN [sub21], die verschiedene Versionen einer Datei miteinander vergleichen und mergen.

Diese Dimplomarbeit analysiert den verfügbaren Support für Vergleichen und Mergen von Diagrammen, sowohl in den am häufigsten verwendeten Cloud-basierten IDEs als auch in vollwertigen IDEs und stellt eine eigene Implementierung für die Theia Cloud-basierte

IDE [the21] bereit. Schon existierende und bewährte Komponenten und Frameworks wurden wiederverwendet, wo immer es geeignet war. Somit werden dem Benutzer bereits bekannte Diff-Visualisierungsmittel, wie beispielsweise Seite-an-Seite-Vergleiche, sowie erprobte Mergemechanismen präsentiert. Die Durchführung des Vergleichs sowie der Mergeprozess werden vom EMF-Compare-Framework übernommen, wobei der Fokus dieser Arbeit auf der Visualisierung der Diffs und der möglichen Mittel zur Konfliktlösung und Diagrammmerging liegt.

Die Ergebnisse wurden im Hinblick auf die Effizienz des Protokolls, der die Kommunikation zwischen den Frontend- und Backend-Komponenten ermöglicht, sowie im Hinblick auf die Verwendbarkeit der entwickelten Visualisierungs- und Merge-/Konfliktlösungsmittel bewertet. Im Falle des Kommunikationsprotokolls wurden die Performance durch Benchmarking gemessen. Für die Bewertung der Benutzerfreundlichkeit wurde die System Usability Scale [B+96] verwendet. Die Ergebnisse zeigen, dass die entwickelten Vergleichs- und Mergefunktionen für die Mehrheit der Befragten einfach zu bedienen und für die täglichen Aufgaben ausreichend waren. Das System hat auf der Skala der Benutzerfreundlichkeit zwischen gut und ausgezeichnet abschnitten.

Abstract

The last decade has brought about countless advancements in the field of technology. Where once software development was a toilsome process done in plain text editors, lacking any sort of syntax highlighting, validation or other means of aiding the developer, now IDEs and specially designed code editors reign supreme in this field. Long gone are the days when building and running even the simplest pieces of software, required the developer to undertake a series of tasks. Now most of these tasks can be achieved through the push of a button in the IDE. Be it called "Run", "Execute" or "Start", the overall functionality of that button is clear: turn the code from text in an editor to a running program that achieves its goal.

Aside from the usability aspect, the last few years have seen improvements in other areas, with formerly niche methodologies becoming more and more mainstream, such as Model-driven development, or MDD for short. Furthermore the emergence of cloud-based IDEs means that developers are no longer bound by the computational capabilities of the machines that are physically available to them. The client-server architecture of such IDEs allows devices with modest specifications to delegate the more intensive tasks to a server and merely concern themselves with sending the instructions and displaying the results.

It is at the intersection between the fields of cloud-based IDEs and MDD where this thesis makes its contribution. MDD allows developers to build complex applications while maintaining a high level of abstraction, and to quickly make changes without concerning themselves with code-specific issues. Because complex applications often have correspondingly complex models behind them, it is often the case that more than one developer will be working on a model and/or diagram of a system. This raises the need for collaborative modeling support for diagram-based representations, in a similar way to the text-based support offered by tools such as Git [git21a] or SVN [sub21], which allow different versions of a file to be compared and merged with each other.

This thesis analyzes the available support for diagram diffing and merging both within the most commonly used cloud-based IDEs, as well full-fledged IDEs, and provides an implementation of its own for the Theia cloud-based IDE [the21]. Already existing, and well established components and frameworks have been reused wherever suitable, thus presenting users with somewhat known diff visualization means, such as side by side comparisons, as well as merging mechanisms. The computation of the comparison as well as the merging process are being handled by the EMF Compare framework, with the focus of this thesis lying on the visualization of the diffs and means of conflict resolution and diagram merging.

The results have been evaluated in regard to the efficiency of the protocol facilitating the communication between the frontend and the backend components of the implementation, as well as in regard to the usability of the developed visualization and merging / conflict resolution means. This evaluation has been conducted by way of benchmarking in case of the communication protocol while the System Usability Scale [B$^+$96] has been used to measure the usability of the implementation. The results themselves show that the developed diffing and merging capabilities were easy to use and sufficient for daily tasks for a majority of respondents, scoring between good and excellent on the System usability scale.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation and Problem Statement . . . 1
1.2 Aim of the Work . . . 2
1.3 Methodological Approach . . . 4
1.4 Structure of the Work and Authorship . . . 5

2 State of the Art **7**
2.1 Cloud IDEs . . . 7
2.2 Model/Diagram diffing and merging . . . 12

3 Design and Implementation **25**
3.1 Architecture description . . . 25
3.2 Implementation details . . . 27

4 Comparison example **63**

5 Summary and Evaluation **69**
5.1 Summary . . . 69
5.2 Evaluation . . . 71

6 Open Challenges and Future Work **77**
6.1 Diff Granularity Setting . . . 77
6.2 Diff Grouping based on Level . . . 78
6.3 Additional properties view . . . 79
6.4 Source code management tool integration . . . 80

List of Figures **81**

Bibliography **83**

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Cloud based Integrated Development Environments (IDEs) facilitate programmers to write, compile, run and test code in the cloud. By having data and applications all stored in the cloud they simplify coding and collaboration, whilst also mitigating any possible issues caused by differences in hardware or operating systems. Yet they are a rather new concept that falls in line with the trend of ever thinning clients and the transfer of the bulk computation tasks to a remote server. As such they still lack some features that would be of considerable help to the developers using them. One such feature would be the ability to support diagram versioning in an efficient manner [OWK03]. An efficient manner would imply the ability to accurately process the differences between versions, merge different versions of a diagram [MNSD17] [FGW$^+$15] and also display this data in a comprehensible way [MM15].

Diagrams are an integral part of many design processes and are constantly used in various industries, be it information technology to model systems, manufacturing to showcase assembly and build processes etc. [AGD18] [DS15].

Remote work has also become commonplace, meaning the need for thin clients has increased together with the portability of the gadgets used, such as tablets, notebooks and even smartphones. This is where cloud based IDEs come into play. More than one person tasked with working on a specific project is also standard, which means each team member will have his or her own version of a diagram at a given time, until they decide to sync them up with the ones on the server.

This is where current capabilities are lacking [KHL$^+$10]. The users need a way to merge their work into the same diagram so that no work is lost, while also being able to clearly see the differences between two versions. This will ensure they are kept up to speed with the work done by other team members.

Frameworks such as EMF compare [SBMP08] are able to compute the differences between diagram versions, it is however not suited for a thin client [KKT11]. This means it cannot be run directly in the browser.

Furthermore the diffing of diagrams is a difficult task in itself, even without the context of cloud based IDEs. This is because a diagram is visual in nature and one has to account for a multitude of possible changes, such as positions, naming etc. When talking about cloud based IDEs the entire process becomes more complex and more challenging. This is because the diff computation is executed remotely, thus resulting in the need for suitable serialization and deserialization of the diagram. Also diagrams can, depending on their intended use case, become quite complex and thus transmitting the entire diagram back and forth may be inefficient. One possible solution to be analyzed in this work is the sending of only the diff elements from the server to the client and letting the client deal with the showcasing of the diffs.

Relating to the client-side showcasing of the diffs, the fact that a Graphical Language Server Protocol (GLSP) [gls21b] and Sprotty [spr21b] will be employed represents an advantage, as GLSP provides a flexible architecture for graphical modelling languages, and the Sprotty framework adds diagramming visualization support to GLSP through animation, navigation and editing capabilities. This means that a whole new array of possibilities is made accessible when it comes to the styling and rendering of diagrams. How these possibilities can be used in order to improve the existing mechanisms for diff showcasing will be explored in this work, be it either through the animation support or through other means.

For these reasons a way to compute the differences and to showcase them to the user has to be developed for cloud based IDEs, more specifically to this thesis, for the Theia IDE [the21].

1.2 Aim of the Work

The goal of this work is to develop the necessary mechanisms for the diffing and merging of diagrams in cloud based diagram editors. This functionality is to be subsequently integrated in the Theia Cloud IDE in terms of a diff editor.

The differences between diagrams will be computed on a server by using EMF compare [Tou06] and then the results will be sent to the client, which will in turn display them to the user. In order to achieve this, a suitable protocol for sending the diff data to the client will be developed. The data will be converted to a JSON format during transmission and will be reconstructed by the client.

Because of the potential complexity and size of diagrams the aforementioned protocol needs to be able to convey information about the diagram differences without sending the entire diagram between the server and the client. The information received has to be sufficient in order for the client to be able to accurately showcase the differences to the users.

A transmission protocol will, however, not be sufficient as the differences in the diagrams will have to be displayed once the client has received the data. In this regard a suitable way to showcase the differences between the diagrams in a comprehensible manner has to be developed, so that the users will be able to understand what is being presented to them at a glance [GWKRM15]. The diagrams will be built on and work with the Graphical Language Server Framework or GLSP for short [gls21b]. GLSP is a framework that provides extensible components enabling the integration of editable diagrams in web applications.

While a client-server architecture and a protocol are required in order to implement the desired functionality, the focus of this work will lie on the merging and diffing of diagrams. Diagrams are far more complex when compared to code files with regard to merging or diffing. For example, properties and positions can be moved and relations between the elements can change. These aspects will all have to be taken into account when merging and diffing in order to be able to correctly identify the changes that took place in the diagram and subsequently showcase them to the user. Ultimately, as the result of this work, functionality for merging and diffing diagrams will be added to Theia [the21].

Perhaps the most challenging part of this work and where the most improvements can be made is the showcasing of differences between the diagram versions [MM15] [Gir06]. While this can easily be achieved by referring to the textual representation of the diagram that has been used for transmission from the server to the client, it is far from trivial to illustrate these differences for the graphical representation of the diagram. One has to decide which differences are pertinent and should be represented and which ones should be blended out as to avoid clutter, e.g., position changes. Furthermore, a collection of pertinent filtering / grouping mechanisms should be defined in order to help the users examine various types of changes without losing sight of the bigger picture.

In order to achieve the goals previously set, the technologies used will be analyzed in order to identify all they have to offer and subsequently use it in the work. This includes the wide variety of showcasing possibilities made available by the powerhouse combination of CSS and SVG. These technologies will aid in the construction of a suitable visual representation for the diagram diffs. In summary there are two main research questions handled in this work, with a series of necessary achievements on the way to answering them. The research questions themselves could be phrased as:

- How to design and implement an efficient application-layer protocol for communicating diff data in a client-server architecture.
- How to design and implement a useful notation to display and navigate diagram diffs.

The aforementioned research questions could be broken down into smaller, more concrete steps thus resulting in the following individual tasks:

- Identification of relevant changes such as additions/deletions of classes, attributes, etc. to the diagram, disregarding the less important ones such as layouting or other purely aesthetic changes, thus keeping the representation from becoming overly cluttered with superfluous information.
- Developing visual means for conflict resolution and user interaction with two diagram versions.
- Automatic merging, should no manual conflict resolution be necessary.
- Detection of relevant, model-specific conflicts and developing of conflict resolution means.

1.3 Methodological Approach

The methodology used in this thesis will be based on Hevner et al. [HMPR04] and his work regarding design science, thus ensuring this thesis tackles a contemporary problem, namely the merging and diffing of diagrams in cloud based IDEs, in a structured manner while aiming to provide a verifiable solution to it, in form of an extension to the Theia IDE.

1. **Problem identification and motivation** The problem with the state of the art is its lack of support for collaborative modeling due to missing graphical diffing and merging capabilities. Upon literature research and the evaluation of the three most popular cloud-based IDEs, none of them offer such capabilities out of the box, nor do they provide a way for developers to implement such functionality themselves. The motivation behind the conception of this thesis becomes apparent once one factors in the increasing capabilities and use of cloud-based IDEs, especially in the industrial sector, where there is significant overlap between the areas of cloud IDEs and model driven development.
2. **Defining the objectives for a solution** The aim of this work is to improve upon the status quo by offering adequate diffing and merging support for diagrams. Concretely we aim to reduce the error rate caused by inadequate merging mechanisms (such as text-based representation of a diagram), increase user understanding of the changes that took place, and provide a faster merging mechanism than what is currently available. The aforementioned functionality is meant to be integrated in the Theia cloud-based IDE.
3. **Design and implementation** The architecture of the system, including the server side, the transfer protocol and the client side has been designed and polished in order to ensure the viability of the subsequent implementation. During the design phase UML diagrams were created in order to crystallize the key components of the software. When it comes to the user interface of the tool, literature research and reviews of existing tools were conducted in order to adhere to best practices as well as any conventions / standards. Once the architecture of the system has been

defined work started on the implementation. As the need arose, adjustments were be made to compensate for potential design flaws that escaped the design phase. The implementation was designed in a way that allows integration with existing technologies that are relevant to the scope of the thesis, such as the EMF Compare and GLSP.

4. **Evaluation** The evaluation of this work has been conducted in a qualitative manner through a series of semi-structured interviews and benchmarks. The efficiency of the developed application-layer protocol was analyzed by performing a series of measurements regarding the timeliness of the roundtrips between the client and the server. In order to ascertain to what extent we have succeeded in developing useful visualization means we will be relying on the System Usability Scale [B$^+$96], or SUS for short. Based on the answers gathered from interviewees a total SUS score has been computed and subsequently relied upon when it came to gauging the usability of the interface and implicitly the developed means for diff visualization.

1.4 Structure of the Work and Authorship

This chapter describes how this thesis is structured and briefly summarizes each of its chapters. In addition to said summary, information regarding the authorship of each chapter, as well as the implementation and division thereof between the authors, will be provided.

Chapter 1 describes the motivation behind the genesis of this thesis, such as the lacking support for collaborative work on diagrams, especially in the context of cloud based IDEs. Furthermore it describes what we aimed to achieve, namely provide graphical diffing and merging capabilities for diagrams in the Theia IDE. Lastly the methodological approach, based on Hevner et al.'s work regarding design science, is described. This chapter has been written by Mr. Dulcă.

Chapter 2 describes what effectively constitutes the status quo regarding diagram diffing and merging capabilities of existing cloud based IDEs, as presented in chapter 2.1 by Mr. Dulcă, as well as the currently available or proposed mechanisms for model / diagram diffing (not limited to a cloud based context), as illustrated in chapter 2.2 by Ms. Avram.

Chapter 3 tackles the concrete design and implementation of the Diff-Merge extension and all of its components, as well as the developed protocol responsible for facilitating the communication between said components. Chapter 3.1 has been done by Ms. Avram, while the implementation in chapter 3.2 has been equally between her and Mr. Dulcă.

In chapter 4 Ms. Avram provides an example of a comparison between two diagrams conducted using the Diff-Merge tool and takes the reader through all the steps from triggering a comparison to concluding a merge in order to offer them a better understanding of the inner workings of the developed tool.

Chapter 5 summarizes the work done during the course of this thesis and illustrates the methodologies used for the subsequent evaluation of the various components of the Diff-Merge tool, such as benchmarking for measuring performance, the System Usability Scale used to gauge the usability. This chapter has been authored by Mr. Dulcă.

Chapter 6 outlines the open challenges and potential for future work that have been identified as a result of the conducted evaluation. Work on this chapter has been divided equally between the two authors, with chapters 6.1 and 6.2 being the work of Mr. Dulcă while chapters 6.3 and 6.4 are the work of Ms. Avram.

CHAPTER 2

State of the Art

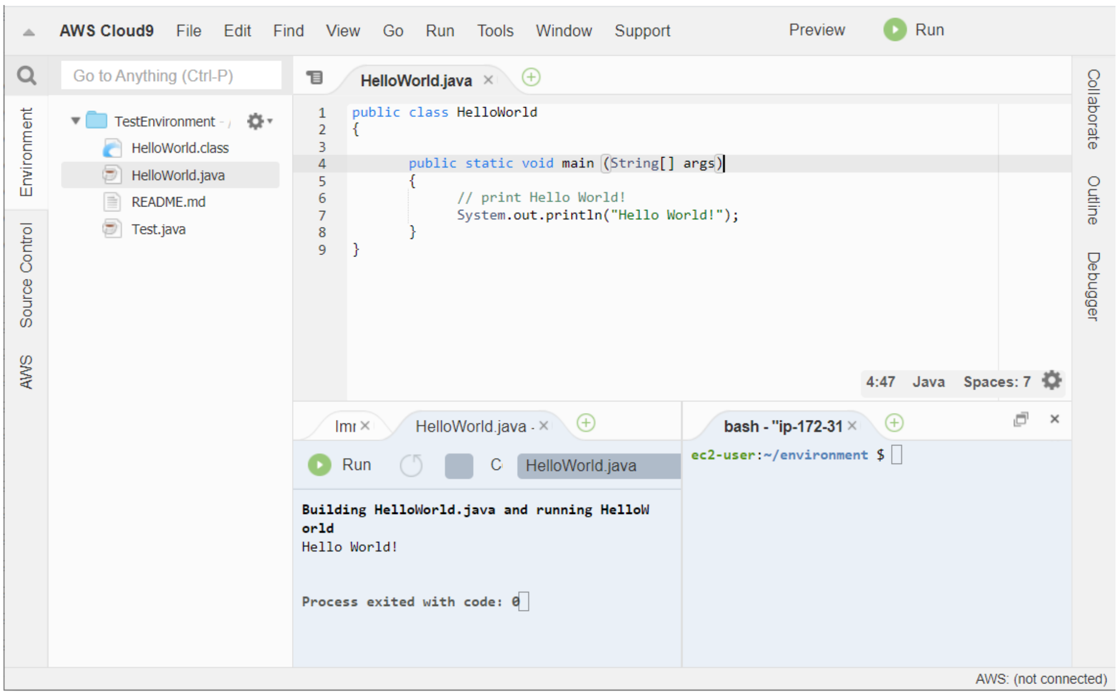
This thesis touches on a number of subjects such as cloud IDEs, models, diagrams and diffing and merging of said models and diagrams. As such this chapter will present the state of the art for those topics, in an attempt to give the reader an overview of the current status quo and thus help him or her view the contents of the work in an appropriate context. Since there is little to no overlap between cloud IDEs and diagram diffing and merging these two topics will be discussed separately, starting with the cloud IDEs.

2.1 Cloud IDEs

The first topic to be discussed in this chapter are the online IDEs, with the top 10 most popular ones being mentioned and the top 3 being analyzed in greater detail. This classification is based on the PYPL [ode21] index, which gets its data from Google Trends [gtr21]. The top 3 make up roughly 70% of the market share on Google Trends thus being warranted a closer look.

Rank	IDE	Market Share
1	Cloud9	39.4 %
2	Repl.it	16.82 %
3	JSFiddle	13.73 %
4	Koding	9.01 %
5	Codio	6.44 %
6	PythonAnywhere	4.66 %
7	Ideone	3.89 %
8	DartPad	2.58 %
9	Goorm	1.22 %
10	Codeanywhere	1.12 %

The most searched for cloud-based IDE is **Cloud9** [clo21], with a market share of 39.4% at the time of this writing. Cloud9 is a cloud IDE offered by Amazon that ties in their Amazon Web Services stack. It can be started seamlessly from the EC2 console and one can be up and running within just a few seconds. Cloud9 allows coding, running the code and debugging directly from the browser. The rich code editor and the debugger are relying on an elastic computing instance that is automatically started in the background whenever a new Cloud9 environment is created. The platform upon which the IDE runs is Linux based, with the user being able to choose from either Amazon Linux (optimized for their own EC stack, offering bleeding edge software repositories etc.) or Ubuntu Server.



```
public class HelloWorld
{

        public static void main (String[] args)
        {
            // print Hello World!
            System.out.println("Hello World!");
        }
}
```

```
Building HelloWorld.java and running HelloW
orld
Hello World!


Process exited with code: 0
```

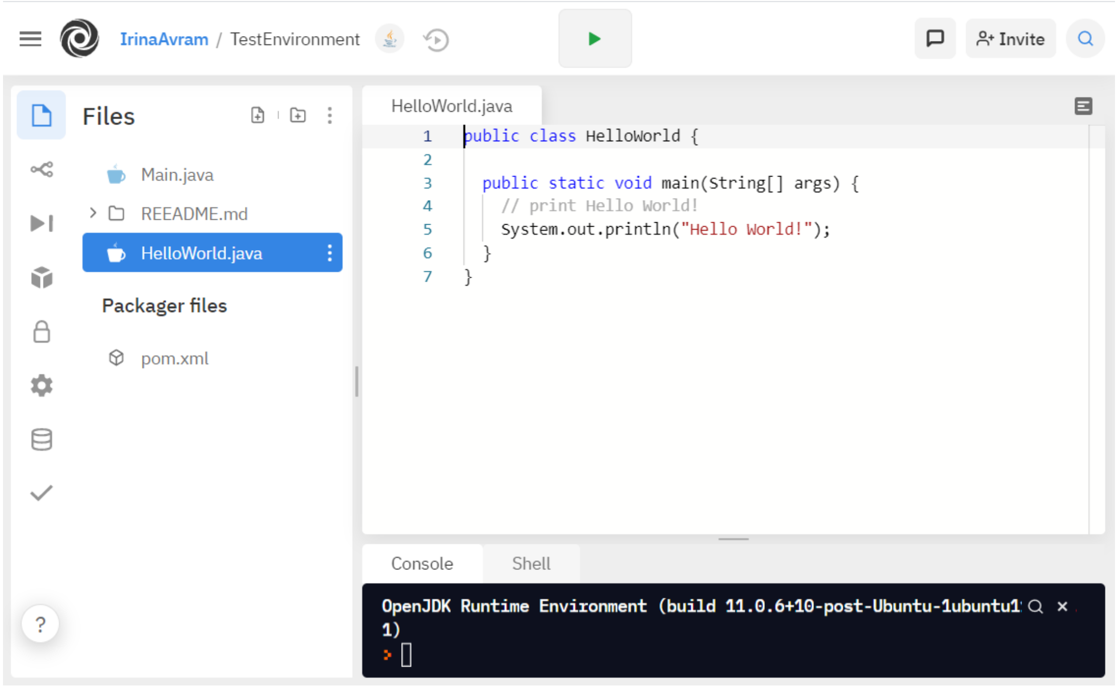
```
ec2-user:~/environment $
```

Figure 2.1: Cloud9 UI

Once running, the IDE offers a familiar sight, as can be seen in figure 2.1, with a project explorer on the left side of the screen and a system terminal in the lower part, with the actual code, or contents of the open file, being shown in the central part of the screen above the terminal and to the right of the project explorer. When it comes to the concrete capabilities of the IDE, it supports the expected features, such as syntax highlighting, code completion, hints and linting for a wide variety of popular programming languages such as C++, Java, Python, Typescript etc. Some languages such as Go, Node.js, Python, C++ and PHP also benefit from debugging functionality in addition to the aforementioned features.

On one hand Cloud9 checks a lot of the boxes that a developer might look for in an IDE with the caveat that it is not truly and entirely free. In order to run it requires either an EC2 instance or a remote server with SSH access. On the other hand, since it is running in a Linux environment, this allows the user to install a myriad of tools that they might need, such as Maven [mav21] or Docker [doc21]. Depending on the developer's needs this combination of factors may impact how suitable a solution Cloud9 IDE really is.

The second entry on the list is **Replit** [rep21]. This cloud-based IDE supports over 50 languages such as PHP, Go, Java, Python, C, C++, Haskell etc. and has a market share of 16.82%.



```
public class HelloWorld {

  public static void main(String[] args) {
    // print Hello World!
    System.out.println("Hello World!");
  }
}
```

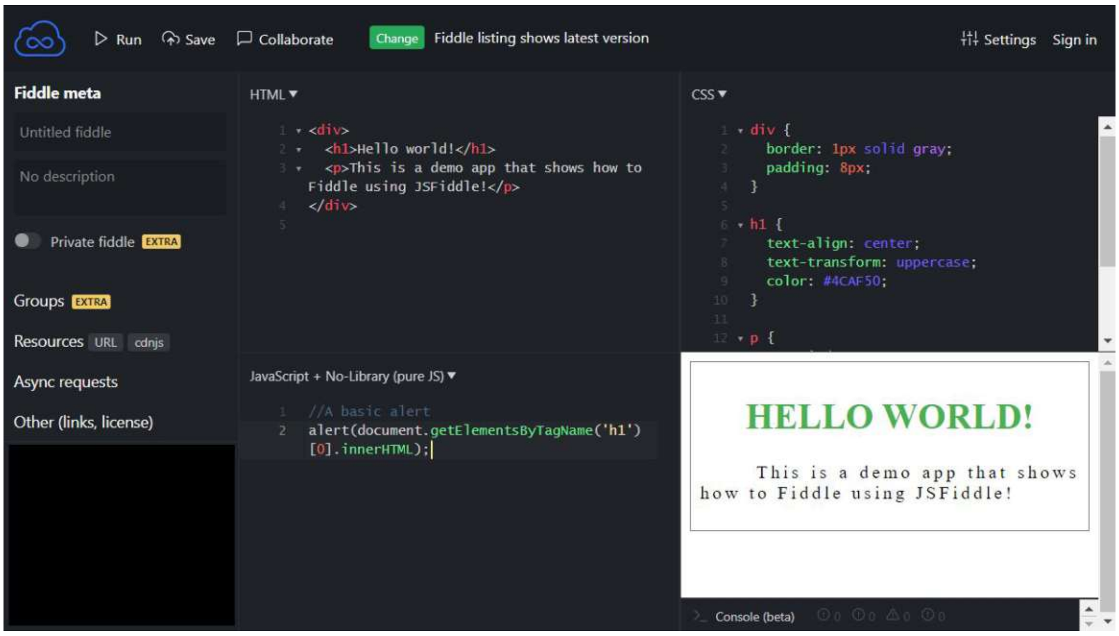
```
OpenJDK Runtime Environment (build 11.0.6+10-post-Ubuntu-1ubuntu1
1)
> 
```

Figure 2.2: Replit UI

The layout, depicted in figure 2.2, is somewhat simple and similar to how IDEs are constructed in general. The user is greeted by three main windows, a project explorer on the left, an editor window beside that, and a terminal below the editor. The project explorer doubles as a version control, settings, and dependency management window. This type of layout is commonplace in a series of IDEs, e.g. Visual Studio Code [vsc21].

Repl.it goes a step further when it comes to its ease of use by offering an intuitive GUI for dependency and package management. The user can simply search for the desired dependency, such as frameworks and libraries. The UI is the same regardless of the programming language, that has been chosen, but the dependency management tool, that is invoked, differs. For example Maven is employed for Java projects, Poetry [poe21] is the dependency and packaging tool utilized for Python and npm [npm21] is used to manage Node.js projects. While all these features might allow Replit to be appealing to some users, there are some caveats that need to be addressed. The source code is public for members of the free plan, similar to GitHub [git21b], where users can fork projects and work on their own copy, without being able to make changes to the original. Paid plans offer the option to make projects, or “repls” as they are called, private, while also offering increased memory, computing power and hosting for running software.

The third contender on the list, with a market share of 13.73% at the time of this writing, is **JSFiddle** [jsf21]. While JSFiddlle is not a cloud IDE in the true sense of the word it does allow users to construct quick prototypes based on HTML, CSS and Javascript, without any registration or costs.



```html
<div>
  <h1>Hello world!</h1>
  <p>This is a demo app that shows how to Fiddle using JSFiddle!</p>
</div>
```

```css
div {
  border: 1px solid gray;
  padding: 8px;
}

h1 {
  text-align: center;
  text-transform: uppercase;
  color: #4CAF50;
}

p {
```

```javascript
//A basic alert
alert(document.getElementsByTagName('h1')[0].innerHTML);
```

HELLO WORLD!

This is a demo app that shows how to Fiddle using JSFiddle!

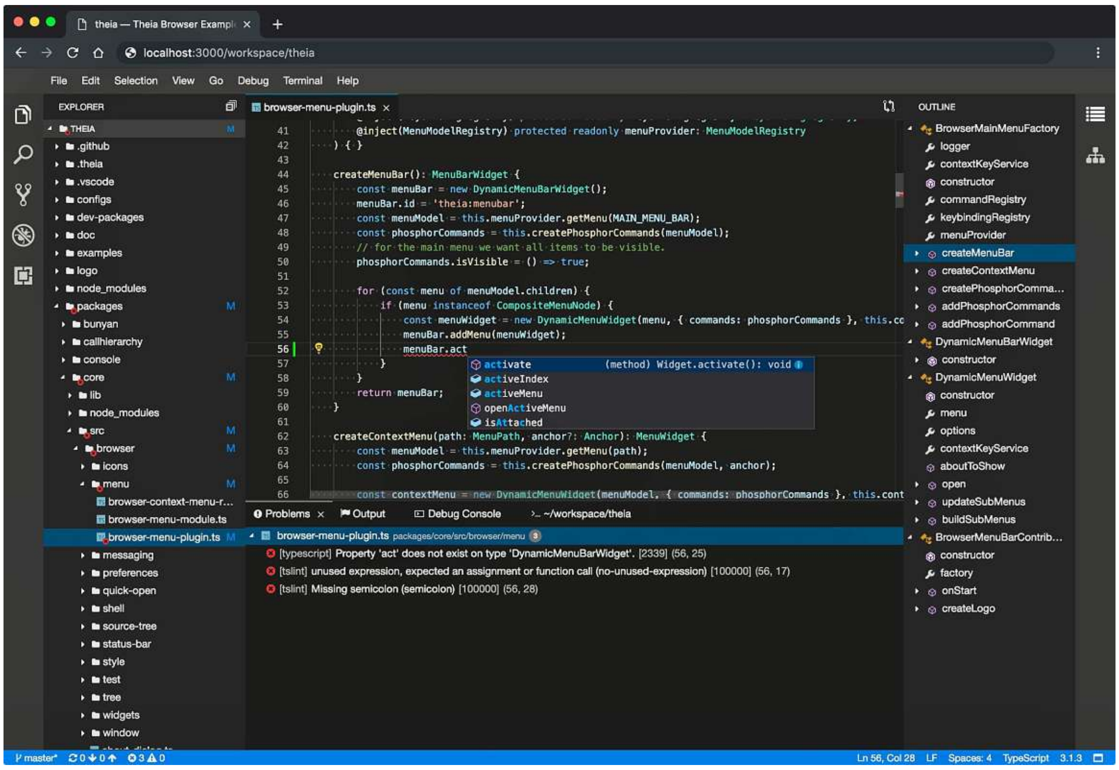
Figure 2.3: JSFiddle UI

Figure 2.3 shows a simplistic layout, with 4 rectangular boxes, HTML, CSS, JavaScript and Result, being shown on the screen. An option to open up a console is also available underneath the Result window. Even though limited in scope compared to other online IDEs, it does offer syntax highlighting, code completion and linting. In spite of the aforementioned limitations JSFiddle has been included in this chapter because of the high ranking in the PYPL index as well as the fact that it is prevalent in the online resources regarding frontend development such as StackOverflow [sta21] where a vast majority of questions and/or answers related to JavaScript or HTML rely on JSFiddle to showcase the examples.

In spite of all the apparent differences between the three aforementioned cloud-based IDEs they all share one common characteristic, or better said, lack thereof. None of them offer support for custom plugins or extensions. This basically means that users are limited to the functionality that comes out of the box with the IDE for the most part.

The precursor to AWS Cloud9, c9.io, did indeed allow users to load third party plugins. This feature was then also migrated to the Cloud9 IDE, in an experimental capacity, only for it to be removed at some later point in time [aws21b] . That being said, Amazon was in the process of evaluating ways to officially support third-party plugins in AWS Cloud9 [aws21a] but at the time of this writing nothing had come of it.

This is where the Theia cloud-based IDE [the21] comes in. Theia is a vendor-neutral, open-source IDE platform that runs in browsers and on desktops. This means it is not necessarily meant to be an end-product in and of itself, instead being targeted at IDE developers who want a VS-Code [vsc21] that runs in the cloud and/or simply want an alternative to forking VS-Code. While it is not the same product, Theia does however run VS-Code extension, which means that users can browse through the almost 27.000 extensions that are already available and mix and match them in order to ultimately obtain an IDE that offers the required features for their tasks, all while running in a browser.



```
        @inject(MenuModelRegistry) protected readonly menuProvider: MenuModelRegistry
    ) { }

    createMenuBar(): MenuBarWidget {
        const menuBar = new DynamicMenuBarWidget();
        menuBar.id = 'theia:menubar';
        const menuModel = this.menuProvider.getMenu(MAIN_MENU_BAR);
        const phosphorCommands = this.createPhosphorCommands(menuModel);
        // for the main menu we want all items to be visible.
        phosphorCommands.isVisible = () => true;

        for (const menu of menuModel.children) {
            if (menu instanceof CompositeMenuNode) {
                const menuWidget = new DynamicMenuWidget(menu, { commands: phosphorCommands }, this.co
                menuBar.addMenu(menuWidget);
                menuBar.act
            }            activate              (method) Widget.activate(): void
        }                activeIndex
        return menuBar;  activeMenu
    }                    openActiveMenu
                         isAttached
    createContextMenu(path: MenuPath, anchor?: Anchor): MenuWidget {
        const menuModel = this.menuProvider.getMenu(path);
        const phosphorCommands = this.createPhosphorCommands(menuModel, anchor);

        const contextMenu = new DynamicMenuWidget(menuModel, { commands: phosphorCommands }, this.cont
```

```
browser-menu-plugin.ts packages/core/src/browser/menu 3
[typescript] Property 'act' does not exist on type 'DynamicMenuBarWidget'. [2339] (56, 25)
[tslint] unused expression, expected an assignment or function call (no-unused-expression) [100000] (56, 17)
[tslint] Missing semicolon (semicolon) [100000] (56, 28)
```

Figure 2.4: Theia UI [the21]

Once running the user is greeted by a familiar look and feel, noticable in figure 2.4, almost indistinguishable from VS Code. A project explorer takes up the left part of the window, with tabs for version control, searching and debugging also available. The lower part is taken up by a terminal while the center section is dedicated to the active editor. While it does not support any language specific features, such as syntax highlighting or autocompletion out of the box, these can be added with ease through either a VS-Code extension or a Theia-specific one. On one hand this means that less functionality is available to the user by default, while on the other hand the total scope of functionality that can be added is much broader than with most cloud-based IDEs.

An extension provides certain functionality through a collection of widgets, commands and handlers. Because in Theia everything is wired up through dependency injection, an extension has to define one or more dependency injection modules. The functionality provided by an extension can be manifold, ranging from an UI extension all the way to contributing a language server in the backend. What all of this means is that developers are free to contribute any functionality they wish and so they did. Diagramming support already exists as an add-on part to the Theia IDE in the form of the Sprotty [spr21b] extension. Sprotty is an open-source, web-based diagramming framework.

It is precisely because of the ease with which it can be extended and the already-available support for diagramming that Theia proves to be a promising candidate for visual diffing and merging of diagrams.

2.2 Model/Diagram diffing and merging

With technology constantly advancing and becoming more complex, the area of software engineering does not fall far behind, as the size and capabilities of the built systems are constantly increasing. As stated by Brosch et al. in “An introduction to model versioning” [BKL+12], one way of coping with said complexity has been to raise the abstraction level of the languages used for the specification of said systems. Model-driven engineering, or MDE for short, has been proposed as one mechanism capable of raising the abstraction level. In the case of MDE the basis used for the automatic generation of an executable system is represented by the models themselves. This fact allows developers to remain closer to the problem domain, rather than focus on individual implementations when building their models.

2.2.1 Classification of versioning systems

Because of the aforementioned complexity and scope of systems, the number of developers required is accordingly large. This renders the support for team-based development of models to be a “crucial prerequisite for the success of MDE”, according to [BKL+12]. This fact draws a parallel to traditional software engineering, where versioning capabilities are required. According to Altmanninger et al. [ABK+09], the use of *text-based versioning systems* such as Subversion [sub21], Git [git21a], or CVS [cvs21], for models has lead to unsatisfactory results, thus proving itself to be inadequate for this particular use case. Because these systems use a text-based representation of the model, information contained in the graph-based representation becomes lost. In order to overcome this drawback, *versioning systems based on the graph structure of a model* have been proposed.

Optimistic versioning systems became popular, as they allowed multiple developers to simultaneously work on the same artifacts, without any of them being locked while it was being worked on by another developer. This way of work means that all of the changes performed by the various developers have to be merged again, in a version of the artifact that contains all the changes. Provided that multiple developers check out the artifacts they plan to work on at the same moment in time, once the first developer has finished his or her work, they can simply check in their changes to the repository. Since no other changes have been submitted in the meantime their changes get directly saved to the repository, as a revised version. Once a second developer attempts to check in their changes however, the merging process will be triggered, as a new version of the artifact has been submitted to the repository between the time the second developer checked out the artifact and the moment they attempted to submit their changes. Once the merging process has been completed the repository version of the artifact will reflect both the changes performed by the first developer, as well as the ones made by the second one. In their work Brosch et al. [BKL$^+$12] talk about the foundations of versioning and describe the goal of versioning systems as being twofold. On one hand they maintain a historical archive of artifacts as they undergo various changes, and on the other hand they help manage the evolution of software artifacts.

In order to better categorize the various technologies for software versioning, Conradi and Westfechtel [CW98] have proposed version models that specify which objects are to be versioned, how versions are identified and organized and how to create new versions and retrieve existing ones. It is within these model versions that Conradi and Westfechtel make the distinction between the *product space* and the *version space*. The *product space* ignores any versioning information and instead only describes the structure of the product, in this case a piece of software. The *version space* on the other hand focuses on the evolution of the artifact by introducing versions and relationships between them, such as deltas. The structure of the product is not taken into account in this case.

Conradi and Westfechtel also differentiate between *extensional* and *intentional* versioning, with the former being most commonly encountered presently. This type of versioning consists of retrieving previously created versions of a software artifact, with concerns regarding efficient storage, version identification and immutability. This is achieved by explicitly checking in each version and attributing each one of them a unique number. Intentional versioning on the other hand consists of the automatic generation of consistent versions from the version space. Properties can be annotated to specific artifact versions and they can be subsequently queried in order to generate a new artifact based on the specific versions of the various properties.

When it comes to designing versioning systems and the impact said design has on the merge result, Brosch et al. [BKL$^+$12] categorize the current merging approaches in two dimensions, a dimension for the product space and a second one concerning "how deltas are identified, represented, and merged in order to create a consolidated version."

The dimension for the product space describes the specific representation of the artifact which is to be merged and it can be either graph- or text-based.

Depending on the specific merge approach, the merge is being conducted either on the versions of the artifact, for a *state-based merging approach*, or it can rely on the operations that have been applied to the artifact between a common ancestor and the two successors, in the case of *operation-based merging approaches*. Regardless of the specific merge approach conflicts might arise during the merge process of two different versions. The two basic types of conflicts that occur, are **update-update**, when two elements have been changed in both versions and **delete-update**, where a version consists of an updated version of an element, whereas the other version has the same element deleted [BKL+12].

In case of *text-based merging* the textual representation of an artifact is used, with the atomic unit of the version text consisting of “a paragraph, a line, a word, or even an arbitrary set of characters”. Because this approach relies exclusively on a textual approach, it does not require any knowledge regarding the content of the versioned files, such as language-specific knowledge. This fact makes the text-based merging approach programming-language-agnostic and grants it a degree of simplicity and efficiency which has ensured its widespread adoption in practice. These advantages constitute a doubly-edged sword however, as they make the merging process prone to introducing compile- and run-time errors. *Graph-based approaches* have emerged as a result of the aforementioned shortcomings of the text-based merging approach. As the name suggests, this type of approaches “operate on a graph-based representation of a software artifact for achieving more precise conflict detection and merging”. This precise conflict detection and merging can be achieved by one of two means, *syntactic* and *semantic* merge approaches [Men02].

The syntax of a programming language is being taken into account by the *syntactic merge approaches* by translating the artifact into an abstract syntax tree prior to performing the merge itself. This allows for the merge to be performed in a syntax-aware manner, thus possibly avoiding syntactically erroneous results, as well as unimportant textual conflicts, for example as a result of formatting the file. The latter advantage of syntactic merging may however lead to the loss of intended formatting during the translation from the graph-based representation back to the textual one.

The semantic merge approaches are more advanced, offering detection for things such as infinite loops and undeclared variables. This is achieved by taking the static and dynamic semantics of a programming language into account, a fact which comes at a price, namely an increased computational complexity and an intrinsic language dependency.

The second dimension according to which current merging approaches can be categorized concerns the *identification, representation and merging of deltas*. In this context the state-based and operation-based merging can be further analyzed. In *state-based merging* the versions of an artifact are being compared to each other in order to establish the aforementioned deltas, and all of the non-conflicting changes will be merged. This type of approach can be applied to both a two-way merge as well as to a three-way one. A two-way merge will be performed between two versions of the same artifact, whereas a three-way merge will take place between two artifact versions and their common ancestor. This allows for deletions to be identified only in case of a three-way merge, as without information about a common ancestor it is impossible to ascertain if a change represents an addition in one of the versions or a deletion in the other one.

Furthermore in order to perform a state-based comparison and merge, a match function is required. The role of this function is to identify the correspondences between the elements being compared across the two versions. As such this function and its degree of accuracy are of paramount importance when it comes to the quality of the resulting merge, thus requiring graph-based approaches to employ more advanced techniques for it.

Assuming such a match function has been defined, the steps required for performing a state-based merge would be as follows: firstly the elements of the common ancestor for the versions to be merged are iterated. For each such element the matching function is called upon to determine the corresponding elements in each of the two versions being merged. Next the algorithm verifies if the matching elements have been modified in either of the two versions. Should this be the case for only one of the versions, then the modified element will simply be used in the final merged version. Should such a change exist in both of the artifact versions, then an update-update conflict is present. If an existing match is unchanged, then the element in question is left as is in the resulting merged version. If one of the elements in the ancestor has a match in one version but not in the other, it means that the element has been deleted in the version where no match is present. As long as there are no changes for the deleted element in the other version, the element can simply be removed in the merge result. On the other hand if the element has been changed in the matching version, a delete-update conflict is present. Elements in the common origin that have no match in either one of the versions are removed from the merge result, as they have been removed in both versions. Once all of the elements in the common ancestor have been processed, all of the unmatched elements from both versions get added to the merge result, as they have been newly introduced in one of the versions.

On the other hand, *operation-based merging* is being carried out by recording and analyzing the sequence of operations that have been carried out on the original version of an artifact. Since the editor used is also responsible for recording the operations, composite operations, like refactorings, can also be recorded. The high degree to which operation-based merging relies on the used editor also brings about some disadvantages, such as having to record and store information that is no longer relevant, such as all of the operations performed on an element that will ultimately be removed at a later time. Because of this, operation-based approaches will often employ cleansing algorithms to sanitize the list of recorded operations.

The steps required for an operation-based merge start with checking the operation sequences of the two versions in regard to commutativity to detect conflicts. Similarly to the match function for the state-based merging, some complex techniques also come into play here, within the decision procedure for commutativity. A simple, yet inefficient variant of this procedure would apply each pair of operations found in the cross product of the atomic operations in both sequences, to the artifact. This would be performed in both possible orders upon which the two results will be checked for equivalence. Should the results be equivalent, then commutativity is present between the operations, otherwise a conflict is present. Once the commutativity check has been carried out, all of the non-conflicting changes from both operation sequences are applied to the common ancestor, thus yielding a merged model.

When compared to their counterpart, operation-based approaches are more accurate and provide more information. They are also faster than state-based approaches when it comes to their run-time performance, and better suited for large models as the size of the model itself has no bearing on the computational effort required. Instead the length of the operation sequences being compared is the one factor that dictates the required effort to perform the merge. Their computational effort depends on the length of the operation sequences, and not on the size of the model itself.

Even though there is a difference between state-based and operation-based merging, the two are intertwined. Some state-based merging approaches do in fact derive the applied operations and subsequently employ operation-based conflict detection. The same holds the other way around, namely some operation-based approaches derive the states based on the operation sequence in order to detect inconsistencies after merging. The concepts employed for conflict detection are therefore similar between state- and operation-based approaches, as they both check for conflicts during the merge and illegal states afterwards.

2.2.2 Model versioning - steps

Once this classification has been made, Brosh et al. [BKL+12] systematically describe the five techniques required in order to achieve versioning support for models:

1. Model Driven Engineering
2. Model Transformation
3. Model Differencing
4. Conflict Handling
5. Merging

They firstly introduce **Model-Driven Engineering**, the goal of which is to allow developers to focus on non-trivial tasks while the translation from models to code is being done automatically. At this point the term metamodeling is introduced, and defined as the modeling of modeling languages.

In an attempt to standardize the key concepts and ensure interoperability between the various development environments, the Object Management Group, or OMG for short, has published the specification for Model Driven Architecture, or MDA [S^{+}00] for short. The aforementioned interoperability between development environments is achieved through the Meta-Object-Facility, or MOF [mof05] for short, as well as the later released MOF 2.0 version [mof14]. This layer of the metamodeling stack represents the only self-defined metamodel for building other metamodels, thus ensuring the interoperability of any metamodels defined within. The great advantage of MDA is the fact that it allows the specification to stay much closer to the problem domain, without being limited by a certain implementation. This is achieved by decoupling the system specifications from the underlying platform.

The Eclipse Modeling Framework, or EMF [SBMP08] for short, should be mentioned at this point, as it supports various subprojects for different MDA tasks such as building modeling editors, comparing models etc. This fact leads EMF to be increasingly more widespread in the field of academia as well as in practice.

The second technique required on the road to versioning support for models is **Model Transformation** [SK03]. A mechanism for synchronizing and transforming models is required in order to not have the developers perform error-prone and repetitive tasks, such as translating models into code. This mechanism is represented by the field of model transformation, and it is central to the model-driven engineering paradigm.

In order to accommodate the high variety of modeling languages, model transformation languages leverage the type-system introduced by metamodeling, thus allowing a syntax for transformations to be defined. The multitude of model application areas, and the need of model transformations to cover a high number of tasks, have led to the emergence of an equally high number of model transformation languages.

The types of transformation are introduced next, beginning with the *Endogenous model transformation*. This type of transformation takes place when both the target as well as the source model belong to the same metamodel. If a source model is being edited, the source will be the same as the target artifact. This is called an in-place transformation. Should a new target model be generated based on the source model, then an out-place transformation takes place. The Eclipse Modeling Framework can be named again at this point, as it allows for programmatic manipulation of models, thus providing support for the model transformations required for model versioning. *Exogenous model transformations* on the other hand, take place between models belonging to different metamodels. Models of a modeling language are used as input in order to create a new model of a different language. The third type of transformation is the *Model-to-text transformation*. These transformations are employed in order to obtain a textual representation of an input model, be it either as code or anything else.

Model Differencing is the next building block towards versioning models, and it can be achieved as follows. Firstly a match is computed between the elements of the two versions, then the differences themselves are obtained and finally they have to be represented for tasks such as merging and conflict detection. Matching is responsible for determining the identity of the model elements. The approaches employed to achieve this have their roots in schema matching for databases and ontology matching in the domain of knowledge representation. Database schema matching touches on various topics such as data extraction, semantic query processing etc.

This in turn has led to the emergence of multiple structures and terminologies which Rahm and Bernstein [RB01] classify in two categories: *individual matchers* and *combining matchers*. Furthermore matchers can be classified based on the level they operate on, such as element or structure level, on the cardinality of the matches, such as one-to-one, one-to-many or many-to-many and on whether the matcher is executed alone or makes use of the results provided by a number of matchers that have been independently executed. These same classifications can also be applied when describing model matching approaches.

Kolovos et al. [KDRPP09] propose a classification that is custom tailored to model matching approaches. This classification is based on the matching criteria for model elements, such as: identity-based matching, signature-based matching, similarity-based matching and matching that is specific to a certain language. *Identity-based matching* makes use of the model elements' UUIDs while the *signature-based matching* relies on a computed signature for each model element. This signature is based on a combination of the element's values, with these 2 types of matching offering a binary result to the matching problem of two model elements, namely they either are a match or they are not. *Similarity-based matching* on the other hand performs a computation based on the value of the model element's features and thus allows for insignificant attributes to be ignored by attaching weights to them. The fourth type of matching, relies on language-specific, user-defined match rules.

When it comes to concrete matching approaches there are a number of alternatives out there. Alanen and Porres [Por05] have developed an algorithm that achieves this, granted only for UML models, but this could have been extended with relative ease as its match function makes use of static identifiers. UMLDiff [XS05] on the other hand, is designed for a specific modeling language. Instead of using static identifiers it employs similarity-based matching, based on an element's name and structure.

Specifically tailored to UML is also the approach developed by Nejati et al. [NSC$^+$07] as well as ADAMS [DLFST09], with the latter leveraging a hybrid matcher that combines identity-based matching and signature based matching.

Not limited to UML are DSMDiff [LGJ07] as well as EMF Compare [BP08], with the former computing a signature based on the element type and name and subsequently taking into account the relationships between all of the previously matched model elements. EMF Compare operates in a similar fashion by employing four different metrics, namely the name, content and type of an element, as well as its relations to the other elements in the model. While it does also offer the possibility for identity-based matching, one has to choose between either identity-based or similarity-based matching, as both of these strategies cannot be combined in the case of EMF Compare.

SiDiff [SG08] is another example of a concrete matching approach, one that does allow for fine tuning of the comparison process (by attaching weights to individual model element attributes), unlike EMF Compare and DSMDiff which require no language-specific configuration.

Once the matching of the elements has been performed it is time to compute and represent the differences, should any exist. The differences themselves can be computed on three different levels, namely the abstract syntax, the concrete syntax and the semantics of the model. Differencing based on the *abstract syntax* is only able to identify differences in the syntactic data. *Concrete syntax differencing* is capable of detecting additional changes, such as layouting in the visualization of the model. *Semantic differencing* on the other hand compares the meaning of models instead of their syntactic representation. Out of the three aforementioned differencing levels, the one that is most relevant to model versioning is the abstract syntax level.

Most of the existing approaches in terms of model differencing are only able to identify atomic operations such as additions, deletions, moves and updates. In addition to these atomic operations, composite operations might also be applied to models. An example of a composite operation would be a refactoring.

In order for the identified differences to be of any use to developers they have to be represented in some way. Different approaches to this exist, with Cicchetti et al. [CDRP07] proposing a list of properties which should be fulfilled by an operation representation. The more important properties describe if a representation is model-based, meaning if it adheres to a metamodel used for computing differences, if it is applicable to compared models and if it is metamodel independent.

When it comes to concrete implementations, DSMDiff[LGJ07] distinguishes between elements to be added, changed or deleted, while SiDiff [SG08] distinguishes between structural, attribute, move, and reference differences. Representations of custom, language specific operations also exist, for example move activity and delete fragment, defined by Gerth et al. [GKLE10] for state machines. EMF Compare [BP08], unlike the aforementioned approaches, uses a model-based representation of differences.

Once the model differences have been computed **conflicts** might be detected. There are a number of definitions for the term "conflict" but in the field of software versioning it is used to describe operations that are contradicting each other.

Depending on the specific operations that have been applied to the two model versions, different conflict types might arise. Mens [Men02] distinguishes between textual, syntactic, semantic and structural conflicts. Structural conflicts arise as a result of a refactoring, leaving the merging algorithm unable to decide how the merge result should be refactored.

According to Brosch et al. [BKL$^+$12] no categorization of merge conflicts has been widely accepted in the field of model versioning. However Taentzer et al. [TELW14] do define a series of conflicts, while taking into account the additional features of EMF-based models. These conflict types are as follows: delete-use, delete-move, delete-update, update-update, move-move, and insert-insert.

The last step required, in order to achieve versioning support for models, according to Brosch et al. [BKL$^+$12] is **merging**. The process of merging non-conflicting changes is a fairly uncomplicated one. In the case of a two-way merge, where deletions cannot be detected, the merged version is constructed as a "joint union of both input artifacts".

This process is slightly different and more reliable in case of a three-way merge where the two versions to be merged share a common ancestor. In this case, the merged version of the two artifacts is obtained by "applying the union of all changes detected between the common ancestor and both revised versions to the common ancestor version". Should a conflict arise however, then no unique merged version can be constructed. This means that conflicts need to be resolved before a merge can be concluded. The most straightforward way of solving conflicts would be manually. This means that the user is responsible for analyzing the two versions and decide which changes he or she wishes to be integrated in the final merged version. This approach, which is also favored by tools like Subversion [sub21] or Git [git21a], works well for line-oriented artifacts where dependent changes are located in a sequence, close together. The same does not hold true for models, whose graph based structure means that such related changes might be scattered across the model.

A visual, graphical side-by-side comparison of the two versions would still not be enough, as the effort required to identify matching elements increases proportionally with the canvas size, which in turn depends on the size of the model itself. To this end, Gerth et al. [GKLE13] have proposed a guided conflict resolution. Regardless of the specific means through which it is achieved, manual conflict resolution will always be error-prone due to the human factor. As a response to this fact Munson and Dewan present a framework that automates this process based on configurable merge policies[MD94]. Such merge policies might for example stipulate that the changes of a specific user might take precedence over to ones of another, or that certain operations outrank others.

It is in this context that Brosch et al. bring up conflict tolerance as opposed to merely conflict resolution. They argue that inconsistencies shouldn't be regarded as merely an undesirable byproduct of collaborative development. According to Nuseibeh et al. [NER01], inconsistencies should be at least temporarily tolerated as they help identify areas of the system that might warrant further analysis. These may be areas where the developers have a diverging understanding of the problem, or simply areas that are too large to be changed at once. Balzer [Bal91] for example relaxes consistency constraints and instead annotates the inconsistent areas with so called pollution markers.

None of the aforementioned conflict handling approaches, be it manual resolution, automatic resolution or conflict tolerance, are a clearly superior alternative to the other ones. Instead they each offer their own advantages and drawbacks. While manual merging requires more effort, it also provides the user with a higher level of control over the final merge result. Automatic conflict resolution on the other hand, reduces the required effort on the user's part but it also limits the user's control over the process. Lastly conflict tolerance offers insight into potential problem areas of the system but not without a cost, as it requires dedicated editors and an artifact grammar that supports pollution markers.

Upon introducing the aforementioned concepts, Brosch et al. [BKL$^+$12] define a set of criteria for evaluating state of the art model versioning systems. Among these criteria are the detection of conflicts between both atomic and composite operations, the detection of inconsistencies as well as the adaptability of the resolution strategies. This thesis will however focus on only a subset of these features, namely the graphical visualization of conflicts, the flexibility regarding the modeling language as well as the flexibility regarding the modeling editor.

Out of the fourteen approaches that they have analyzed, 6 of them offer modeling language and editor flexibility, while only 4 of them offer graphical visualization during the conflict resolution. The first approach that is relevant to the topic of this thesis would be CoObRa, a versioning framework created by Schneider et al. [SZN04] for the CASE UML tool Fujaba [fuj21], with *CASE* standing for computer-aided software engineering and *Fujaba* being an acronym for "From UML to Java and back again". It supports graphical conflict resolution but, perhaps as the name suggests, it does not offer any flexibility regarding the modeling language or the editor. Furthermore this project is not being actively developed anymore at the time of this writing.

The next contender on the list would be the approach by Mehra et al [MGH05]. They provide a plugin for the Pounamu CASE tool. This plugin offers graphical visualization of differences and offers flexibility regarding both the editor and the modeling language, as the diagrams are serialized in XMI before being converted into a graph representation for the actual comparison.

Another option offering visualization and editor flexibility is the IBM Rational Software Architect or RSA [ibm21] for short. It is a UML modeling environment based on the Eclipse Modeling Framework. The downside to RSA is the fact that it is limited to two languages, namely C++ and Java EE.

The semantically enhanced model version control system SMoVer [RAB$^+$07] is a candidate that supports no means of visualization but does in turn offer flexibility when it comes to the modeling editor and language used. In spite of this language independence, language specific semantic views can be implemented in order to adapt the system to a specific modeling language. The identification of differences is based on UUIDs, thus rendering the system independent from the modeling editor. This genericity of the differencing does however mean that it can not be adapted to a certain modeling language.

A solely theoretical work, with no implementation, by Westfechtel [Wes10] offers flexibility in the choice of language and editor similar to how SMoVer does, while similarly offering no visualization means. This approach does however focus on conflict-detection and assumes that the differences are obtained by EMF Compare.

This brings us to the open-source model comparison framework EMF Compare [Tou06]. While it does not offer any visualization means for the conflict resolution in and of itself it does provide algorithms for both two- and three-way merging. Furthermore EMF Compare also provides merging mechanisms coupled with conflict detection capabilities. All of these features are independent from any editor or language used, but allow for programmatic extension so that the end result would be custom tailored to a specific language.

An approach that builds on the previously described concepts and frameworks is AMOR [amo], an adaptable model versioning framework developed jointly by the Vienna University of Technology [tuw21], the Johannes Kepler University Linz [jku21], and SparxSystems [spa21]. The goal of AMOR, as stated by Brosch et al. is "to combine the advantages of both generic and language-specific model versioning by providing a generic, yet adaptable model versioning framework". This means that generic versioning support is being provided without any dependency to the used language or editor. In order to achieve this, AMOR leverages the Eclipse Modeling Framework. The design of their solution revolves around a series of requirements, such as, but not limited to, a user-friendly visualization of conflicts, an integrated view that showcases all the merge-relevant information in a single diagram, and a model-based representation that ensures the merge information is being represented as model elements.

In tune with the aforementioned aim for a user-friendly visualization AMOR generates a dedicated model versioning profile, containing difference and conflict models, that glues available information into the model. This versioning profile provides different stereotypes for the different change types. They distinguish between "Add", "Delete", "Update", "Move" and "CompositeChange". Furthermore each of these stereotypes is then assigned into one of two categories, namely "MyChange" and "TheirChange", in order to provide the user with a clear understanding of who performed the change in question. Conflict-specific stereotypes, such as "UpdateUpdate" or "DeleteUse" are also defined in the versioning profile, one for each conflict pattern of the conflict metamodel.

The conflict resolution itself is based on the tentative merge and allows for easy implementation of actions such as "take my change", "take their change" or "revert this change", thanks to the included stereotypes. AMOR takes conflict resolution one step further and implements a recommender system for conflict resolution, as certain types of conflicts are likely to reoccur.

As presented in this chapter, online IDEs offer virtually no support for model or diagram diffing and even traditional, non-cloud-based, tools offer only a limited combination of editor and language independence coupled with visual mechanisms for conflict resolution. One of the aforementioned candidates has however proven itself to be promising as it checks most of the boxes required by this work. This candidate is EMF Compare. It offers diffing and merging capabilities, coupled with conflict detection and the possibility to leverage all of these features in a programmatic fashion. This means that all of the information obtained from EMF Compare during a comparison can be serialized and shipped off to another component for further analysis and/or processing.

Once this is done, a programmatic call can be made to trigger a merge, resolve a conflict or perform a new comparison.

CHAPTER 3

Design and Implementation

This chapter will provide a brief introduction to the existing components that are being reused together with an overview of the system architecture. Subsequently the implementation will be discussed, while also going into more detail regarding components such as Theia, Sprotty or EMF Compare and the specific functionality that is being reused. The newly implemented components, namely the Diff-merge Theia extension representing the frontend, as well as the Diff-merge backend, will be presented alongside the reasoning that led to the final implementation of said components.

3.1 Architecture description

In order to achieve the goal of adding graphical diffing and merging functionality to the Theia IDE, a number of existing technologies will be used, some of them will be extended, and certain parts will be designed and developed from scratch. This chapter aims to offer an overview of the tool's architecture, as shown in figure 3.1, as well as briefly explain the role each of the components play in the graphical diffing and merging process.

Perhaps the best place to start would be the Theia IDE itself. Theia is an IDE platform that is easily extensible due to the modular way in which it has been designed. This means that even though diagramming support is not available straight out of the box as it were, it can easily be added through an extension. For the scope of this thesis, the diagramming support has been added to Theia in the form of the GLSP client integration and the Sprotty [spr21c] open-source diagramming framework. While Sprotty is responsible for rendering the diagram shapes, the GLSP client offers the necessary support for the Workflow Modeling Language [gls21a].

This language was developed by EclipseSource [ecl21] and is an executable and interpretable language that is meant to describe workflows within a system, such as the process of brewing coffee executed by a coffee machine. To support this language, a GLSP backend is needed to provide all of the language specific logic, without adding to the complexity of the IDE itself. The resulting diagrams have sufficient complexity to render them suitable for showcasing the graphical diffing and merging capabilities developed within this work.

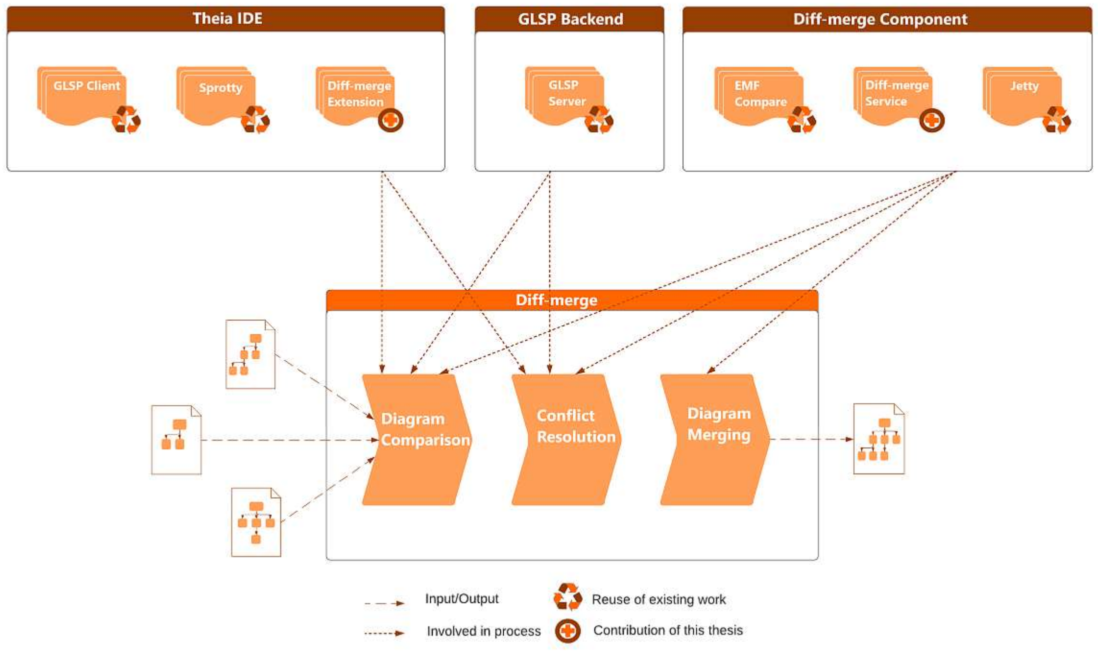


Figure 3.1: Architecture Diagram

This means that one can generate an instance of the Theia IDE that contains diagram editing capabilities by re-using existing work. Now the only thing missing is the Diff-Merge functionality. This is added by once again making use of the ability to create custom Theia extensions. This Diff-Merge Extension is responsible for displaying the differences between two/three versions of a diagram and allows the user to perform merges and resolve conflicts. In short, the extension adds new UI elements and user interaction means to the Theia IDE.

While the Diff-Merge Extension is handling the client-side of the equation, the heavy lifting is being performed by the backend Diff-Merge Component. This component is responsible for performing the diffing, merging, and conflict resolution based on the input from the client. Once these operations are performed, the results are sent to the client to be displayed to the user. The communication between the client and backend takes place through REST requests, handled by a Jetty [jet] server which is included in the

Diff-Merge Component. These requests are then forwarded to the Diff-Merge Service, which has been developed during the course of this work. This service is responsible for interpreting the commands received from the client and calling the appropriate functions in EMF Compare. EMF Compare is being programmatically called upon to perform diffing, conflict detection and merging. All of the aforementioned operations can be performed on the Workflow Modeling Language, which again renders it an appropriate candidate to showcase how graphical diffing and merging capabilities could look like in a cloud-based environment.

The results obtained from EMF Compare are then processed by the Diff-Merge Service so that they may subsequently be sent back to the Diff-Merge IDE Extension to be displayed.

3.2 Implementation details

This chapter presents the implementation of the diffing and merging functionality, spanning from the UI in the frontend to the programmatic calls to EMF Compare in the backend. We begin by offering an introduction to all of the technologies used within this thesis, with the more relevant aspects being described in more detail. Moving forward we will describe the implementation of the graphical diffing and merging functionality as a pair consisting of a Theia extension and a Java backend, as well as the protocol developed in order to facilitate the communication between the two.

3.2.1 Theia Diff-Merge extension

Theia IDE

In order to offer a better understanding of what has been implemented and why, a closer look at Theia's inner workings is in order. Theia is an open-source framework that facilitates the creation of both web-based IDEs, as well as classical native desktop versions thereof. In order to achieve this using the same source, Theia uses two processes, a frontend and a backend. These processes communicate with each other through JSON-RPC [jso21] messages exchanged through Websockets, or through RESTful APIs [res21] via HTTP. In case of the desktop application, both the backend and the frontend run locally, while in case of a web-based application the backend would run on a remote host. Both processes have a dependency injection container that allows for extension contributions.

There are two means through which functionality can be added to Theia, namely extensions and plugins. Theia extensions are npm [npm21] packages that expose any number of dependency injection modules that contribute to the creation of the dependency injection container. These extensions are being consumed by adding a dependency to the npm package in the package.json file of the application and allow developers to extend and customize existing Theia extensions such as the Monaco Editor [mon21].

Plugins on the other hand are more limited in their scope, meaning that developers have to stick to predefined APIs without any possibility of tweaking something that is outside of the scope of the available API. As an upside to this limitation, plugin code is being run in a separate process, thus not blocking the Theia core processes. Furthermore they can be loaded at runtime without requiring a recompilation of the entire IDE. However because of this limitation, a Theia plugin has proven itself to not be sufficient in order to add the graphical diffing and merging functionality to the IDE, an extension being required in order to achieve this.

Before the Diff-Merge extension itself can be developed, a version of the Theia IDE that offers diagramming support has to be created. This can be obtained through the Eclipse GLSP Examples repository [gls21a], which provides a working integration of GLSP and Theia with support for the Workflow Modeling Language. Concretely this provides the glue code required for the integration of diagram editors built on the graphical language server platform with the Theia IDE. In order to support the Workflow Modeling Language GLSP builds upon the web-based diagramming framework Sprotty.

Sprotty

The graphical language server is responsible for handling and storing all of the information regarding the diagram and the operations that can be performed upon it, while the client is only tasked with handling the information necessary to render the diagram. This visualization is performed through Sprotty, which relies on a reactive architecture that easily facilitates the arbitrary distribution of concerns between a client and a server [spr21a]. Furthermore this falls perfectly in line with the scenario of the Language Server Protocol, with Sprotty extending the language server, which constitutes the base around which GLSP is built, namely the separation between language logic and IDE integration.

Because GLSP builds upon the same principles, as well as the same communication protocol as Sprotty, it is only fitting that the latter be briefly introduced. This will offer a better understanding of how the graphical diffing and merging functionality has been implemented as well as offer some insight into why certain parts work the way they do. The diagram is stored in the so-called ***SModel***, a graph model that organizes the elements in a tree, constructed based on the properties "parent" and "children". All elements inherit from the *SModelElement* which has an unique string ID. Furthermore the root of the tree will always be represented by an instance of *SModelRoot* which contains an index of the model that allows for a fast element lookup based on their ID.

In order to allow for a client-server architecture to be built the aforementioned graph model has to be serializable. This JSON serialization of an *SModelElement* is called its schema and it uses the IDs of elements for cross-referencing. For example the schema of an *SEdge* would use the IDs of *SNodes* to reference its source and target, as show in the code fragment below.

```
//Serializable schema for SEdge.
export interface SEdgeSchema extends SModelElementSchema {
   sourceId: string
   targetId: string
   routerKind?: string;
   routingPoints?: Point[]
   selected?: boolean
   hoverFeedback?: boolean
   opacity?: number
}
```

Operations on the graph model are described by ***Actions***. They are serialized JSON objects that get exchanged between the client and server. Actions are sent through the *ActionDispatcher* and they can either originate in the *ModelSource* or in the *Viewer*. Upon receiving an action the *ActionDispatcher* converts it to a command by using the appropriate *ActionHandler*. These *Commands* describe the actual behavior of the operation and typically implement the following methods: *execute()*, *undo()* and *redo()*. Each of these methods take the current model as well as the command execution context as input parameters and either return the new model as a result or return a promise thereof. The interactions between the aforementioned components are depicted in figure 3.2 below.

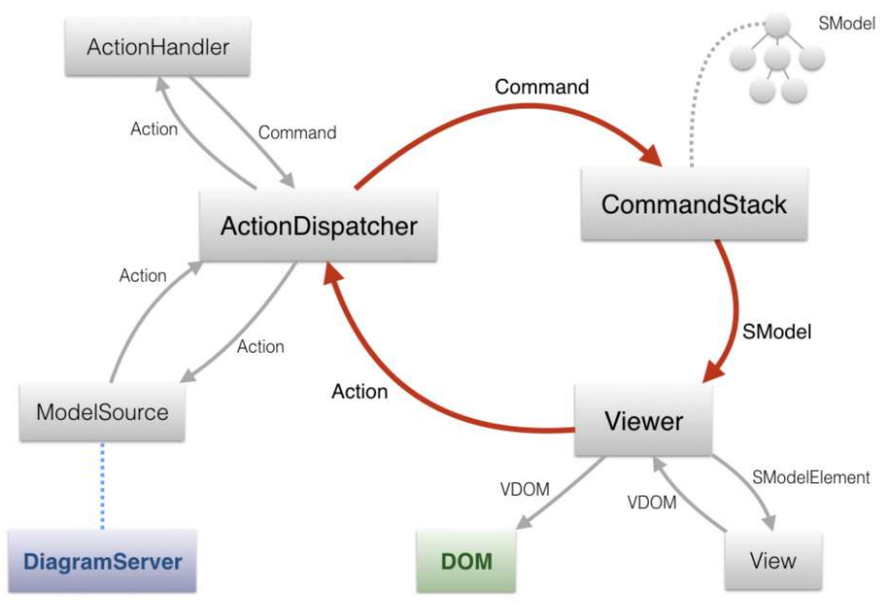


Figure 3.2: Sprotty Architectural Overview [spr21d]

GLSP

Now that the basic features of Sprotty have been briefly described, we can start introducing GLSP-specific features and thus further elaborate on the foundation upon which the Diff-Merge extension has been developed. In order for the client to display any sort of diagram, it has to be fetched from the server. This is achieved by the client sending a *RequestModelAction* to the server. The response to that request will either be a *SetModelAction* or an *UpdateModelAction* should the model have been subjected to any updates. The model received by the client will adhere to the GLSP metamodel, or more concretely in our case, to the Workflow Modeling Language metamodel, that is an extension of the GLSP metamodel.

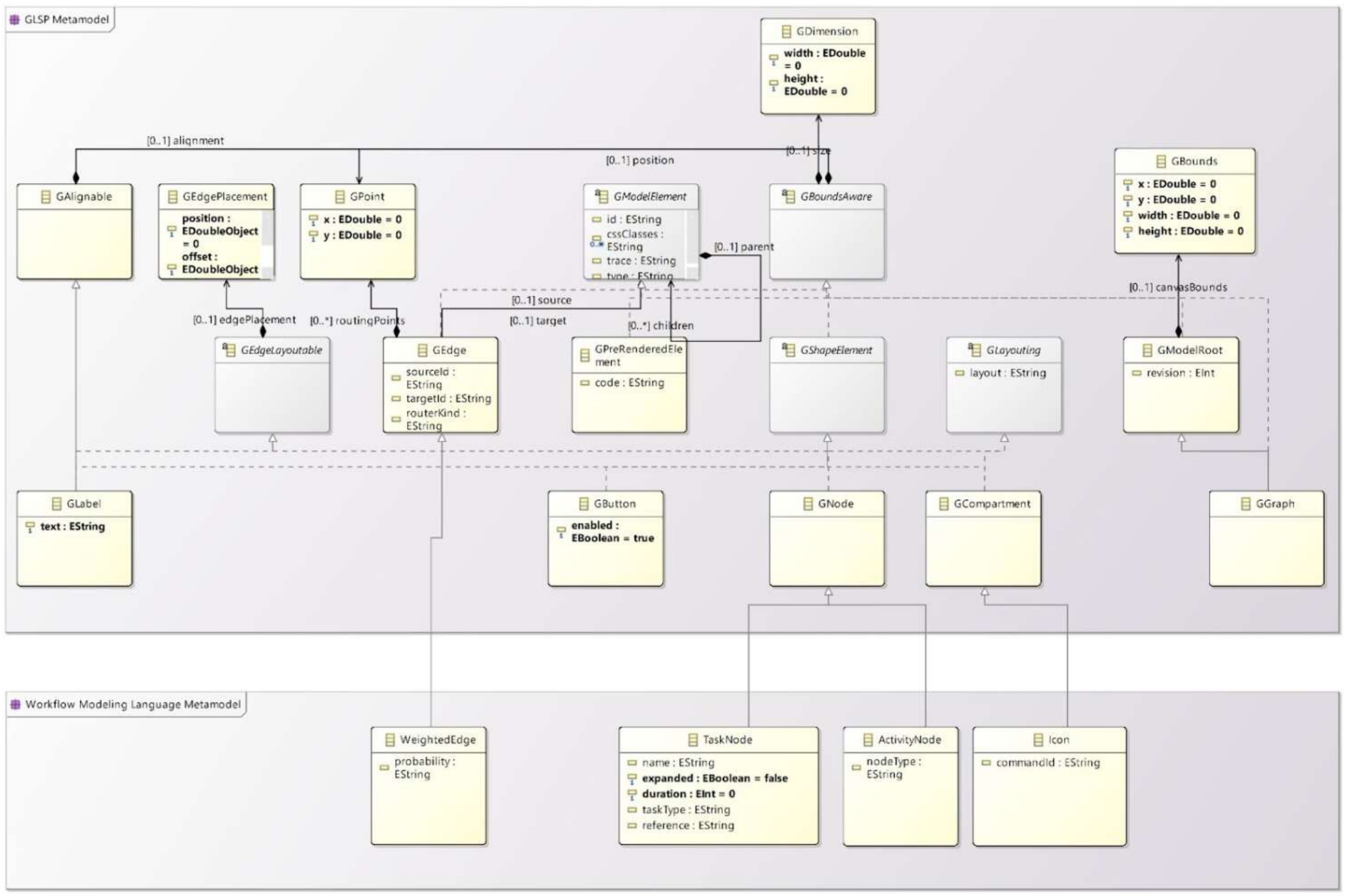


Figure 3.3: Simplified GLSP metamodel and Workflow Modeling Language metamodel

As illustrated by figure 3.3, and similarly to Sprotty, every model element will be a subtype of the *GModelElement*, with each of them in turn having between 0 and an undefined number of children. The model root is a *GGraph* in our case, which is a subtype of *GModelRoot*, and it contains the model itself, consisting of *TaskNodes*, *ActivityNodes*, Icons and *WeightedEdges*. Besides the language-specific extensions, model elements also contain data relevant to their visualization such as dimensions *(GDimension)*, bounds *(GBounds)*, edge placements *(GEdgePlacement)* etc.

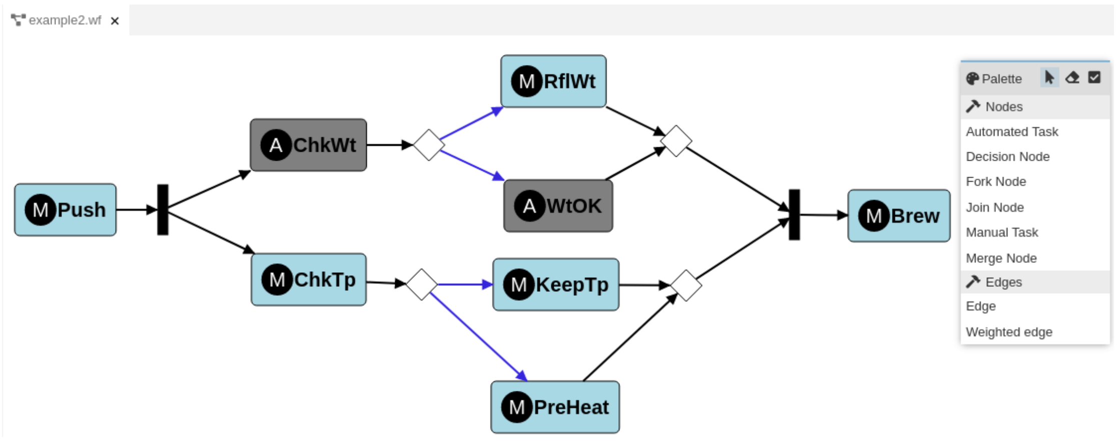


Figure 3.4: Graphical representation of a Workflow Modeling Language Diagram

Figure 3.4 on the other hand, depicts an example of a Diagram constructed by using the Workflow Modeling Language, or WFML for short. As also illustrated by the figure, in order to allow for the modeling of processes the WFML implements several concepts, which we will briefly describe in the following paragraphs. Perhaps one of the most important elements of the WFML is the **task**. As the name suggests, it is used to model a specific task that is to be executed at a certain point within the workflow. Tasks are represented as rectangles with rounded corners, with a label containing the name of the said task, as well as an icon describing its type. As such tasks will be either **manual**, which will be adorned with an "M" icon, or **automated**, which will be marked by the "A" icon. The distinction between the two types is further made clear by the background color of the task, namely blue denoting a manual task while gray is used to aid in the distinction of automated tasks.

In order to depict the transitions between the various diagram elements, **edges** are used. Similarly to the tasks, they too have a type, namely one can have **standard edges**, or **weighted edges**. While both types are depicted as lines ending in arrowheads, the visual distinction between the two is once again made based on the color, with the standard edges being black and the weighted ones being drawn in a blue color.

Additionally the WFML also supports so called **activity nodes** which aid in the coordination of the workflow from and to the other elements. There are four different types of activity nodes present in the WFML at the time of this writing. Each of them will be briefly described in order to paint a picture of what workflows could be depicted by using the tools at hand. **Fork nodes** are used to depict parts of the workflow that take place in parallel. They accept one incoming edge and multiple outgoing ones. Such nodes are depicted by a vertical bar that has one incoming edge and multiple outgoing ones. Once the parallel workflow segments have been completed they are joined back together by using a **join node**. Join nodes are used in conjunction with fork nodes and are represented similarly, if somewhat mirrored, namely they accept multiple incoming edges and have a single outgoing one. **Decision nodes** are used to choose a certain path of a possible workflow based on the certain guards. They are depicted by using a diamond symbol that accepts one incoming edge and multiple outgoing edges. As it was the case with fork and join nodes, decision nodes also have their, so to speak, counterpart, namely the **merge nodes**. Once the alternative workflow has been completed it needs to be rejoined again, so we will be once again faced with a diamond symbol. This time however, there will be multiple incoming edges, representing the possible alternatives the decision node has chosen from, and only one outgoing edge, depicting the common, singular workflow that follows.

WFML Example

Now that we have introduced and described the building blocks of the WFML we can start describing the example depicted in figure 3.4 in a more natural language and focus on the bigger picture, the depicted process, and on the interactions between the elements instead of regarding them individually. The initial step of the coffee making process illustrated in figure 3.4 is easily recognizable by its lack of incoming edges. The manual task of pushing a button, represented by the *"Push"* task, kicks off the entire process. Once the process has been initiated we encounter a fork node that branches off in two parallel workflows. The upper branch concerns itself with the level of water in the tank, while the lower one is concerned with the temperature. Starting on the top branch we reach the automated *"ChkWt"* task which is responsible for checking the water level. In order to achieve this it is connected to a decision node, which has one incoming edge and two outgoing ones, one for each possible result of the water level evaluation. The incoming edge is a standard one, while the outgoing ones are weighted edges, which are easily recognizable as such through the blue color in which they are rendered. Should the water level be too low, we will reach the manual task *"RflWt"*, which represents the process of filling the water tank back up with water. Should the water level prove to be high enough during the verification performed by the guard on the decision node, we will reach the automated task called *"WtOK"*.

Regardless which route is taken, we will end up at the merge node that merges the two alternative workflows back up again, easily identifiable by the two incoming edges and the singular outgoing edge of the diamond. From there we move on to the join node where the two parallel workflows meet back up again and continue as one. The join node is the mirror image of the fork node, with two incoming edges and one outgoing one.

Going back to the fork node and taking a look at the lower branch of the parallel workflow we reach the manual task *"ChkTp"* which, similarly to *"ChkWT"*, is responsible for checking the temperature of the coffee machine's heating element. This check leads us to a decision node, where once again we are faced with two alternative workflows. If the temperature is adequate, the manual task *"KeepTp"* will be reached, which, as the name would suggest, is responsible for maintaining the current temperature for the heating element. Should the temperature check conducted by the decision node reveal that the heating element's temperature is too low, the manual *"PreHeat"* task will be executed. During the execution of this task the heating element is turned on and its temperature rises, so that the coffee will actually come out of the machine hot instead of lukewarm.

Once we are done with either one of the two temperature related tasks we reach a merge node which, like the one in the upper branch of the diagram, leads to a join node where the parallel workflows come together again. From there, the only outgoing edge is followed and the manual *"Brew"* task is reached. This is the step during which coffee is actually brewed and poured into the recipient.

This example was constructed in order to showcase the capabilities of the WFML and does not necessarily accurately reflect the coffee brewing process that might be carried out by a real coffee maker. The WFML is complex enough to allow for meaningful examples to be crafted so that the intended meaning behind it is accurately conveyed to the user without having to overwhelm them with complex mechanisms and a steep learning curve for the language.

GLSP diagram editor

Now that we have discussed the WFML let us take a look at the context in which users are meant to interact with it, namely the **diagram editor** in which it is rendered. Through the integration of the GLSP new functionality is added to the Theia IDE, in the form of the GLSP diagram editor. This editor adds a series of diagram editing features to the IDE, with the most notable ones being briefly described in the following paragraphs. Upon double clicking a diagram in the project explorer, the diagram editor opens up. The diagram itself is rendered within a **canvas** that can be panned around and have its zoom level adjusted. Besides the elements of the diagram, the user is also presented with a **tool palette** that allows for the addition of diagram elements by selecting them from the list and clicking the canvas. In the case of nodes or tasks, the element will spawn at the clicked location. When it comes to the edges, these can only be added to an element that supports it, such as a task or a node, and not directly to the canvas or another edge. In order to aid the user in connecting edges in a valid way, the mouse pointer changes to either a cross for a valid edge endpoint or to a hand with a crossed out circle for invalid edge endpoints. In addition to adding elements to the canvas, the tool palette also permits the users to select an eraser tool that will delete any diagram element clicked, to enable diagram markers that provide additional information about elements and mark invalid ones, such as a merge node that has more than one outgoing edge, and finally the user can go back to the mouse pointer tool.

Within the canvas itself, diagram elements can be moved by clicking and dragging them around with the possibility to edit them through a double click. Upon hovering over an element a tooltip opens, containing information about that element, such as the type of a task or its duration. By right clicking, either on an element or in the canvas, a context menu is opened up, that allows the user to perform certain actions depending on where they have performed the right click. The functions offered by this menu are as follows:

- **New**: Allows the user to add a new task, manual or automated.
- **Paste**: Allows the user to paste content from the clipboard.
- **Go to**: Allows for navigation between markers and nodes as well as to the documentation.
- **Copy/Cut/Delete**: If one or more diagram elements is selected, the user can choose to copy, cut or delete them. These three options are greyed out should the context menu be opened without any elements selected.

Perhaps the first new addition to the IDE that gets noticed is the presence of a new entry on the menu bar, called ***"Diagram"***. This menu allows the user to interact with the diagram editor and perform layouting changes. Concretely one can execute the following actions through the ***"Diagram"*** menu:

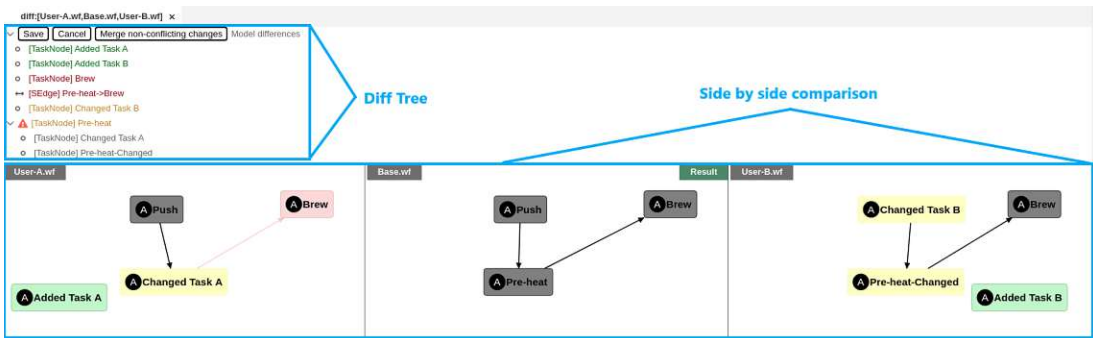
- **Center**: Aligns the diagram in the center of the canvas.
- **Export**: Saves a copy of the diagram, in the SVG format, in the same folder as the original one.
- **Fit to screen**: Adjusts the zoom level of the diagram view so that all the diagram elements fit within the visible region of the canvas.
- **Layout**: Aligns all of the diagram elements in a grid-like manner.
- **Align**: This entry has its own sub-menu, allowing the users to align diagram elements both in an absolute as well as a relative manner to the other diagram elements (e.g. align left or left of first/last selected element).
- **Resize**: Allows users to resize elements in a similar way to the alignment option, meaning they can choose between a minimum, maximum, and average width/height, or they can refer to the height or width of the first/last selected element.

With the exception of the *Align* and *Resize*, all other entries in the***"Diagram"*** menu can be accessed via keyboard shortcuts.

Representation and Navigation

As previously stated, this functionality provides a base that is sufficient in order for the graphical diffing and merging capabilities to be developed upon. When evaluating the state of the art in regard to merging and diffing we looked at how diffs are represented in the tools available, both when it comes to diagrams as well as text-based representations. Out of the myriad of tools integrating some form of diffing mechanisms, almost all of them offer the possibility of a **side by side comparison**, regardless of the type of comparison being carried out. This common denominator of sorts is of course also present in tools that have the ability to carry out model/diagram diffing, such as the Eclipse Papyrus [pap21] modeling environment. This type of visualization allows users to identify individual changes at a glance, while also being able to maintain an overview of the diagram, and thus perceive the changes in their context instead of individually and isolated. While this holds true for relatively simple diagrams, the situation changes slightly once the complexity of said diagrams increases. Especially when opening up multiple editors or viewers side by side, the screen real-estate can quickly be taken up and diagrams might expand beyond the area of the canvas that is visible on the screen.

On one hand this problem might be somewhat mitigated by adjusting the zoom level so that everything fits, but on the other hand this might quickly render the representation incomprehensible, especially in the case of complex diagrams. For example, elements in a diagram might be connected, either through edges between states, in case of the workflow modeling language, or some other means. So upon seeing a change, it might not be immediately clear to the user what consequences a certain modification to an element might entail within the diagram as a whole. In order to mitigate this issue additional means for showcasing the differences between the versions are required. Here is where a **diff-tree** comes in. A diff-tree is highly similar to the file or directory tree, that project or package explorer uses to list the folder structure, having the same representation but displaying differences instead of files. The purpose of adding this diff-tree to the representation is twofold, firstly it aggregates differences, that might otherwise be situated on opposing sides of the diagram, in a concise manner, thus offering the users a quick overview of what changes have transpired and secondly it helps users identify exactly where a specific change has occurred within a diagram by centering the side by side representation on the clicked change. Both the diff-tree and the side-by-side comparison are depicted in figure 3.5.


diff:[User-A.wf,Base.wf,User-B.wf]

- [TaskNode] Added Task A
- [TaskNode] Added Task B
- [TaskNode] Brew
- [SEdge] Pre-heat->Brew
- [TaskNode] Changed Task B
- [TaskNode] Pre-heat
  - [TaskNode] Changed Task A
  - [TaskNode] Pre-heat-Changed

Figure 3.5: Side-by-side Representation and diff-tree

This way users will be able to view the diagrams in their entirety, as well as an exhaustive list of changes, represented by the diff tree. Should they wish to inspect a specific change in more detail, all they would have to do is click on the change in the tree and the diagrams will get centered on the specific change, allowing them to also assess the semantic changes that might have occurred by observing all of the other diagram elements surrounding the specific change.

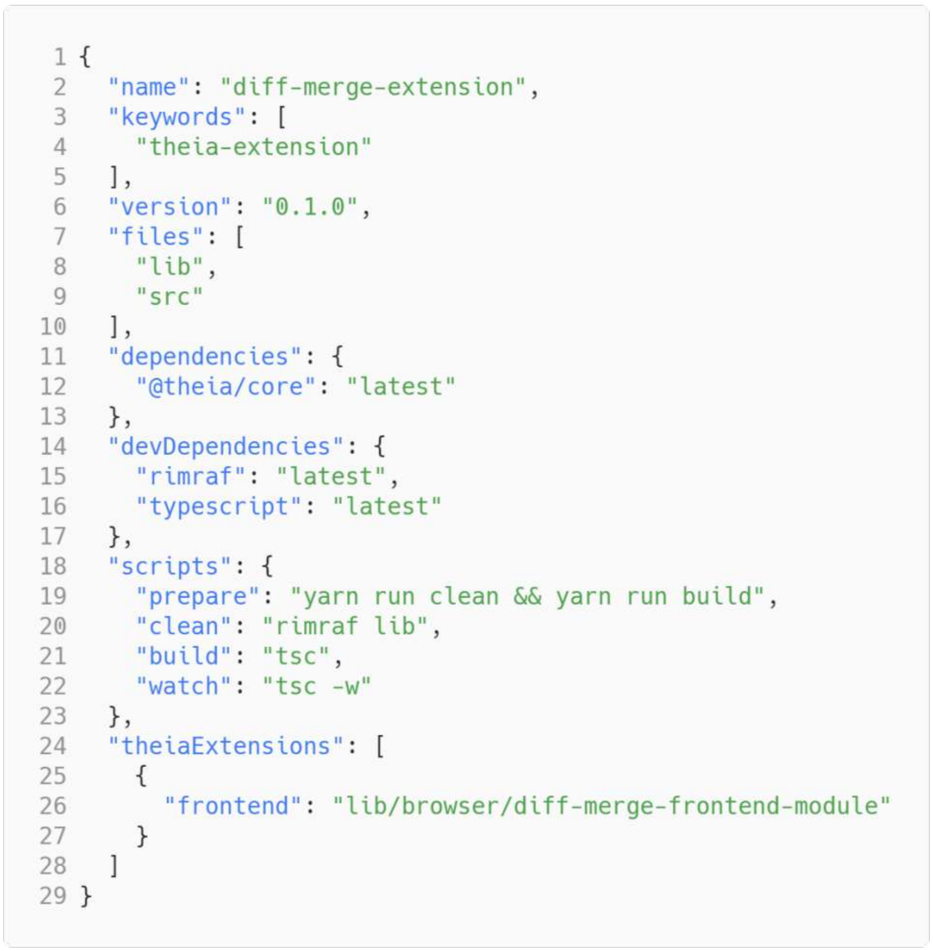
Theia Extension Structure

Once a suitable representation of the differences has been settled upon it was time to delve deeper into the specifics of Theia in order to ascertain how the aforementioned features can be implemented, what the possible limitations are, what can be reused, what can be adapted, and what has to be developed from scratch. As mentioned in the beginning of this chapter, a Theia extension is used in order to add the diffing and merging functionality to the IDE. One could author a Theia extension entirely from scratch, but luckily that is not necessary as there is a way to have the boilerplate code automatically generated by using the Eclipse Theia - Extension Generator [ext21]. This extension generator is a Yeoman [yeo21] based tool that guides the user through the generation of the scaffolding code, and allows them to choose from a few sample extensions that can be automatically generated and subsequently expanded upon:

- **hello-world**: A simple extension that contains a command and menu item which displays a message.
- **widget**: The basis for a widget, including a toggle command, an alert message and a button that displays a message.
- **labelprovider**: A simple extension that provides a custom label including an icon for a specific file type (".my" is used as an example file type).
- **tree-editor**: A simple tree-editor extension, including an example file, allowing users to interact with the JSON hierarchy of it and create and delete nodes
- **empty**: A minimal, empty extension containing a frontend module and a generic contribution.
- **backend**: An extension demonstrating how to communicate with backend services.

For this thesis the hello-world example extension template has been used as a starting point (renamed to "**diff-merge-extension**"), as it generates the bulk of the required boilerplate code without adding unnecessary features, such as superfluous menus or contributions, that would have to be removed later. Upon execution the extension generator generated the appropriate folder structure as well as some configuration files have been generated. Out of these files a few are of special interest, starting with the *package.json* file depicted in figure 3.7 below.

This file specifies the package's metadata as well as dependencies to the Theia core package as well as any additional dev dependencies that might be required.



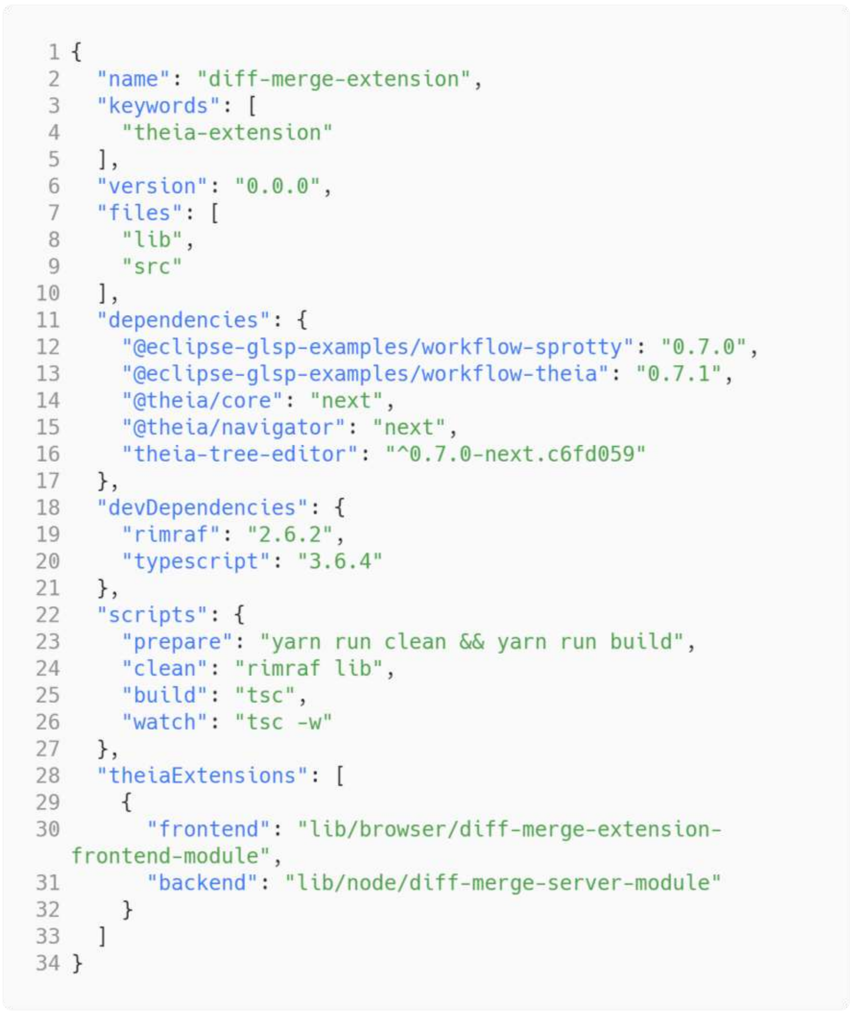
```
 1 {
 2   "name": "diff-merge-extension",
 3   "keywords": [
 4     "theia-extension"
 5   ],
 6   "version": "0.1.0",
 7   "files": [
 8     "lib",
 9     "src"
10   ],
11   "dependencies": {
12     "@theia/core": "latest"
13   },
14   "devDependencies": {
15     "rimraf": "latest",
16     "typescript": "latest"
17   },
18   "scripts": {
19     "prepare": "yarn run clean && yarn run build",
20     "clean": "rimraf lib",
21     "build": "tsc",
22     "watch": "tsc -w"
23   },
24   "theiaExtensions": [
25     {
26       "frontend": "lib/browser/diff-merge-frontend-module"
27     }
28   ]
29 }
```

Figure 3.6: package.json file for a newly generated extension

The more important elements of this file are perhaps the keywords on line 4, which allow the Theia app to accurately identify extensions as such and install them from the *npm* package registry. A little further along, starting on line 11, we have the dependencies block, which features all the requirements of the extension. Because this extension is freshly generated, its only requirement is the theia core app. In order to reach the aforementioned basis for development, the following packages have been added as dependencies:

- **@eclipse-glsp-examples/workflow-sprotty**: This package contains the configuration for the sprotty-glsp diagrams of the workflow example language.

- **@eclipse-glsp-examples/workflow-theia**: This package contains the glue code to integrate the GLSP Workflow example language into a Theia application.
- **@theia/navigator**: This package contains the file explorer widget, which can be used to easily view, open and manage files corresponding to a given workspace
- **theia-tree-editor**: This package contains a framework for building a tree master detail editor for editing model based data in Eclipse Theia.



```json
{
  "name": "diff-merge-extension",
  "keywords": [
    "theia-extension"
  ],
  "version": "0.0.0",
  "files": [
    "lib",
    "src"
  ],
  "dependencies": {
    "@eclipse-glsp-examples/workflow-sprotty": "0.7.0",
    "@eclipse-glsp-examples/workflow-theia": "0.7.1",
    "@theia/core": "next",
    "@theia/navigator": "next",
    "theia-tree-editor": "^0.7.0-next.c6fd059"
  },
  "devDependencies": {
    "rimraf": "2.6.2",
    "typescript": "3.6.4"
  },
  "scripts": {
    "prepare": "yarn run clean && yarn run build",
    "clean": "rimraf lib",
    "build": "tsc",
    "watch": "tsc -w"
  },
  "theiaExtensions": [
    {
      "frontend": "lib/browser/diff-merge-extension-
frontend-module",
      "backend": "lib/node/diff-merge-server-module"
    }
  ]
}
```

Figure 3.7: package.json file for the Diff-Merge Theia extension

Another relevant block of the *package.json* file is the theiaExtensions block, starting on line 24 in the originally generated file, which lists the JavaScript modules that export DI modules defining contribution bindings of the extension. For the generated example only a frontend capability was provided. In order to support the communication between the Theia IDE and the DiffMerge backend, a new entry has been added to the block. This defines a contribution to the node backend which will be responsible for making the REST requests to the DiffMerge backend, where the diffing and merging process will take place. Aside from the additions to the generated file, versions of some dependencies have also been fixed in order to avoid problems during the development process that might arise as a consequence of a new dependency version being released. The resulting package.json is depicted above, in figure 3.7.

The Theia instance resulting from the dependencies depicted in figure 3.7 contains all the building blocks required to develop the diffing and merging functionality. In order to gain a better understanding of what exactly needs to be implemented, scenarios which depict the workflow of the diffing and merging process were created, in a somewhat similar fashion to user stories. Subsequently these scenarios were broken down in key tasks that could be implemented and tested independently from one another. This offered both an overview of what the components of the extension should be, as well as how they should be interacting with each other. By dividing the diffing and merging processes into smaller sub-tasks it became clear how the information flow of the system needed to be structured, as well as which parts would have to communicate with the DiffMerge backend.

The identified key workflows were the diffing process and the merging process. The diffing process, where the user compares two versions of a diagram and views the differences between them has been divided up in the following sub-tasks:

- Selection of the files/versions to be compared
- Fetching the comparison from the DiffMerge backend
- Visualization of the differences
- Visualization of conflicts
- Interaction with the diagram (panning, zooming, etc.)

The merging process, as seen from the Theia side consists of the following sub-tasks:

- Merge/discard entire change sets
- Merge/discard individual changes
- Solve conflicts

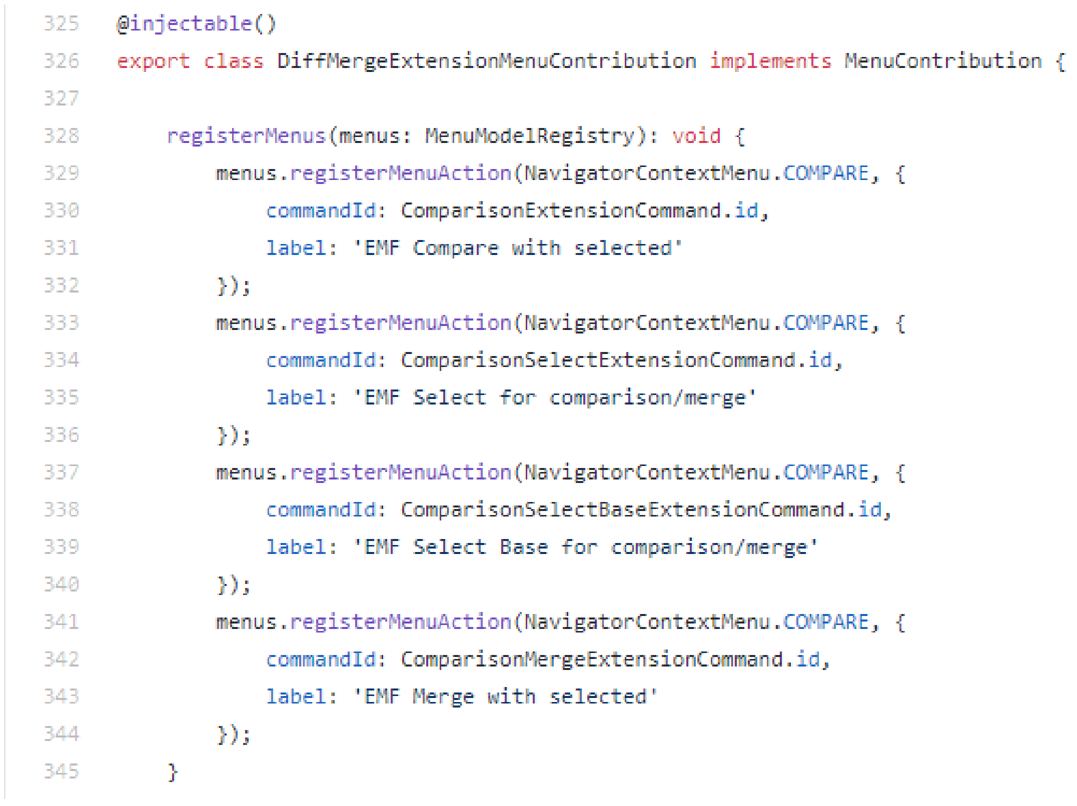
Functionality and Implementation

Perhaps the most appropriate way of approaching the implementation would be in a step by step manner, by going through the steps of the diffing and merging processes. So before a comparison between two diagram versions can be performed, the versions themselves have to be selected. In order to limit the complexity of the development environment and potential issues caused by adding a myriad of technologies to the mix we have decided to focus on the core goals of this thesis, namely the diffing and merging of diagrams. As such we have opted to forgo an integration between the *DiffMerge* extension and *Git* while the core functionality was still under development. Instead local files have been used to simulate the different versions of diagrams. This allowed us to test and debug the functionality faster, by merely performing changes in local files, simulating various edits to the diagrams, while also limiting the amount of variables that might cause problems with the diffing/merging process, such as an integration with Git, where the diffing/merging process would be triggered automatically upon pushing the changes to the origin. Furthermore the ability to trigger these processes arbitrarily allows for closer analysis and testing of corner cases, whereas a "production-ready" integration of the entire toolchain might inadvertently obfuscate potential issues that might arise. This approach has been adopted for both the two-way comparison / merge between two arbitrary versions of a diagram as well as the three-way comparison / merge between two versions of a diagram and their common ancestor.

In order to allow users to choose which files they want to compare entries have been added to the context menu of the file explorer. These entries and the actions they allow users to perform are as follows:

- **EMF Select for comparison/merge**: Marks a file as selected for a subsequent comparison or merge operation.
- **EMF Compare with selected**: Triggers a comparison between the previously selected file and the currently selected one.
- **EMF Select Base for comparison/merge**: Selects a file that will represent the common ancestor to be used in a subsequent three-way comparison/merge.
- **EMF Merge with selected**: Merges the non-conflicting changes of the currently selected file to the previously selected file(s).

These entries to the context menus have been added through a new Theia *MenuContribution*, as depicted by figure 3.8. The *DiffMergeExtensionMenuContribution* registers multiple menus, each with their own *MenuPath* and *MenuAction*. The *MenuPath* describes the location where the new entry should be added, in our case in the context menu of the navigator, next to the pre-existing text based comparison option, as indicated by *NavigatorContextMenu.COMPARE*. The second argument represents the action associated with the menu entry, which consists of a mandatory command ID, as well as some optional attributes such as *label*, *icon*, *order*, *alt*, etc.



```
@injectable()
export class DiffMergeExtensionMenuContribution implements MenuContribution {

    registerMenus(menus: MenuModelRegistry): void {
        menus.registerMenuAction(NavigatorContextMenu.COMPARE, {
            commandId: ComparisonExtensionCommand.id,
            label: 'EMF Compare with selected'
        });
        menus.registerMenuAction(NavigatorContextMenu.COMPARE, {
            commandId: ComparisonSelectExtensionCommand.id,
            label: 'EMF Select for comparison/merge'
        });
        menus.registerMenuAction(NavigatorContextMenu.COMPARE, {
            commandId: ComparisonSelectBaseExtensionCommand.id,
            label: 'EMF Select Base for comparison/merge'
        });
        menus.registerMenuAction(NavigatorContextMenu.COMPARE, {
            commandId: ComparisonMergeExtensionCommand.id,
            label: 'EMF Merge with selected'
        });
    }
```

Figure 3.8: Context menu additions for diffing/merging functionality

Once the entries have been added to the menu, the handlers for the commands had to be added to the *CommandRegistry*. The code snippet below illustrates the command handler for the selection of the base file for the comparison. It has been defined in-line and it sets the *baseComparisonFile* variable to the URI of the currently selected file, which is also the file for which the context menu has been called. Additionally it uses the automatically injected message service to display a message to the user, letting them know that the base file for the comparison has been selected. The handler for selecting the first file is highly similar to the aforementioned one, with the handler for the comparison execution being more complex and encompassing a number of the previously listed sub-tasks of the diffing process.

```
// Code fragment for registering the command for base selection
 registry.registerCommand(ComparisonSelectBaseExtensionCommand, {
          execute: async () => {
             this.baseComparisonFile =
                 UriSelection.getUri(this.selectionService.selection);
             this.messageService.info("Selected base file");
          }
      });
```

Now that the desired files have been selected it is time to ask the DiffMerge backend to compare them and deliver the result back to the DiffMerge Theia extension where it will be displayed to the user. The component responsible for handling the communication with the DiffMerge backend is the *ComparisonService*. This component is incidentally also the contribution to the Theia node backend that has been added on line 31 in the package.json file depicted in figure 3.7. The *ComparisonService* is an interface that will be automatically injected at run-time, whose implementation is responsible for making calls to the REST API endpoints exposed by the DiffMerge backend, an excerpt of which can be found below.

```
// Code fragment for requesting a three way comparison from the
    DiffMerge backend
async getThreeWayComparisonResult(basePath: string, file1Path:
    string, file2Path: string): Promise<ComparisonDto> {
   const resp =
      fetch('http://localhost:8080/diff/compareThreeWay/diagram?base='
      + basePath + '&file1=' + file1Path + '&file2=' + file2Path)
      .then((res: { json: () => void; }) => res.json())
      .catch((error: any) => {
         console.error('There has been a problem with your fetch
            operation:', error);
      });
   return resp;
}
```

As showcased by the code fragment above, the implementation of the *ComparisonService* is merely calling upon the node-fetch module [nod21] to perform a get request with parameters, and return the result. This implementation is then used as a singleton to bind the *ComparisonService* interface. The binding, depicted in the code fragment below, takes place in the InversifyJS container module [inv21] that has been declared as the backend entry (diff-merge-server-module) in the *theiaExtensions* block of the package.json file.

```
//DI Container module containing the ComparisonService bindings
export default new ContainerModule(bind => {
   bind(ComparisonServiceImpl).toSelf().inSingletonScope();
   bind(ComparisonService).toService(ComparisonServiceImpl);
   bind(ConnectionHandler).toDynamicValue(context => new
      JsonRpcConnectionHandler(ComparisonServicePath, () =>
      context.container.get(ComparisonService))).inSingletonScope();
});
```

At this point, the user has chosen the files they wish to compare, the comparison request has been sent to the DiffMerge backend, and assuming all the validation checks passed, the result of that comparison has been delivered back to the DiffMerge Theia extension as a *ComparisonDto* that now needs to be appropriately visualized. This *ComparisonDto* mirrors the structure of the *Comparison* object returned by EMF Compare to some extent, in that it contains an array of matches which in turn contain sub-matches which eventually contain the diffs themselves, as well as the corresponding diagram element in the left, right and base file.

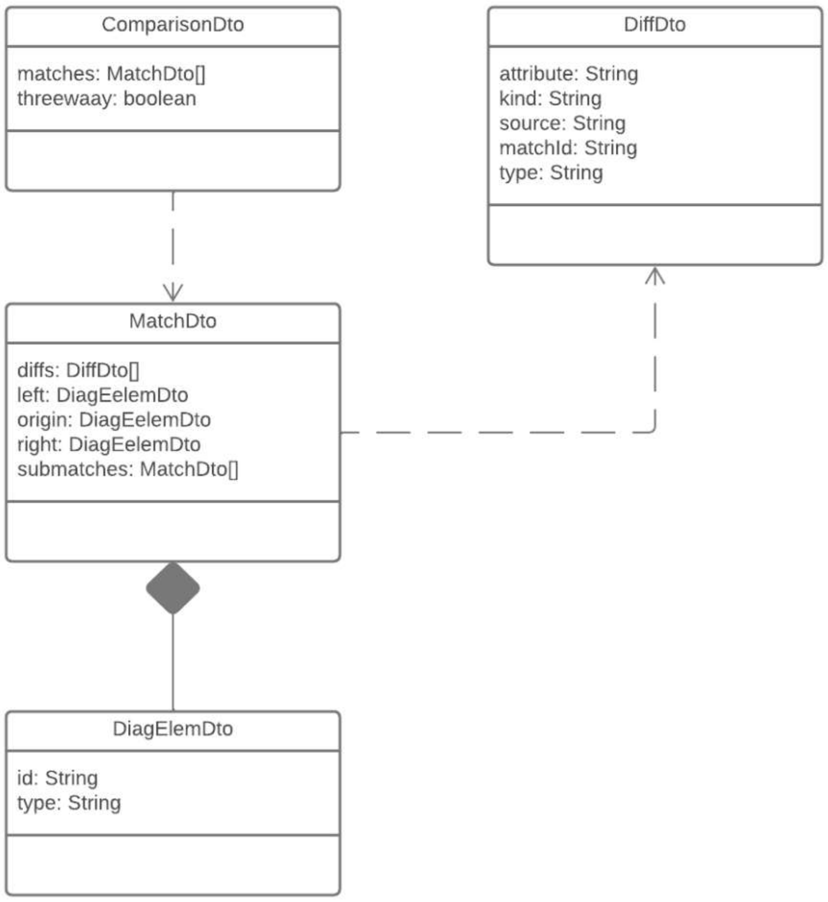


Figure 3.9: ComparisonDto Structure

In order to wrap up the diffing process, we have to circle back to the handler for the *ComparisonExtensionCommand*, which is responsible for triggering a number of sub-tasks involved in the diffing process. Firstly it checks to ensure at least enough files for a two-way comparison have been selected, before calling upon the *ComparisonService* to pass the request on to the DiffMerge backend and return the resulting *ComparisonDto*.

As shown in figure 3.9, the structure of the *ComparisonDto* is relatively simple, containing only an array of matches and a *threeWay* flag that will be relevant for displaying the diffs as it allows for the appropriate UI elements to be set up and initialized. The *MatchDto* type is used to represent the matches between elements of the different versions being compared. As such it contains entries for the possible locations where a match was found, namely any of the two diagram versions, denoted here by the left and right attributes, as well as the common ancestor, origin. These entries of type *DiagElementDto* make it possible to accurately identify the version(s) affected by a change and/or a potential conflict. In the context of this thesis the *DiagElementDtos* are regarded as atomic units, which will not be further split into sub-elements. How the granularity can be adjusted will be discussed in detail in the section describing the DiffMerge backend. Furthermore a *MatchDto* may have an arbitrary number of sub-matches which are used to represent nested elements within the diagram, as well as a list of actual differences. The *DiffDto* class represents a difference, such as an addition, change or deletion of an element in a diagram. Information like the source of the change and its kind are all stored within the *DiffDto* and will be relied on when it comes to displaying the changes.

In order to actually display any of the information contained in the *ComparisonDto* an appropriate environment has to be created. There are three types of components involved in making up this environment:

- **DiffMergeDiagWidget**: A diagram widget, capable of displaying a diagram and rendering differences within it.
- **DiffTreeWidget**: A tree widget, used to render the diff tree.
- **DiffSplitPanel**: A container of sorts, used to house the widgets that contain all the relevant information, such as the diff tree and the diagrams themselves.

In order to display the diagrams the *DiagramWidget* used to display WFML diagrams has been extended and reused. The **DiffMergeDiagWidget** is an extension of the aforementioned *DiagramWidget* which allows the tool palette to be disabled, as it serves no purpose in the diffing and merging process. Furthermore, depending on the screen resolution, the tool palette even has the potential of cluttering the UI and covering diagram elements, thus making the process more cumbersome for the users. In addition to that, the extended widgets also take care of various widget initialization steps, including centering the diagram in the middle of the widget once it has been loaded.

As a base for the diff tree, the *TreeWidget* from the Theia core package has been used. Out of the box this widget allows for handling of selected elements, searching, and styling of the tree elements through the Decorator support that is included. Building upon this, the **DiffTreeWidget** has come into being. Unlike the extension to the diagram widgets, which mainly dealt with minor customizations, the *DiffTreeWidget* adds quite a bit more to its existing counterpart. In order to allow for an interaction with the tree widget to trigger any action in any of the diagram widgets, the *DiffTreeWidget* needs a reference to the two or three diagram widgets. As a result, these references have been added. These references allow for behaviors such as centering the diagram widgets on a specific change that has been clicked in the tree. Additionally, since the final diff tree would group the changes by their type, we need to be able to distinguish between additions, deletions and changes and thus three arrays of *DiffTreeNodes* have been added, together with the actual comparison object received from the backend, resulting in the structure depicted by the code fragment below.

```
//Basic structure of the DiffViewWidget
@injectable()
export class DiffTreeWidget extends TreeWidget {
   public comparison: ComparisonDto;
   private baseWidget: DiffMergeDiagWidget;
   private firstWidget: DiffMergeDiagWidget;
   private secondWidget: DiffMergeDiagWidget;
   private _additions: DiffTreeNode[];
   private _changes: DiffTreeNode[];
   private _deletions: DiffTreeNode[];

   setDiagWidgets(comparison: ComparisonDto, baseWidget:
      DiffMergeDiagWidget, firstWidget: DiffMergeDiagWidget,
      secondWidget?: DiffMergeDiagWidget) {
      this.comparison = comparison;
      this.baseWidget = baseWidget;
      this.firstWidget = firstWidget;
      if (secondWidget) { this.secondWidget = secondWidget;}
   }
   ...
}
```

The aforementioned decorator allows the change of the look and style of the tree items within a widget. The type of decorator to be applied is determined based on the three *DiffTreeNode* arrays, each type of change receiving its own color coding via a decorator, with additions being marked in green, changes in yellow and deletions in red. This is achieved through a newly implemented *DiffTreeDecorator*. This class implements the *TreeDecorator* interface present in the Theia core. In order to invoke the custom decorator on the diff tree, it has to be bound in the dependency injection container of the Theia Diff-Merge extension, as shown in in the code fragment below.

```
//Binding of the diff decorator
function bindDiffTreeDecorator(parent: interfaces.Container): void {
   parent.bind(DiffTreeDecorator).toSelf().inSingletonScope();
   parent.bind(DiffLabelProvider).toSelf().inSingletonScope();
   parent.bind(DiffDecoratorService).toSelf().inSingletonScope();
   parent.rebind(TreeDecoratorService).toService(DiffDecoratorService);

}
```

Once the diff tree has been populated, the *DiffTreeDecorator* is called and delivers an object of type *MaybePromise<Map<string, WidgetDecoration.Data»*. This object is filled by iterating through the three node arrays in the *DiffViewWidget*, namely the additions, changes and deletions. Each entry from those arrays will get a corresponding mapping that contains the Id of the *DiffTreeNode* coupled with the desired widget decoration data, which allows for various customizations such as fonts, background colors, captions, tooltips etc. In this specific case the fonts have been customized in order to grant them the desired color with the code for the addition decorations being illustrated below.

```
//Excerpt showcasing the generation of decorations for additions
decorations(tree: Tree): MaybePromise<Map<string,
    WidgetDecoration.Data>> {
     ...
    const additions: DiffTreeNode = (tree.root as
       DiffTreeNode).children[0] as DiffTreeNode;
    for (const child of additions.children) {
       result.set(child.id, { fontData: { color: "darkgreen" } } );
    }

    return result;
}
```

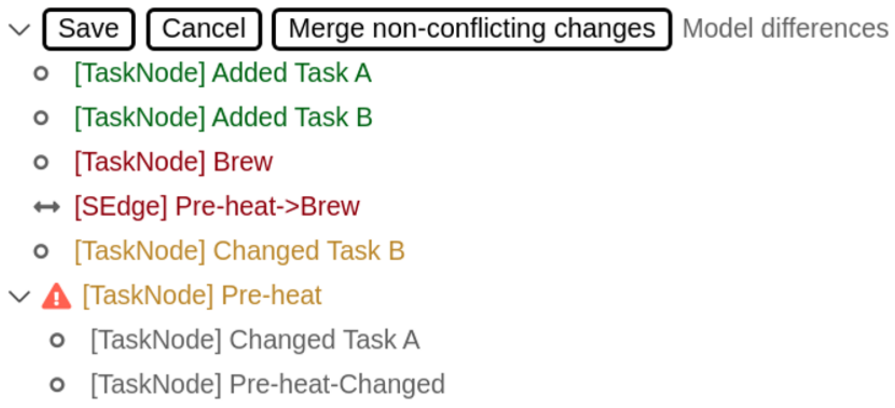
This *DiffTreeNode* type extends the already existing *SelectableTreeNode*, *CompositeTreeNode*, and *ExpandableTreeNode* types to include information pertinent to the displaying of differences and the interaction with the *DiffMergeDiagWidgets*, with the most important additions being listed below.

```
//Structure of a DiffTreeNode
export interface DiffTreeNode extends SelectableTreeNode,
    CompositeTreeNode, ExpandableTreeNode {
    id: string;
    changeType?: string | undefined;
    elementType?: string | undefined;
    source?: string | undefined;
    target?: string | undefined;
    icon?: string | undefined;
    modelElementId: string;
    diffSource?: string | undefined;
    ...
}
```

These additions made in the *DiffTreeNode* type fulfil a number of roles, such as allowing for edges to be displayed and correctly identified by their source and target, as well as centering on certain diagram elements based on their model element Id. The centering of the *DiffMergeDiagWidgets* on an element occurs by means of a *CenterAction*, a Sprotty feature centering the viewport of a diagram widget either around a specific element, a series of elements or simply the center of the canvas. The identification of the model elements occurs based on their element Ids. The Id of a model element, coupled with the *diffSource* attribute of a *DiffTreeNode* allows for accurate viewport centering both in case of changes to the diagram elements, where the same model element is present in both / all three diagrams, as well as in case of additions or deletions, where one or more versions the diagram might not contain a model element with a certain Id. Should the user click on a change in the diff tree, a *CenterAction* will be dispatched to all the *DiffMergeDiagWidgets* telling them to center around the model element with the specified Id.

Should the user however click on a deletion or addition in the diff tree, then the *diffSource* attribute is checked to identify the origin of the addition or deletion, and thus ascertain which widget(s) contain the model element in question. In this case, a *CenterAction* will be dispatched to the widget where the change originated. When dispatching a *CenterAction* we differentiate between two types of elements, namely nodes and edges. This distinction is made based on the *elementType* attribute of the *DiffTreeNode*. In case of a node, the *modelElementId* is used for the centering, while centering on edges in the diagram requires a bit more information in order to ensure that the centered viewport is actually of any use to the user.

Merely centering the viewport based on the Id of an edge, can often leave the endpoints of said edge outside the visible part of the canvas, especially if the source and target of the edge are positioned far apart from each other. In order to mitigate this problem the *CenterAction* is being performed on the ends of the edge, indicated by the source and target attributes of the *DiffTreeNode*. This ensures that the two diagram nodes connected by the edge are visible and users are able to understand the semantic implications of the change made to the edge. The *elementType* attribute is also responsible for indicating what icon a certain *DiffTreeNode* will get. To this end we distinguish between 5 types of nodes, starting with the root of the tree which will have no icon, but instead will have three buttons appended to it, namely a **Save** and a **Cancel** button, pertaining to the entire merge process, similar to other products out there, as well as a button intuitively labeled **Merge non-conflicting changes**. The second type of node is the *GLSPGraph* which has been anointed with a project-diagram icon, followed by task nodes which are preceded by a solid circle icon, as well as edges which have an arrow depicted next to them. The fifth type of node, representing a conflict, is marked by a warning symbol (an exclamation mark enclosed within a triangle) of a bright orange color. This serves to further distinguish it from the other changes and communicate to the users that this node requires special attention. In addition to the icon, it also contains two child-nodes representing the two conflicting changes. Figure 3.10 depicts the diff-tree menu items along with some of the different types of nodes.



[TaskNode] Added Task A
[TaskNode] Added Task B
[TaskNode] Brew
[SEdge] Pre-heat->Brew
[TaskNode] Changed Task B
[TaskNode] Pre-heat
[TaskNode] Changed Task A
[TaskNode] Pre-heat-Changed

Figure 3.10: diff-tree representation

Once that widget is centered around the correct element, its viewport is queried via a *GetViewportAction*, which is also a Sprotty feature. Once the correct viewport for the element is obtained, it is passed on to the other *DiffMergeDiagWidget(s)* by dispatching a *SetViewportAction* containing the previously queried viewport. This process allows us to center *DiffMergeDiagWidgets* around what is essentially a blank space within the diagram, and thus offer a better overview of the diffs in a compare-and-contrast manner.

Aside from diffing-related functionality, the diff tree is also responsible for handling some merging tasks, such as allowing the users to apply/ignore certain changes, as well as save or discard the current result. Starting from the top, both figuratively and literally, the Save and Cancel buttons in the diff tree are the first and perhaps most broad functionalities added in this regard. The two buttons refer to the merging process and they allow a user to either persist the current state of the diagram, or discard the entire merge process altogether, thus reverting any potential changes that might have been merged so far. Both buttons merely perform a request to the DiffMerge backend, which is ultimately responsible for handling the logic behind the chosen action.

In order to ensure that the merge process is entirely reversible up until the point when the **Save** button is clicked, a temporary file is created once the first change is applied. This file is a copy of the original file and it will remain untouched for the entire duration of the merge process. As any change is applied in real time to the original file, this copy is required in order to easily allow the users to revert back to the original state of affairs, should they wish to do so. Once the Save button has been clicked, a new file containing the merge result is created (marked by the ending "_MERGED"), and the temporary file replaces the original file, thus ensuring the initial file remains untouched.

The **Cancel** button on the other hand restores the original state of the files by replacing the edited ones with the temporary file copies that have been created when the comparison was first triggered. But simply replacing the files does not suffice in order to restore the original comparison result. As previously stated, because any merged changes get directly applied and persisted to the original files, a new comparison of the now reverted files needs to be triggered, in order to allow users to begin the merging process anew. The replacement of the files is being performed by the backend, with the temporary files being purged as soon as the originals have been replaced by them. Immediately after, a new comparison is triggered and the result is delivered back to the DiffMerge Theia extension. Once a new comparison result has been received by the frontend, *DiffMergeDiagWidgets* as well as the *DiffTreeWidget* are refreshed to display the new information and a message is displayed by the message service, letting the user know that the changes have been reverted.

The functionality of the **Merge non-conflicting changes** button is true to its name. When clicked it tells the *ComparisonService* to ask the backend to merge an array of diffs. This array consists of all the diffs in the tree, except the conflicts. Once those diffs have been merged a new comparison is delivered to the frontend and once again the *DiffMergeDiagWidgets* as well as the *DiffTreeWidget* are refreshed to display the new information. In this case the diff tree will only contain entries representing merge conflicts while the diagram widgets will show the result of the partial merge as well as the conflicting changes from which the user has to pick. Once again the message service will display a message informing the user that the non-conflicting changes have been applied.

In addition to the three diagram-spanning buttons, users are able to apply or discard individual changes via the context menu opened by right clicking them in the diff tree. In order to achieve this, a new menu contribution has been implemented, similar to the one used to select the files and trigger the comparison in the first place. This *MergeDiffMenuContribution* presents users with two options upon invoking the context menu of a diff from the diff tree: Apply and Discard. Once a diff has been chosen to be merged, a check is performed to see if a two-way or a three-way comparison has been made. In a two-way comparison, one file has been selected as the base for said comparison, and another one is regarded as containing all the changes. In this case a call is made to the *ComparisonService*, similar to the one responsible for generating the comparison in the first place. A fetch request goes out to the DiffMerge backend asking it to perform a merge of the selected diff. The diff is uniquely identified based on its source, which file it originates from, as well as the id of the respective model element. Once the backend has merged the desired diff and returned a new comparison to the frontend, the diagram widgets, as well as the diff-tree widget get refreshed to reflect the new state of things.

Should the user have chosen to execute a three-way comparison, the merging process for individual diffs becomes slightly more complicated with some additional checks being required. When choosing to apply a diff, in a three-way comparison/merging scenario, one has to once again identify the file that diff originates from. As opposed to the previous scenario where diffs would always originate from the other file, besides the base one, in a three-way merge a diff has two possible origins, namely each one of the two files that are being compared to the base one. So in order to identify the origin of the diff we have to look at the *diffSource* attribute of the selected *DiffTreeNode* representing the diff. Once the source has been ascertained, the merging process is identical to the one for the two-way comparison/merge, namely the *ComparisonService* is invoked and the merging of the diff is delegated to the DiffMerge backend. Upon receiving the result, the comparison and diff tree are once again refreshed to reflect the updated state of the diagram.

It is perhaps worth mentioning at this point that subsequent comparisons will be invoked using the original base file even though users are being presented with the changed one. This is done in order to avoid the introduction of fictitious changes. For example let's say an addition on the left side has been merged to the base file. Should a new comparison take place using this changed base file, it would yield a new deletion on the right side. This is because the original addition on the left side is now also present in the base file, thus effectively turning it into a deletion on the right side, as that is the only side where the added element is missing. By doing this we allow users to incrementally generate the desired merge result in a way that is visual, as applied diffs immediately get added and displayed to the base diagram version, while also avoiding artifacts caused by subsequent comparison computations using an ever-evolving base file.

The process for discarding a certain diff is more or less a one to one replica of the merging, with the only difference being the revert boolean flag that gets set to *true* when invoking the *getSingleMergeResult()* method of the *ComparisonService*.

In order to create the UI layout containing the two to three *DiffMergeDiagWidgets* as well as the *DiffTreeWidget*, the existing *SplitPanel* implementation from Theia core has been extended. This new element represents a convenience wrapper which allows the widgets within it to be arranged into resizable sections and its main structure is visible in the code snippet below.

```
//Structure of a DiffSplitPanel
export class DiffSplitPanel extends SplitPanel implements
    StatefulWidget, Navigatable {

   public widgetId = 'diffSplitPanel';
   public uri: URI;
   public leftWidget: DiffMergeDiagWidget;
   public baseWidget: DiffMergeDiagWidget;
   public rightWidget: DiffMergeDiagWidget;

   public setSplitPanel(splitPanel: DiffSplitPanel) {
      this.addWidget(splitPanel);
   }
}
```

For the specific task of creating the UI for the comparison and merging process, the custom *DiffSplitPanel* includes three *DiffMergeDiagWidgets*, used to display the three diagrams side by side in a three-way comparison. The method responsible for the initialization of the aforementioned *DiffSplitPanel* is showcased by the code fragment below.

```
//Excerpt from the DiffPanel initialization for a three-way
   comparison
public initThreewayDiffPanel(leftWidget: DiffMergeDiagWidget,
   baseWidget: DiffMergeDiagWidget, rightWidget:
   DiffMergeDiagWidget, uri: URI) {
      this.leftWidget = leftWidget;
      this.baseWidget = baseWidget;
      this.rightWidget = rightWidget;
      this.addWidget(leftWidget);
      this.addWidget(baseWidget);
      this.addWidget(rightWidget);
      leftWidget.actionHandlerRegistry.register(SetViewportAction.KIND,
         new ViewPortChangeHandler(baseWidget, rightWidget));
      baseWidget.actionHandlerRegistry.register(SetViewportAction.KIND,
         new ViewPortChangeHandler(rightWidget, leftWidget));
      rightWidget.actionHandlerRegistry.register(SetViewportAction.KIND,
         new ViewPortChangeHandler(leftWidget, baseWidget));

}
```

In addition to the three widgets used to display the diagrams, custom handlers have been implemented and registered in order to handle *ViewPort* changes triggered by the user. Such changes occur when the user pans around in a diagram or adjusts the zoom level of the diagram. These *ViewPort* changes cause a *SetViewPortAction* to be dispatched from the originating widget's action dispatcher.

```
//ViewPortChangeHandler and ForwardedAction
export class ViewPortChangeHandler implements IActionHandler {
  ...
   handle(action: SetViewportAction): ICommand | Action | void {
      if (!(action instanceof ForwardedAction)) {
         this.otherWidget.actionDispatcher.dispatch(new
            ForwardedAction(action));
         if(this.threewayWidget) {
           this.threewayWidget.actionDispatcher.dispatch(new
              ForwardedAction(action));
         }
      }
   }
}
export class ForwardedAction extends SetViewportAction {
   readonly kind: string = "viewport";

   constructor(public readonly setViewportAction: SetViewportAction)
      {
        super(setViewportAction.elementId,
            setViewportAction.newViewport, setViewportAction.animate);
    }
}
```

In order to keep the diagram widget's *ViewPort* in sync when the user interacted with one of them, a new action, the *ForwardedAction* has been introduced, together with a new handler, the *ViewPortChangeHandler*. This handler gets registered to the diagram widgets during the initialization of the *DiffSplitPanel*. Through this new handler, all actions that get dispatched as a result of *ViewPort* changes get intercepted. This is where the newly introduced *ForwardedAction* comes in. Each time a *SetViewPortAction* gets handled, a new *ForwardedAction*, containing the *ViewPort* changed by the user, gets dispatched to the other diagram widget(s). A new handler is not necessary as the new action merely extends the one that already exists. In order to avoid an endless loop of forwarded actions, caused by a change triggering another change in return, a check is performed. A new *ForwardedAction* is being dispatched only if the handled action is not already a forwarded one, as evidenced by the code above.

Circling back to the original *SplitPanel* that has been extended to suit our needs, one has to mention one minor limitation. An orientation is required, be it either horizontal or vertical. So since a *SplitPanel* can only perform one type of split at a time, two nested ones have been used in order to achieve the desired combination of horizontal and vertical splits. A horizontal *DiffSplitPanel* contains the three *DiffMergeDiagWidgets* used for the side by side diagram comparison. This panel is in turn nested inside of a vertically split one, which contains the diff tree in the top section and the horizontal split panel in the bottom section.

3.2.2 Backend Diff-Merge Component

As previously mentioned, the DiffMerge Java backend is responsible for most of the computations regarding the comparison, as well as the merging process for diagrams. The functionality of this component is made available through a Jetty[jet] servlet container that exposes a RESTful API through the Jersey [jer21] library. These REST endpoints allow the frontend to request a series of operations:

- **Diagram comparison**: Returns the result of either a two way or three way comparison between diagrams
- **Diagram merging**: Merges two diagrams and persists the resulting merged version
- **Single change merging**: Merges a single change from one diagram version to another
- **Save changes**: Persists the current version of the diagrams, usually invoked after one or more individual changes have been merged

- **Revert changes**: Discards the current versions of diagrams that may have some changes already merged to them, and reverts to the original state before the beginning of the comparison

The comparison and merging functionality itself is being offered by the EMF Compare framework which has been added as a dependency to the DiffMerge backend component, but before Workflow Modeling Language Diagrams can be fed into and processed by EMF Compare some configuration is required. Before we dive into the specific details of the WFML use case, an overview of the comparison process from EMF Compare's perspective is in order.

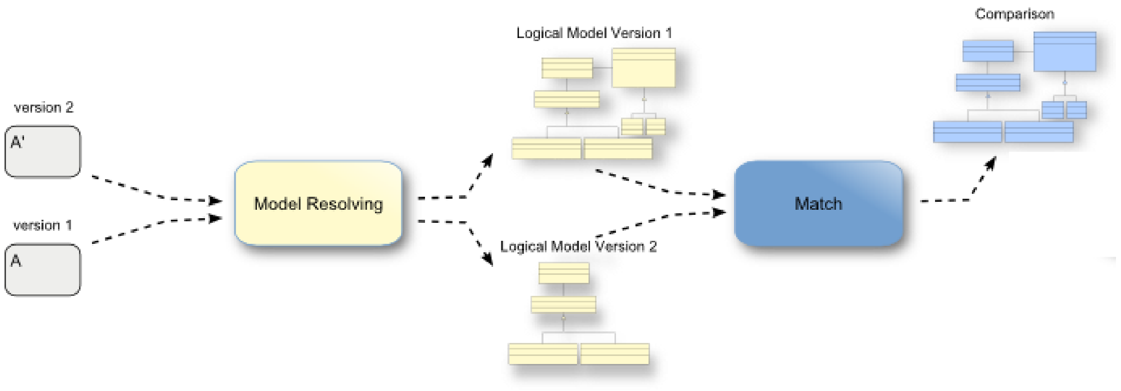


Figure 3.11: EMF Compare comparison process [emf21]

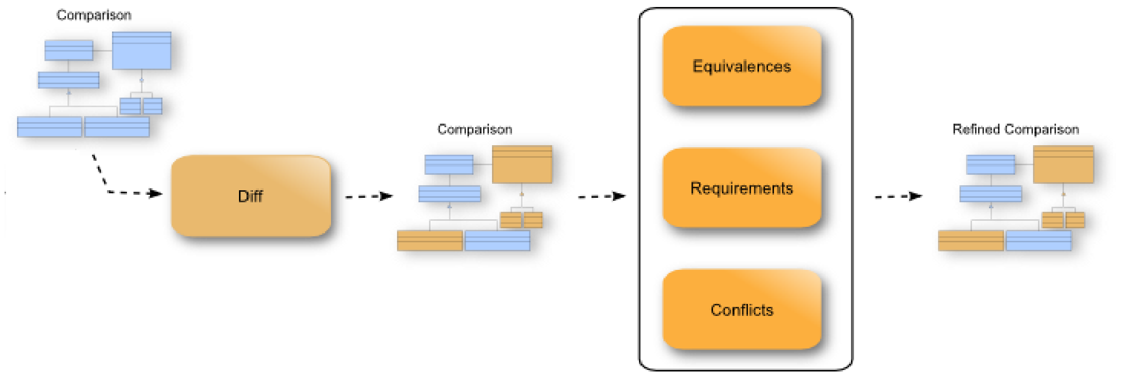


Figure 3.12: EMF Compare differencing process [emf21]

As evidenced by figures 3.11 and 3.12 above, the comparison process conducted by EMF Compare can be split into six main steps, represented by the rounded rectangles. It is worth noting that while only a two-way comparison is depicted, the process is similar, consisting of the same steps for a three-way comparison. During the **model resolving** stage the framework parses the models, in our case the files selected by the user, and identifies all the required parts so that a comparison of the whole logical model can be conducted. Once the models have been resolved, a **matching** is performed on them. This means that the model elements are being iterated over and mapped together, either in pairs of two or groups of three, depending on the type of comparison conducted. This determines correspondences of elements across the two or three files, for example determining that an element A from one file corresponds to element A in the second, and / or potentially third file. Now that elements have been matched, the **diffing** process can begin. During this phase the mappings will be analyzed and it will be determined whether the two or three elements of a match are equal or if any differences are present. Once the differences have been determined they have to be analyzed in order to ascertain three things. The first one would be **equivalences**, where two distinct differences might in reality generate the same change and will be linked together as such. The next step is to determine potential **requirements** that might be necessary for the merging of differences. This would be the case for nested elements for example, where an inner element would not be able to be merged without its parent. Once the potential requirements have been computed it is time to check for **conflicts**. In this phase EMF Compare browses through the found differences and checks for potential conflicts, with the caveat that conflicts will only actually be detected in the case of a three-way comparison, as a common ancestor is required in order to establish a base version for a certain element.

Now that the comparison process has been briefly introduced, we can move on to the specific configuration steps required in order to compare Workflow Modeling Language diagrams. As illustrated by Figure 3.3 in chapter 3.2.1 describing the Theia extension, the WFML is based upon and extends the GLSP Metamodel. As such we have to ensure that EMF Compare is able to perform the first step of the comparison process, namely resolve WFML models/diagrams in the first place.

```
//EMF Compare configuration
IEObjectMatcher matcher = DefaultMatchEngine
      .createDefaultEObjectMatcher(UseIdentifiers.WHEN_AVAILABLE);
  IComparisonFactory comparisonFactory = new
     DefaultComparisonFactory(new DefaultEqualityHelperFactory());
  IMatchEngine.Factory matchEngineFactory = new
     MatchEngineFactoryImpl(matcher, comparisonFactory);
  matchEngineFactory.setRanking(20);
  IMatchEngine.Factory.Registry matchEngineRegistry = new
     MatchEngineFactoryRegistryImpl();
  matchEngineRegistry.add(matchEngineFactory);
  EMFCompare comparator = EMFCompare.builder()
     .setMatchEngineFactoryRegistry(matchEngneRegistry).build();
```

This is achieved through the series of steps showcased in the code snippet above and described in the paragraphs below. Firstly, because WFML elements extend *GModel* elements, they have unique identifiers. We need to tell this fact to the matcher, so that

Ids will be relied upon when deciding which elements correspond to each other across the various diagram versions. Next up we need a comparison factory for the comparison itself. This requires no special configuration for our use case, with the *DefaultComparisonFactory* sufficing for our goals. With the matcher and comparison factory at hand we can now use the two to create a *MatchEngineFactory*, which is ultimately responsible for defining a general contract of a matching engine. This will then serve as the entry point of the comparison process, and has to be added to the match engine factory registry. In our specific case this registry will only contain one entry, namely the aforementioned match engine factory. Once we have created a *MatchEngineFactoryRegistry* containing our desired factory, we can use this to build an EMFCompare object configured with the previously given engines, which is effectively a comparator that will handle the comparison of WFML diagrams.

An EMFCompare Comparison is obtained as a result of the comparison between the two/three versions of a diagram.

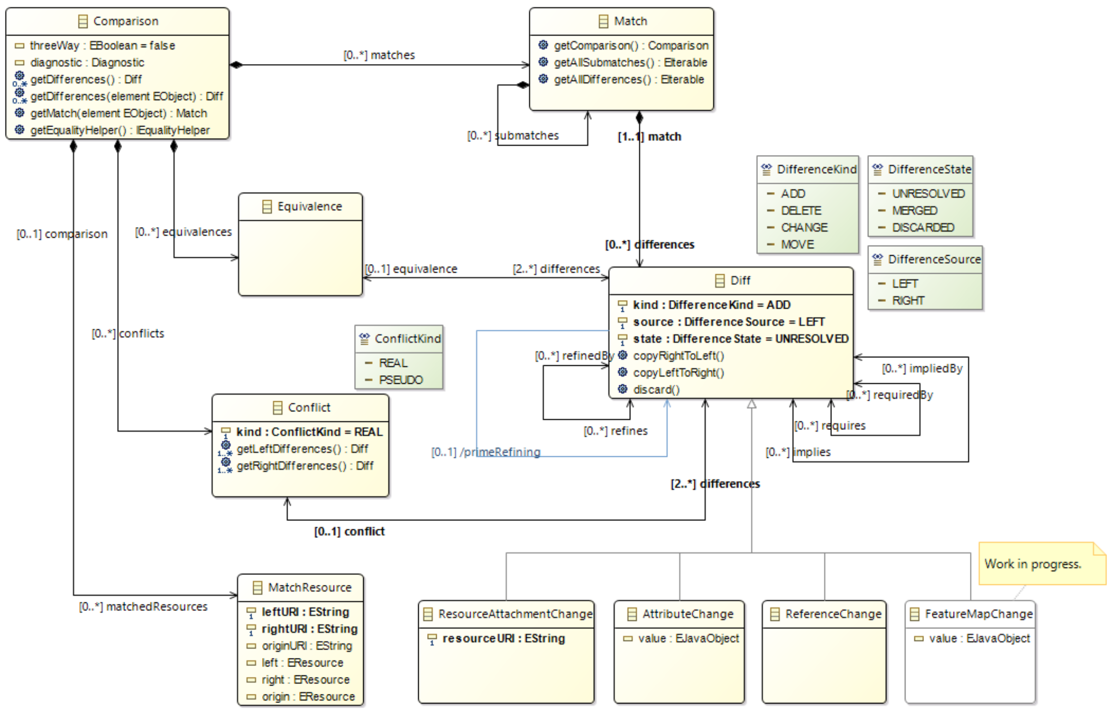


Figure 3.13: EMF Compare Metamodel [emf21]

The EMF Compare metamodel depicted in figure 3.13 describes the single model used by EMF Compare to represent all of the information pertaining to the comparison. The root of this model is a Comparison object, which is created at the beginning of the matching process and will be subject to a series of refinements during the remaining steps of the comparison process. A comparison will contain a boolean flag to distinguish between a two way and a three way comparison, as well as a series of Diffs which represent all the changes detected, a series of matches for the elements, with a series of sub-matches of their own (e.g. for nested elements) and the Diffs found in the match, as well as a series of conflicts, which would only be detected in case of a three-way merge.

Diffs themselves contain information about their type, such as *ADD*, *DELETE*, *CHANGE*, and *MOVE*, as well as their source, namely the left or the right version. This information is not only required in order to determine where a change originated from, so that it can later be displayed accordingly in the frontend, but also to correctly identify and depict conflicts.

Conflicts are only detected in case of three-way comparisons, as a common ancestor of the two versions is required in order to ascertain if a diff leads to a conflict or not. EMF Compare distinguishes between two kinds of conflicts, namely *PSEUDO* and *REAL*. The first kind, *PSEUDO* conflicts describe situations when both sides of the comparison have technically changed when compared to their common ancestor, but matter of factly the two sides are now equal. This means that the end result is the same on both the left and right side, thus requiring no action in order to solve them. *REAL* conflicts on the other hand, describe situations where the value on all three sides is different, meaning there have been changes made, to both the left and right version and they are now not equal. These conflicts require resolution before a merge can be performed.

Once the Comparison object is initialized all of the required information is present and the comparison process is finished. Before this information can be sent to the frontend it has to be processed and ultimately converted to a *ComparisonDto* Java object. The structure of this DTO [CK03] corresponds to the one present in the frontend containing an array of matches as well as a boolean flag specifying if the comparison in question is a three-way or a two-way one.

The goal of the aforementioned processing of the comparison information is twofold. On one hand we aim to remove superfluous information so that the amount of data sent between the backend and frontend components is restricted to the bare minimum required in order to display the differences. For example references between matches and diffs are required, whereas equivalences are not. On the other hand, during the processing the granularity of the diffs can be tweaked in order to achieve the desired result. In the concrete case of the WFML we are interested in changes pertaining to two types of elements, edges and tasks. While *TaskNodes* themselves are nested elements, containing labels and icons, the choice has been made to only represent the diffs on a *TaskNode* level. This means for example that the addition of a new node would bring with itself the addition of a new icon and label as well, but this would only clutter the UI without providing an advantage to the users or allowing them to gain new insight into the changes that occurred.

The aforementioned processing consists of a few steps that will be described in the following paragraphs. Firstly we need to know which elements in our diagram have been identified and matched by EMF Compare to begin with. The code fragment below represents the steps taken in order to process a comparison and convert it to a *ComparisonDto*.

```java
//Comparison processing
@Override
public ComparisonDto getComparison(String left, String right, String
    origin) throws InvalidParametersException, IOException {

   Comparison comparison = compare(left, right, origin);
   ComparisonDto comparisonDto = new ComparisonDto();
   List<Match> matchList = comparison.getMatches();
   List<MatchDto> matchDtoList = new ArrayList<MatchDto>();
   comparisonDto.setThreeWay(comparison.isThreeWay());
   for(Match match:matchList) {
      MatchDto matchDto = mapMatch(match, comparison.isThreeWay());
      if(matchDto != null) {
         if(matchDto.getSubMatches()!=null) {
            matchDtoList.add(matchDto);
         }
      }
   }
   comparisonDto.setMatches(matchDtoList);
   return comparisonDto;
}
```

The matches are fetched from the comparison object and then an attempt to map each one to a *MatchDTO* is made. This is achieved by iterating through the match list and invoking a mapping function on each match. The mapping function sets the left, right, and, in case of a three way comparison, the origin attributes in the DTO. Next we need to add the corresponding differences to our match DTO. These diffs are fetched from the EMF Compare match object and are being parsed before being added to the diff list of the match DTO. During this parsing phase we can check the diff kind and decide if/which types should be excluded from the visualization. As depicted in fig 3.13, EMF Compare distinguishes between 4 kinds of diff, *ADD*, *DELETE*, *CHANGE*, and *MOVE*. As the position of an element has no semantic meaning within the WFML, we have decided to ignore these kinds of diffs, as they would only clutter the visualization without adding any meaning to it.

At this point we have mapped the match together with its corresponding diffs. Nested diagram elements, such as task nodes containing labels and icons, do not need special handling in case of additions and deletions, as we are only interested in differences on a task node level, meaning it suffices to convey to the user that a task node has been added or deleted, without explicitly telling them that the corresponding labels and icons have also been either added or deleted to/from the diagram. When it comes to diffs of type *CHANGE* we also need to take a look at the submatches for any given match, as the change might have occurred within one of the nested elements, such as a label. Therefore we iterate through the submatches for any given match, check to see if they contain any diffs, and if they do we take those diffs and append them to the DTO for the match itself. Again, this is done in order to preserve the desired granularity when displaying the diffs, as telling the user that a task node contains changes within its label would bring no added value but instead only add more entries to the diff tree in the frontend, thus making the UI seem cluttered without conveying any new relevant information. The aforementioned statement rings true only for the particular case of the WFML, where changes can basically only occur within the label of a task node and therefore do not need any further localization within the node itself. Other types of diagrams might contain a series of relevant attributes nested within an element such as a task node, in which case the granularity of diff representation can be adapted to suit the individual needs of the use case. The basic process to achieve this is to map the diffs, as they are, to a DTO up to the desired level, and then simply append all the diffs within the nested elements to the DTO representing the finest granularity level desired.

Once the matches and their diffs have been successfully mapped to the corresponding DTOs, the list of matches is appended to the comparison DTO and is sent to the frontend to be displayed. This process is done each time a comparison is triggered, be it a comparison for the sake of comparing two diagram versions, or a comparison triggered as part of the merging process.

Aside from providing the frontend with a comparison when asked to, the Diff-Merge backend also handles merging and conflict resolution. When asked by the frontend to merge two files, representing different versions of a diagram, the Diff-Merge backend passes the task on to EMF Compare. A comparison is conducted between the two / three files and then the found differences are applied to one of the files under comparison. In our implementation the EMF Compare method *copyLeftToRight()* is called which copies the diffs to the right file. The possibility exists to also copy diffs the other way around, from right to left, but for this use case, namely merging two versions that exhibit no conflicts, the distinction is irrelevant, as the end result would be identical and the direction in which the diffs are copied would have no bearing whatsoever on the outcome of the merging process.

Should the two versions contain conflicting changes however, then the user will have to take action in order to complete the merge. The Diff-Merge backend will handle merging all of the non conflicting changes. The comparison object will then only contain the conflicting changes, out of which the user can decide which one they would like to apply / discard in order to complete the merge.

The third and final task for which the backend is responsible consists of merging individual changes. In order to allow the users to build up the final result step by step by adding or discarding changes to the last common diagram version, a number of steps are required. In order to ensure the consistency of the files in case of an aborted merge process and to avoid having to keep track of the already merged diffs, a copy of the base file is created at the beginning of the merging process. Then depending on the origin of the diff to be merged, a two-way comparison will take place, either between the left version of the diagram and its base, or the right one and its base. What this achieves is an incremental generation of the final merge result. The difference between this, and copying the diffs to either the left or right version, by calling either *copyLeftToRight()* or *copyRightToLeft()*, consists in the fact that copying individual changes to the base file already starts from a blank slate. Initially the base version is untouched by any changes conducted in either diagram version, left or right. In order to achieve the same result one would have to decide which version serves as a base, either left or right, and subsequently discard all the changes that have been made to that version, only to allow for the possibility of adding them later, one by one.

A new comparison of the diagrams is automatically computed before applying a new change or conducting an automatic merge for a few reasons. Firstly, the architecture of the components makes it so that the Diff-Merge backend and the frontend only communicate for short periods at a time, when a REST call is made. This means that the EMF Compare comparison object stops existing shortly after it has been converted to a DTO and sent off to the frontend, thus requiring a new comparison computation each time a diff is to be applied. Furthermore it is entirely possible that the diagram files under comparison / merge have been modified outside of the Diff-Merge backend. A new comparison ensures that the diffs are computed based on the current version of the files.

3.2.3 Communication protocol

The communication protocol between the Theia Diff-Merge extension and the diff-merge backend has already been touched upon a number of times while describing other parts of this thesis. This chapter serves to aggregate all of the information that has been sprinkled throughout the chapters describing the frontend and backend components and expand upon it. The application-layer protocol designed for the communication between the frontend and the backend defines how the two components pass messages to each other, particularly by defining how the request and response messages should look and what they should contain. While an efficient protocol has been one of the goals of this thesis from the get-go, the design of it was an iterative process as opposed to designing it once and subsequently implementing it. This is because of the feature by feature way in which the implementation took place. As new features were added to the extension the requirements of the protocol evolved right alongside them.

Since the client, in this case the Theia Diff-Merge extension, is only responsible for rendering the diagrams and the diffs without making any computation in this regard, the communication protocol is ultimately tasked with conveying all the information necessary to compute the comparison and merge results on the backend side, as well as all the information required in order to display the diffs to the user in the frontend extension.

Perhaps one of the most important characteristics when it comes to the efficiency of a communication protocol is its speed. And as is the case with all communications over a network, the speed available over the network is limited, either by the ISP or the hardware running locally such as routers and switches. With these variables being out of the control of a developer, one other way to increase the speed, with which messages between two applications reach their destination, is to keep the total amount of data that needs to be sent to a minimum.

We will begin by exploring the requests made from the frontend to the backend such as the ones made when a new comparison is required, when changes are to be merged or when a merge process is concluded and files need to be persisted. In our case the minimal information that has to be sent over to the backend when performing a request varies slightly depending on the operation. When triggering a comparison or an automatic merge only the paths of the diagram files are required. This concretely means that we will only need to send 2 or 3 string parameters with each request, depending whether we are conducting a two- or three-way comparison/merge.

Persisting files when concluding a merge as well as reverting all applied changes similarly require only the file paths for the diagrams to be passed along, as this allows the backend to conduct all of the necessary computations. This fact constitutes an additional advantage as it eliminates the need to store any sort of change lists on the frontend side that could potentially have a negative impact on performance. Applying individual changes on the other hand involves 2 additional parameters aside from the file paths. These parameters are the ID of the diagram element and a boolean flag which lets the backend know whether a specific change is to be applied or discarded. Merging multiple changes, such as when the user decides to apply all non-conflicting changes, also requires another parameter next to the diagram file paths. In this case, the additional parameter is a list of diagram element IDs upon which changes are to be applied.

The operations themselves, such as triggering a comparison or merging a diff, all have their own API endpoints, thus not requiring any information regarding the operation type to be included in the request and passed along to the backend.

When looking at the other direction of communication things become a little more complicated. This is mainly due to the fact that the backend does all of the computations while the frontend is tasked with displaying the results. In contrast to the requests coming to the backend, their responses are more expansive. Each request requiring a computation to be done by the backend will receive a *ComparisonDTO* as a response. Some aspects regarding the *ComparisonDTO* have been briefly touched upon in chapters 3.2.1 and 3.2.2 and we will now go into more detail concerning its structure and attributes. The structure is based on the Comparison object returned by EMF compare whose metamodel is depicted in figure 3.13. A *ComparisonDTO* is essentially a top level wrapper containing a list of matches consisting of *MatchDTOs*, and a boolean flag describing whether or not the returned result refers to a three-way comparison, as also evidenced by figure 3.9 showcasing the structure of a *ComparisonDTO* and its nested elements. A MatchDTO on the other hand is slightly more complex. As somewhat apparent from its name, this type of DTO is used in order to represent an element that has been matched across diagram versions by EMF Compare. In order to handle nested elements within diagrams, it contains a list of submatches of the same type as itself. It also contains three more attributes, left, right and origin. These attributes are used to describe an element across the three diagram versions that might be compared. Any of these three could be null, as elements can be either added to a diagram, thus being present only in the version where they have been added, or they can be deleted from the diagram thus only remaining present in the origin and/or other version of the diagram. In order to facilitate the displaying and handling of conflicts a new type has been introduced, the *DiagElemDTO*. This is nothing more than a wrapper class containing the ID and type of a diagram element. This information is relevant when multiple changes have been made to the same element across different diagram versions, as it allows the frontend to display each version of the change. The final and perhaps most important attribute contained by the *MatchDTO* is a list of diffs, in the form of a *DiffDTO* array. It is perhaps worth noting at this point that a match will be added to the matches list of the *ComparisonDTO* only if it or one of its submatches contains a diff. Otherwise it will be ignored as only information regarding diffs is relevant. The *DiffDTO* object contains information relevant to the diff itself. Its attributes describe the kind of diff (e.g. addition), the source of the diff (left or right), and the ID of the diagram element where the diff was found. In addition to the aforementioned attributes a *DiffDTO* also contains two more attributes, one for the type of change (e.g. *Referencechange*/*Attributchange*) and another one describing the changed attribute in case of an *Attributechange*.

CHAPTER 4

Comparison example

At this point the inner workings of the graphical diff merge tool have been explained in detail but some aspects might still seem somewhat abstract or hard to fully comprehend. In this regard we will briefly reiterate through all the steps that take place from the beginning of a comparison up until the conflicts have been solved and the merging process has been concluded, based on a concrete example. In order to aid in illustrating said process a number of figures will be employed, both of the implementation itself as well as simple WFML diagrams.

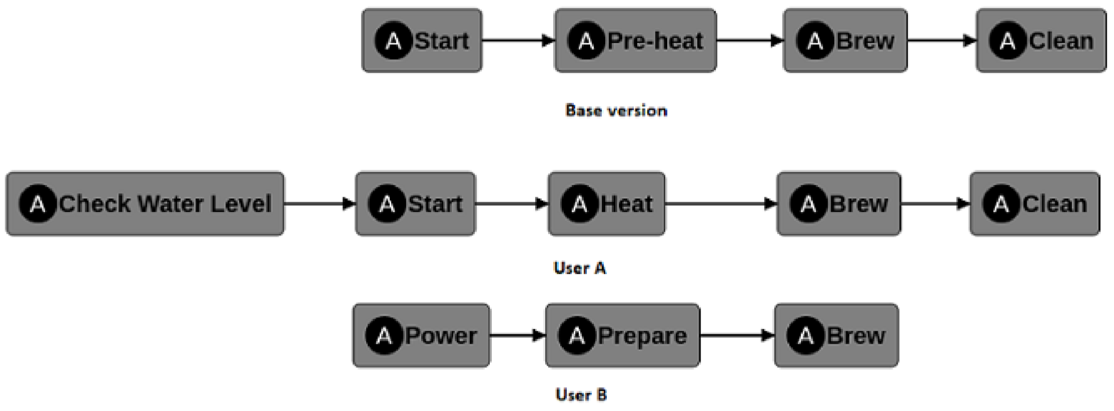
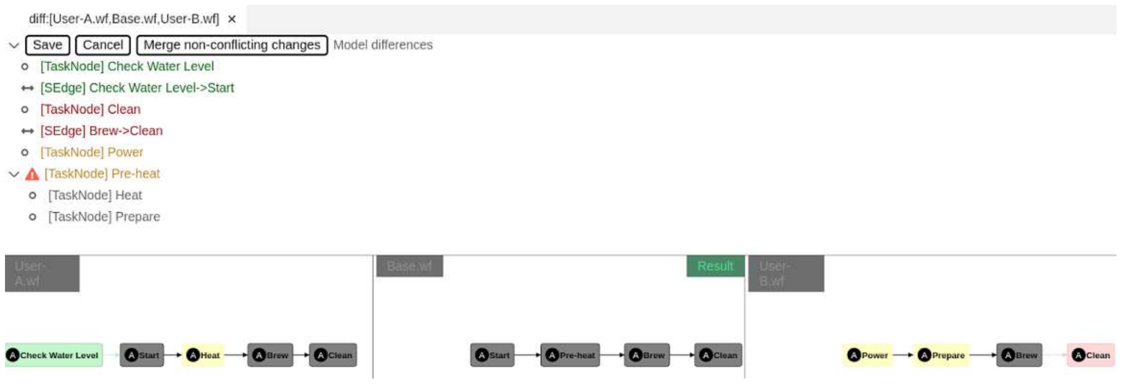


Figure 4.1: Example of files

Let us assume we find ourselves in the scenario depicted in figure 4.1. We have a diagram depicting the workflow of a coffee machine, with its four tasks: *Start*, *Pre-heat*, *Brew* and *Clean*. Next we have two variations of the base version, annotated with User A and User B. These depict two diverging states of the base diagram that emerged as a result of two users checking out the base version from a version control system such as Git and subsequently conducting a number of changes to it.

On one hand we have User A, who added a new task at the beginning of the coffee making process, *Check Water Level*, together with an edge connecting it to the Start task, as well as renamed a task, from *Pre-heat* to simply *Heat*. On the other hand User B renamed two tasks, namely *Start* became *Power* and *Pre-heat* turned to *Prepare*. Aside from those two changes User B also removed the *Clean* task and the edge leading to it.

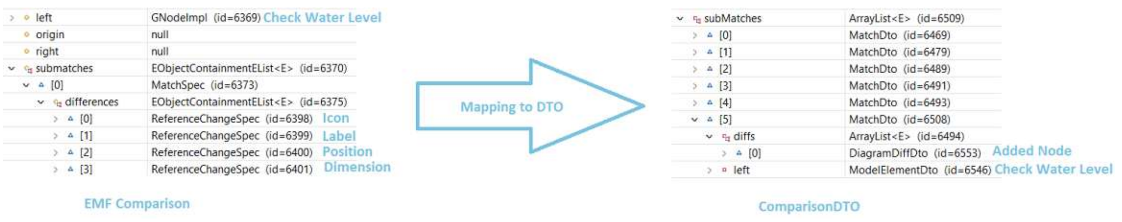
Now let us assume that the users' work is done and they are ready to commit and push their changes back to the version control system. The first of the two users to attempt this would be successful in their endeavor without encountering any issues, as the version of the diagram they are trying to push is a direct successor of the version already present in the version control system. The second user however would be faced with a merge conflict when attempting to push their changes, as the upstream file in the version control system has been changed in the meanwhile and a particular change is conflicting. This would be the task originally entitled *Pre-heat* which has been modified by both User A and User B.



- [TaskNode] Check Water Level
- [SEdge] Check Water Level->Start
- [TaskNode] Clean
- [SEdge] Brew->Clean
- [TaskNode] Power
- [TaskNode] Pre-heat
  - [TaskNode] Heat
  - [TaskNode] Prepare

Figure 4.2: Diagram Comparison

It is at this point that a comparison would be triggered and the user would be greeted by the comparison interface depicted in figure 4.2. While this process could be perceived as almost instantaneous by the user, quite a few things happen in the background in order to make this possible. Firstly a call is made to the Diff-Merge backend asking it to perform a comparison between the three files. In order to conduct said comparison the backend needs the paths to the diagram files, in our case suggestively named *User-A.wf*, *Base.wf* and *User-B.wf*.



```
left            GNodeImpl  (id=6369)
origin          null
right           null
submatches      EObjectContainmentEList<E>  (id=6370)
  [0]           MatchSpec  (id=6373)
    differences EObjectContainmentEList<E>  (id=6375)
      [0]       ReferenceChangeSpec  (id=6398)
      [1]       ReferenceChangeSpec  (id=6399)
      [2]       ReferenceChangeSpec  (id=6400)
      [3]       ReferenceChangeSpec  (id=6401)
```

```
subMatches      ArrayList<E>  (id=6509)
  [0]           MatchDto  (id=6469)
  [1]           MatchDto  (id=6479)
  [2]           MatchDto  (id=6489)
  [3]           MatchDto  (id=6491)
  [4]           MatchDto  (id=6493)
  [5]           MatchDto  (id=6508)
    diffs       ArrayList<E>  (id=6494)
      [0]       DiagramDiffDto  (id=6553)
    left        ModelElementDto  (id=6546)
```

Figure 4.3: Mapping to ComparisonDTO

Armed with this information EMF Compare performs the necessary computations and returns a Comparison object. As mentioned before, and evidenced by figure 4.3, the structure of the generated comparison will strongly depend on the structure of the diagram elements. In this case the added task Check Water Level contains four distinct differences, more specifically additions, due to its nested structure. A WFML task contains an icon, a label, a position and a dimension. However displaying these 4 differences would grant the user no additional understanding over the changes that occurred and instead may only clutter the UI and render the comparison/merging process more cumbersome and unintuitive. In order to mitigate this issue we decided to only distinguish between differences on a task node / edge level. As a result of this, the *ComparisonDTO* only contains one diff for this particular task node, namely the addition of the Check Water Level task. The same mapping mechanism is applied for all change types, resulting in a *ComparisonDTO* that contains no superfluous information and can be sent to the frontend where it can be processed further and displayed. This contrast can be clearly seen in figure 4.3 where the EMF *Comparison* contains four separate differences associated with the addition of a single task node, whereas after the mapping, the *ComparisonDTO* only contains one diff associated with the newly added task node.

At this point the frontend has obtained the information required in order to display the comparison. For this to happen the diff-tree is initialized and populated with all the diffs, the *DiffMergeDiagWidgets* are initialized with the appropriate files, *User-A.wf*, *User-B.wf* and *Base.wf*, and the *Splitpanels* have the diff-tree as well as the diagram widgets added to them, before being opened and displayed to the user. Now all that remains is to decorate the diff-tree and mark the corresponding diagram elements accordingly. Once this is done the user is presented with the view depicted in figure 4.2. The diff-tree lists the addition of a new task node *Check Water Level* and its corresponding edge by User A, the deletion of the *Clean* task together with the edge leading to it by User B, as well as the renaming of the *Start* task performed by User B. The conflict caused by the concurrent changes made to the task node originally known as *Pre-heat* is marked in the diff-tree by the presence of a bright orange warning sign preceding its original name. The two changes conducted by users A and B are represented by the children of this node, *Heat* and *Prepare*.

Using this view the user can now navigate between the diffs by clicking on them in the diff-tree, to which the three *DiffMergeDiagWidgets* will respond by centering either on the diff in question or on the blank space where the diff should be. For example if one were to click on the newly added task node *Check Water Level* the left widget would center on the node itself while the middle and right widgets would center on the space before the first node in the chain of tasks. Additionally panning the camera or adjusting the zoom level within one of the widgets will propagate the same action to the other two resulting in all three widgets being in sync with each other.

Furthermore users can now apply individual changes by right clicking them in the diff-tree and selecting *Apply*. Discarding diffs works in a similar manner. Applied diffs will become visible in the middle widget, which initially shows the base version of the diagram, while simultaneously representing the version that would be persisted should the user click the *Save* button. This fact is also indicated by the green *Result* banner in the top right corner of the middle *DiffMergeDiagWidget*. Each application or discarding of a diff from the tree results in a call to the Diff-Merge backend where EMF Compare is relied upon to perform the actual manipulation of the files. When applying a diff, a new comparison is triggered between the file containing the desired change and the one containing the base diagram version. The information regarding the origin of the change and implicitly the file containing it is stored in each individual node of the diff-tree representing a diff. This information is passed on to the backend which uses it to trigger the appropriate merging process.

In contrast to the mapping performed when constructing the *ComparisonDto* to be sent to the frontend, we are now interested in all of the changes contained by the nested elements of a task node. In order to correctly alter and ultimately obtain a valid WFML diagram all of the elements contained by a task node have to be added. This means that when applying a diff we start from the task node itself and subsequently also apply the diffs contained in its submatches. For the concrete case of the *Check Water Level* task we would also apply the diffs concerning the associated *Icon*, *Label*, *Position*, and *Dimension*.

Diffs can also be applied in bulk by making use of the *Merge non-conflicting changes* button. This makes a request to the backend asking it to merge an array of changes consisting of all the non-conflicting diffs.

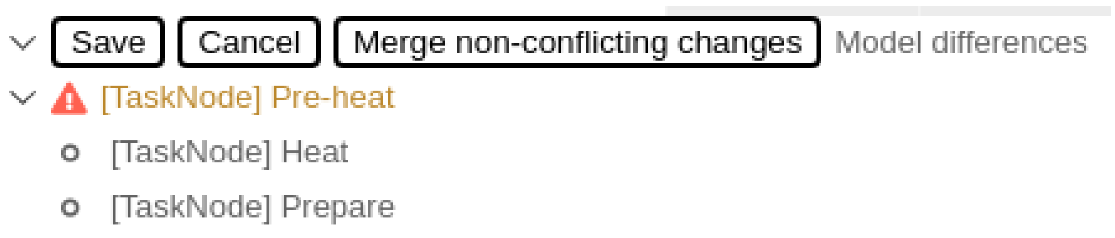


Figure 4.4: Diff tree after applying all of the non conflicting changes

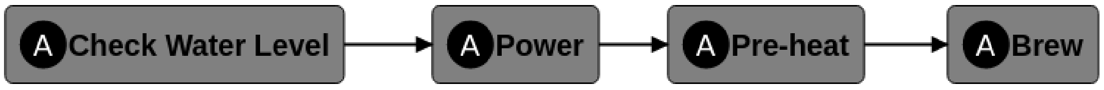


Figure 4.5: Merge result after applying all non conflicting-changes

In our concrete example applying all non-conflicting changes would yield the results depicted by figures 4.4 and 4.5. The diff-tree will now only contain the conflicting changes, namely the ones pertaining to the task node originally known as *Pre-heat*, with its two alternative renames to *Heat* and *Prepare* respectively. The middle diagram widget would contain the diagram shown in figure 4.5, where the *Check Water Level* task and its edge has been added (User A), the *Start* task has been renamed to *Power* (User B) and the *Clean* task, together with the edge leading up to it have been deleted (User B). As expected the conflicting changes have not been applied, thus leaving the *Pre-heat* task node untouched by this bulk merge.

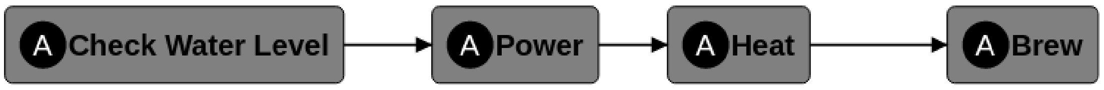


Figure 4.6: Completed merge result

Now all that is left is for the user to complete the merge process by resolving the conflict through the application of one of the two changes, either *Heat* or *Prepare*. Assuming the user decides to apply the *Heat* designation to the task and then click the *Save* button the current comparison view will close and the final merge result depicted in figure 4.6 would be opened in a new diagram widget.

If at any point during the merging process the user decides to abort, they can do so by clicking the *Cancel* button atop of the diff tree, which tells the backend to revert all the applied changes and loads the comparison anew. This is where the backup of the base made at the beginning of the merging process file comes into play, by overwriting the already modified file.

Once the *Save* button is clicked the current merge result is written to a new file and the base file gets reverted to its original state so that in the end all of the original three files remain untouched.

CHAPTER 5

Summary and Evaluation

This chapter will offer a summary of our work on this thesis and present the methodology used in evaluating the Diff-Merge tool, as well as the results obtained in said evaluation.

5.1 Summary

This thesis aimed to design and implement an effective and efficient notation to display diffs in a diagram. In addition to this we also set out to design and implement an efficient application-level protocol for the communication of diff data in a client-server architecture.

Because we set out to achieve the aforementioned tasks in the context of cloud-based IDEs we took a look at the top three cloud-based IDEs (according to the PYPL index) and ascertained their capabilities regarding diagram diffing and merging. This analysis, described in more detail in chapter 2, concluded that the top three cloud-based IDEs namely Cloud9, Replit and JSFiddle, not only lacked any capabilities of diagram diffing whatsoever, but neither did they allow for third party plugins or extensions to be added to them, thus essentially closing off any possibility users might have had of developing custom tailored features to suit their needs. In contrast to the aforementioned IDEs the Theia IDE supports both third party plugins and extensions, the distinction between the two being discussed in chapter 3.2.1.

Since diagram diffing capabilities were virtually non-existent in the context of cloud-based IDEs we took a look at conventional IDEs in order to help us decide on suitable diff representation means as well as conflict resolution processes. In this regard we have opted for a side-by-side comparison of the diagrams coupled with a simple color coding used to distinguish between diff types, similar to the ones commonly encountered when diffing text files. Green has been used to mark additions to the diagram, while yellow signifies a change, and red marks the deletion of a diagram element. Additionally a so called diff-tree has been implemented to aid with navigation between the individual diffs, as diagrams can often have an extensive layout causing them to not always be visible in their entirety, depending on the zoom level of the canvas they are rendered in. This diff-tree has a similar layout to common file explorers, listing the color coded diffs while also centering the diagrams on a specific diff upon it being clicked within the tree. This ensures that the desired diff is visible across all versions of the diagram.

When it comes to diagram merging and conflict resolution during the merge process we have decided to opt for the tried-and-tested options that have become commonplace across widely-used IDEs. Users can decide to either apply or discard individual changes and / or allow the IDE to automatically apply all non-conflicting changes. We have deemed these capabilities to be sufficient for the merging and resolution of conflicts within diagrams, as they allow users full control over the diffs that ultimately get to be included in the final merge result.

While the Diff-Merge extension is responsible for the visualization and user interaction, the diff computation and merging process is handled by the Diff-Merge backend. It is relying on the EMF Compare framework to compare the diagrams and subsequently perform any merging operations that might be requested of it. The results obtained from EMF Compare are then processed and serialized for transmission to the Diff-Merge Theia extension. Additionally the backend also exposes an API that is then consumed by the frontend allowing the triggering of merges of one or more differences, thus acting essentially similar to a middleware between the Diff-Merge Theia extension and the EMF Compare framework.

The communication between the frontend and backend takes place via REST requests with the request/response messages being defined by the developed application layer protocol. Requests made by the frontend only contain the information that is required in order for the backend to be able to perform the required operation. This means that any given request will at most contain the file paths for the diagrams, a boolean flag, and an array of diagram element IDs. Because the different operations have dedicated API endpoints, the aforementioned request parameters are sufficient in order to perform all the necessary tasks within the scope of this thesis.

Responses from the backend have retained a similar structure to the *Comparison* object returned by EMF compare while having the bulk of their attributes removed as they are no longer relevant outside of EMF compare, with the attributes that are relevant for diff visualization being included in the response. Concretely these responses have taken the form of *ComparisonDto* objects. A *ComparisonDto* contains an array of matches and a flag signaling if the result refers to a three-way comparison. A match describes a diagram element that has been identified across the diagram versions being compared and will only be added to the *ComparisonDto* should it or one of its submatches (as is the case with nested elements) contains a diff at a certain point down the line. The individual diffs themselves are contained as a list within the matches. The exact structure of the *ComparisonDto* and its nested elements is depicted in figure 3.9 and described in detail in chapter 3.2.3.

5.2 Evaluation

The goal of this thesis was to explore and develop the field of diagram diffing and merging in the context of online IDEs. Concretely this task involved developing a protocol to be used for the communication between the frontend and backend as well as developing useful visualization means for diagram diffs. Because of their nature, these two aspects of the implementation have been evaluated individually using different methodologies.

Because the implementation reuses a number of existing components, such as EMF Compare, or the Theia DiagramWidgets it is important to create a clear delimitation between the protocol and the rest of the components when conducting the benchmarking, at least when it comes to the response time. Whereas the network communication can be easily measured based on the created sample diagrams, the speed of the protocol and its implicit impact on UI perception need to be regarded separately from the other parts of the application. In order to achieve this clear delimitation timestamps have been recorded at key positions in the comparison / merging process, such as when an EMF Comparison object begins to be processed, when the backend sends the response, when the frontend receives the *ComparisonDTO* and when the diffs have finished rendering. By measuring at these points in the comparison process we can accurately ascertain the time required to serialize the result, send it over the network and ultimately display it to the users.

Concretely each class of diagram has been tested for a number of 50 times with the following three diagram classes being defined:

5.2.1 Small Diagrams and Change Sets

These diagrams contain a total of 50 diagram elements and have associated change sets comprised of 10 changes (additions, changes, and deletions). The network times for this class of diagram showed a mean time of 6.14 *ms* for the network transmission of the data and a mean total time of 995.12 *ms* until the comparison is fully rendered and the user can begin interacting with it. On the two opposite ends of the spectrum we have the fastest time for network transmission of 5 *ms* while the slowest one took 9 *ms*. The fastest run took 935 *ms* until the rendering finished whereas the longest running test required 1033 ms for the same task. Both the transmission as well as the total times proved to be relatively constant with standard deviations of 0.83 *ms* and 22.41 *ms* respectively.

Figure 5.1 below shows the total times recorded for the individual test runs based on small diagrams.

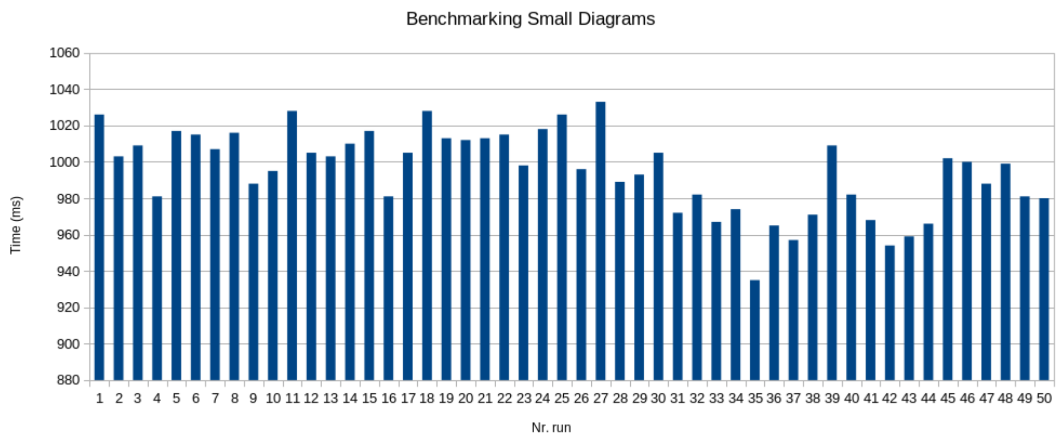


Figure 5.1: Benchmarking Small Diagrams and Change Sets

5.2.2 Medium Diagrams and Change Sets

These diagrams contain a total of 250 diagram elements while their change sets contain a number of 50 changes consisting of a multitude of additions, changes, and deletions. The network times for medium diagrams measured a mean time of 26.98 *ms* for the network transmission of the comparison and a mean total time of 2316.74 *ms* until the rendering finished. The fastest time for network transmission measured 22 *ms* while the slowest one took 32 *ms*. The fastest run took 2070 *ms* until the rendering finished whereas the slowest one required 2515 *ms*. The recorded values were once again consistent throughout the measurements with a transmission standard deviation of 2.29 *ms* and a total time standard deviation of 144.85 *ms*.

Depicted in figure 5.2 below are the total times recorded on medium diagrams.

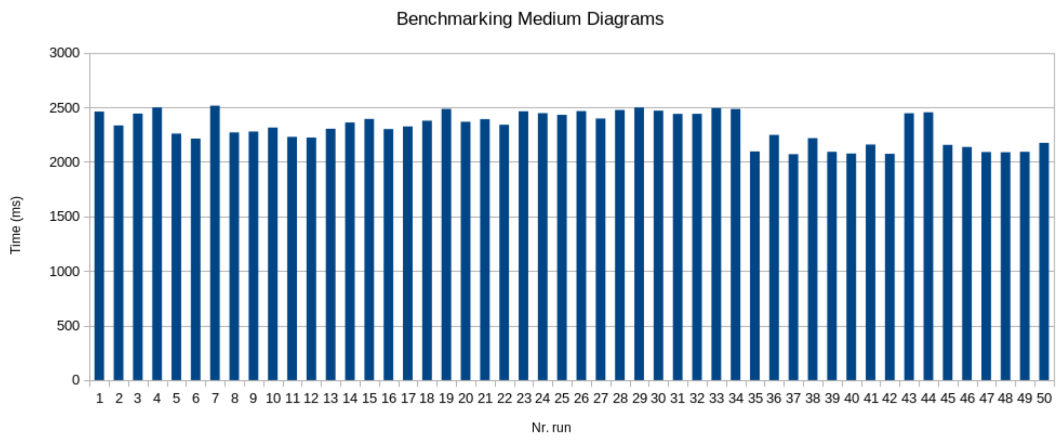


Figure 5.2: Benchmarking Medium Diagrams and Change Sets

5.2.3 Large Diagrams and Change Sets

These diagrams contain a total of 500 diagram elements coupled with a change set composed of a total number of 100 additions, deletions, and changes. Data transmission took on average 45.66 *ms* while the total time, including comparison rendering measured an average of 2935.14 *ms*. The fastest network transmission of the comparison measured 36 ms while the slowest one took 66 ms. When it comes to the total time, including rendering, we measured a shortest time of 2627 *ms* and a slowest one of 3316 *ms*. The standard deviation for the large class of diagrams is 8.12 *ms* for transmission and 194.85 *ms* for the total times.

The total times for the individual test runs performed on large diagrams are showcased in figure 5.2 below.

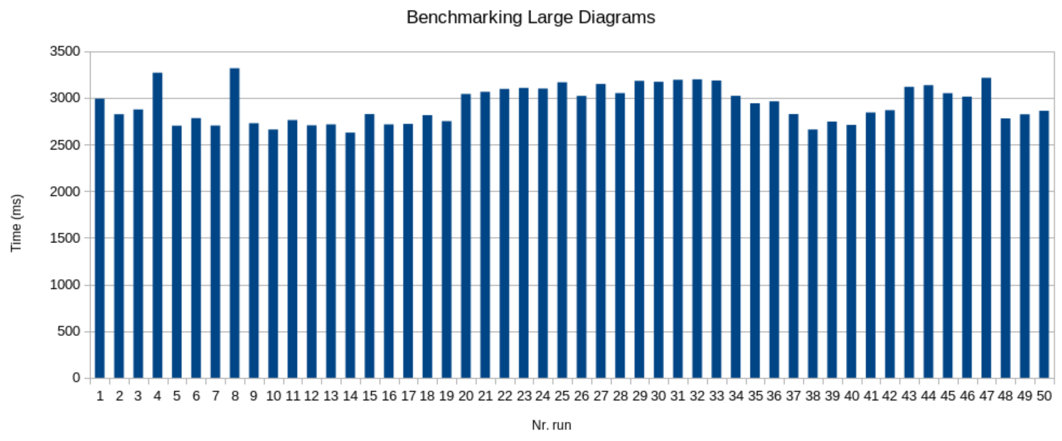


Figure 5.3: Benchmarking Large Diagrams and Change Sets

5.2.4 Visualization Usability

When it comes to the usability and usefulness of the diff visualization, the System usability scale, or SUS for short [B$^+$96] has been used. This evaluation method relies on a 10 item questionnaire in order to measure the usability of a system. Respondents are then asked to choose one of the 5 available options for each item, ranging from *Strongly agree* to *Strongly disagree*. Since its development by John Brooke in 1986, the SUS has effectively become an industry standard when it comes to evaluating the usability of a system, being referenced almost 1 million times (993.000) in literature, according to Google Scholar [goo22]. Part of its popularity can be attributed to the fact that it allows for the quick and easy collection of a user's subjective usability rating regarding a specific system. We have opted to conduct this part of the evaluation in a qualitative manner, relying upon the experience of the partners involved in the HybriDLUX project [hyb]. The developers from the industry partner AVL [avl] have extensive experience developing products using domain specific languages, or DSLs for short. The logic of these products is often represented as diagrams, similar in structure and design to a state machine, as this representation is much more concise than the textual representation of the DSL files. Because of the evolving nature of their products multiple developers often work on the same files. This fact means that the developers from AVL are in a unique position allowing them to offer qualified feedback regarding the usability of the Diff-Merge Theia extension.

A series consisting of 4 videos showcasing the developed diffing and merging functionality within the Theia IDE has been recorded to serve as a base for the SUS scale evaluation. A brief introduction of the developed extension, together with the previously mentioned 4 videos has been added to a Google Forms [goo] questionnaire alongside the 10 SUS statements. The 10 statements included in the questionnaire are as follows, each of them allowing for an answer between 1 and 5, where 1 represents *Strongly disagree* and 5 represents *Strongly agree* :

1. I think that I would like to use the system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use the system.
5. I found the various functions in the system were well integrated.
6. I thought there was too much inconsistency in the system.
7. I would imagine that most people would learn to use the system very quickly.
8. I found the system very cumbersome (awkward) to use.
9. I would feel very confident using the system.
10. I would need to learn a lot of things before I could get going with the system.

It is worth mentioning at this point that the SUS scale used for this evaluation includes a slight modification in item number 8. The term *awkward* has been included in addition to the term *cumbersome* which is usually used in such scales. This addition has been made in order to increase comprehension, as the original term proved somewhat problematic for a significant proportion of non-native English speakers, according to an article [Fin06] by Kraig Finstad from the Intel Corporation [int].

Two additional free text questions have been added to the Google Forms in order to further leverage the experience and knowledge of the AVL developers. The first question aims to gauge if the available functionality is sufficient and satisfactory for a normal workflow when diffing and merging diagrams and has been worded as follows: *How would you proceed when conducting a merge given the available functionality (Auto-merge, applying individual changes)? Would the available options be sufficient for your desired workflow?*

The second free text question is *Compared to the unified MDML Diff Visualization, what would be the advantages/disadvantages of a side-by-side representation*, and it aims to evaluate the suitability of the side-by-side representation when comparing two diagram versions. This question basically asks the respondents to compare the Diff-Merge extension with a solution they already have, namely one that presents diffs in an unified view, similar to the result of a merge where the origin of changes is marked. This form of the questionnaire has been forwarded to 8 respondents from within AVL and the results will be presented in the following paragraphs.

When asked if the provided functionality would be sufficient for their workflow, the majority answered that they are satisfied with the offered options. One respondent however has suggested that choosing either one of the changes, left or right, might not be sufficient in some cases. This suggestion will be further explored in chapter 6.3.

Regarding the advantages of a side-by-side visualization compared to a unified one, the majority of respondents consider that a side-by-side comparison generates a less complex diagram that is "*easier to handle*" whereas a unified representation would be more suitable for a high number of diffs within a complex diagram.

Circling back to the SUS we first need to conduct a small series of calculations before being able to interpret the results. The scoring of the SUS will be done on an individual basis for each respondent and then a mean score will be computed. The scoring will be conducted according to the "SUS - A quick and dirty usability scale" paper by John Brooke [B+96] with the first step being to map each item's contribution to a value between 0 and 4. For the odd numbered items this is achieved by subtracting one from the respondent's score, while for the even numbered items the contribution is 5 minus the respondent's score for that item. These scores are then summed up and multiplied by 2.5 in order to obtain the overall value of system usability on a scale from 0 to 100.

Scoring our SUS questionnaires has yielded the following results: 87.5, 92.5, 87.5, 67.5, 92.5, 77.5, 85, and 65, with a mean score of 81.87 across all participants. Now that we have these numbers it is time to convert them to more meaningful statements about the usability of the evaluated system, in our case the Diff-Merge extension. According to a retrospective by John Brooke [Bro13] a score of 68 signifies an average rating. Further milestones are also defined, namely 75 being a "*good*" score, 85 signifying an "*excellent*" score, whereas a score of 100 would be the "*best imaginable*".

Based on the achieved results, both from the benchmarking as well as the System Usability Scale scores, we can conclude that we were successful in attaining our goal of developing a useful and usable diff visualization for diagrams in the context of cloud-based IDEs, coupled with an efficient application-layer communication protocol. Potential areas of improvement have also been identified and explored in chapter 6 of this thesis.

CHAPTER 6

Open Challenges and Future Work

This chapter discusses the limitations of the current implementation and offers insight into possible improvements that could be made upon the current state. The bulk of these improvements will refer to the visualization of the diffs thus implicitly focusing on the Theia Diff-Merge extension.

6.1 Diff Granularity Setting

The current implementation relies on a hard-coded granularity setting. As described in chapter 3.2 the level on which diffs are represented in the frontend is programmatically set and changing it requires alteration of the backend code. While this change is trivial in nature, it is nonetheless somewhat cumbersome as it requires some knowledge of the EMF Compare framework as well as the Java programming language. Furthermore it cannot be done on the fly, as any change to the Diff-Merge backend would require a restart of the java application in order for changes to take effect. This has no negative impact on the current implementation however, as the WFML is limited in scope and capabilities and thus the need for variable granularity does not arise. Future work could improve upon this limitation by adding the possibility to adjust the granularity of the diffs displayed in the diff tree in order to accommodate various types of nested elements or attribute changes, without requiring any code changes.

Adding an extra parameter describing the desired setting to the requests sent to the Diff-Merge backend would allow for this setting to be changed on the fly, directly from the frontend and would not require any additional computation to be performed by the Theia extension (e.g. filtering out undesired levels) as the backend would deliver the comparison, directly containing the desired granularity level for the diffs.

A potential user interface design allowing for on the fly granularity adjustment is showcased below in figure 6.1.

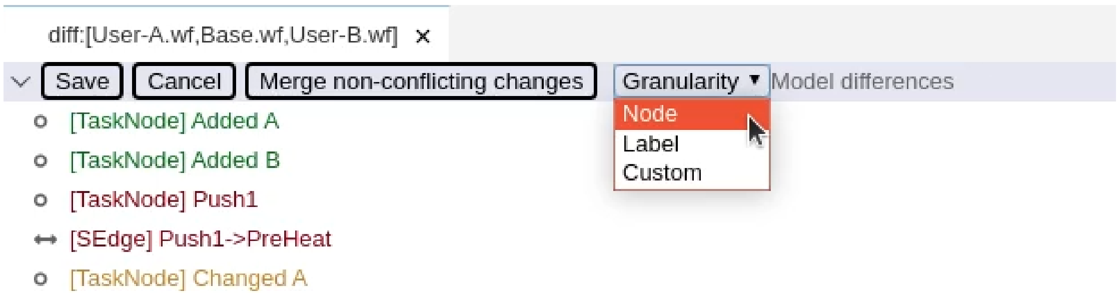


Figure 6.1: Granularity adjustment mock-up

6.2 Diff Grouping based on Level

Closely related to the ability to alter the granularity of the identified diffs would be their grouping within the diff tree itself. At the moment, as a result of the chosen granularity coupled with the nature of the WFML, diffs are represented in a flat manner within the diff tree. Only conflicts get bundled together under a common entry in order to illustrate the conflicting diffs to the user. Should the option exist to identify diffs on multiple levels within a diagram element, it would also be necessary to group said diffs as such in order to convey which diff is nested within which parent element.

A possible way of doing this would be similar to how the tree of a file explorer handles folders and how the conflicts are currently represented, as children containing the specific diff appended to the parent node. This would allow for entries with a potentially indefinite number of nested elements to be represented within the diff tree. In order to offer an overview of the diffs contained by a node's children a counter of sorts could be added to the right of the entry itself, describing the total number of additions, deletions and changes contained within the children of the specific entry.

The implementation of the aforementioned functionality could look as depicted below in figure 6.1. In this particular example, the task nodes *Added A* and *Added B* would each contain three elements, namely a dimension, a text label and an icon, and thus have a *(+3)* appended to them, to let the user know they contain 3 new additions. The deleted task node *Push1* would in turn contain 3 deletions while the task node *Changed A* contains 2 changes, namely one regarding the text label and another one regarding its dimension as it now has to accommodate the changed label.

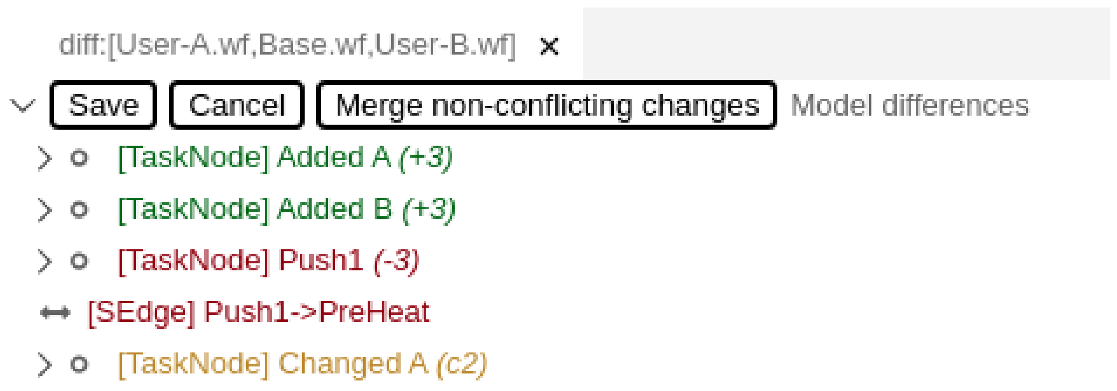


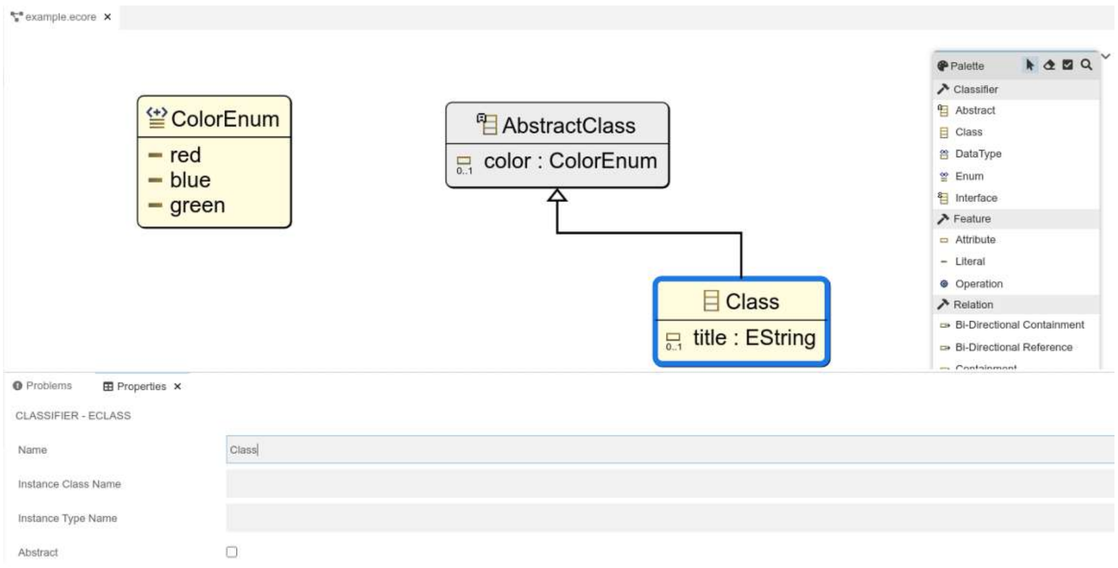
Figure 6.1: Diff grouping based on level

6.3 Additional properties view

At the moment users have the ability to construct the final merge result as they see fit, even though that ability comes with some caveats. They can currently choose which changes to apply but the ability to conduct new changes directly in the merging interface only benefits from limited support. Because the same diagram editors that allow for diagram creation are reused, with some customizations, they do retain the ability to edit the diagrams being displayed within them (in a limited capacity as the tool palette has been disabled). This ability is not explicitly advertised anywhere in the Diff-Merge extension, as it could lead to unexpected artifacts. Because new comparisons are triggered at various points during the merging process, changes made from the merging interface might get inadvertently rolled back or they might lead to the introduction of unexpected new diffs as the edits made here would be detected as changes to the files. The need to conduct new changes at merge time stems from semantic reasons, as a new, third change, might reconcile two already existing changes, from a semantic point of view. This possibility has also been brought up by one of the interviewees during the SUS evaluation.

A way to mitigate this problem would be the addition of a property view for diagram elements with a twofold goal. On one hand this view would allow users to conduct changes which could then benefit from special treatment during any subsequent comparisons conducted as part of the merging process, so that any artifacting or loss of changes is avoided. On the other hand a property view would also allow users to edit certain attributes of diagram elements that might not necessarily be displayed within the diagram itself. This aspect would be heavily dependent on the modeling language and the specific diagram representation but it remains a possibility nonetheless.

The concrete implementation of this feature could be similar to the one associated with the Theia Ecore [eco21b] modeling editor, depicted in figure 6.1 below.



CLASSIFIER - ECLASS

Name: Class

Instance Class Name

Instance Type Name

Abstract ☐

Figure 6.1: Theia Ecore Properties View [eco21a]

6.4 Source code management tool integration

In order to trigger a comparison users are required to manually select the two / three files to be compared, the two user versions and in the case of a three-way comparison / merge also the file representing the last common ancestor of the two versions, designated as the base version of the diagram. While this is sufficient for the scope of this thesis, the Diff-Merge Theia extension would stand to benefit greatly from an integration with an actual source code management system such as Git [git21a] or SVN [sub21].

Such an integration would eliminate the need for users to concern themselves with anything else, other than their own version of a diagram. Should a conflict arise when attempting to push their changes to the upstream, they would then be presented with the comparison and merge interface where they could solve it. All the required information such as the conflicting diagram version as well as the base version would be automatically fetched from the remote server where the user's file is to be pushed, thus absolving the user from having to select any other diagram version. This would greatly resemble the workflow to which many developers are accustomed to by having used source code management tools such as Git for text-based representations of code files.

The expected advantage of such an approach is a reduction in the complexity of the user's workflow and increased familiarity between the Diff-Merge extension and other versioning tools, such as the ones used for text-based representation.

List of Figures

2.1 Cloud9 UI 8
2.2 Replit UI 9
2.3 JSFiddle UI 10
2.4 Theia UI [the21] 11

3.1 Architecture Diagram 26
3.2 Sprotty Architectural Overview [spr21d] 29
3.3 Simplified GLSP metamodel and Workflow Modeling Language metamodel 30
3.4 Graphical representation of a Workflow Modeling Language Diagram 31
3.5 Side-by-side Representation and diff-tree 35
3.6 package.json file for a newly generated extension 37
3.7 package.json file for the Diff-Merge Theia extension 38
3.8 Context menu additions for diffing/merging functionality 41
3.9 ComparisonDto Structure 43
3.10 diff-tree representation 48
3.11 EMF Compare comparison process [emf21] 53
3.12 EMF Compare differencing process [emf21] 53
3.13 EMF Compare Metamodel [emf21] 55

4.1 Example of files 63
4.2 Diagram Comparison 64
4.3 Mapping to ComparisonDTO 65
4.4 Diff tree after applying all of the non conflicting changes 66
4.5 Merge result after applying all non conflicting-changes 67
4.6 Completed merge result 67

5.1 Benchmarking Small Diagrams and Change Sets 72
5.2 Benchmarking Medium Diagrams and Change Sets 72
5.3 Benchmarking Large Diagrams and Change Sets 73

6.1 Granularity adjustment mock-up 78
6.1 Diff grouping based on level 79
6.1 Theia Ecore Properties View [eco21a] 80

Bibliography

[ABK+09] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why model versioning research is needed!? an experience report. In *Proc. Models and Evolution Workshop, Denver, CO, USA*, pages 79–90. Citeseer, 2009.

[AGD18] Deniz Akdur, Vahid Garousi, and Onur Demirörs. A survey on modelling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91:62–82, 2018.

[amo] AMOR. `http://www.modelversioning.org/`. [Online; accessed 30-05-2021].

[avl] AVL. `https://avl.com`. [Online; accessed 26-01-2022].

[aws21a] AWS Discussions Forun. `https://forums.aws.amazon.com/message.jspa?messageID=834797`, 2021. [Online; accessed 30-05-2021].

[aws21b] AWS Toolkit. `https://docs.aws.amazon.com/cloud9/latest/user-guide/toolkit-welcome.html`, 2021. [Online; accessed 30-05-2021].

[B+96] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[Bal91] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering*, pages 158–165, 1991.

[BKL+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 336–398. Springer, 2012.

[BP08] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[Bro13] John Brooke. Sus: a retrospective. *Journal of usability studies*, 8(2):29–40, 2013.

[CDRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *J. Object Technol.*, 6(9):165–185, 2007.

[CK03] William Crawford and Jonathan Kaplan. *J2EE Design Patterns: Patterns in the Real World*. " O'Reilly Media, Inc.", 2003.

[clo21] AWS Cloud9. `https://aws.amazon.com/de/cloud9`, 2021. [Online; accessed 30-05-2021].

[cvs21] Concurrent Versions System. `http://www.nongnu.org/cvs`, 2021. [Online; accessed 30-05-2021].

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[DLFST09] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genoveffa Tortora. Concurrent fine-grained versioning of uml models. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 89–98. IEEE, 2009.

[doc21] Docker. `https://www.docker.com`, 2021. [Online; accessed 30-05-2021].

[DS15] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[ecl21] EclipseSource. `https://eclipsesource.com/`, 2021. [Online; accessed 30-05-2021].

[eco21a] A property view for Eclipse Theia. `https://eclipsesource.com/blogs/2021/03/10/a-property-view-for-eclipse-theia`, 2021. [Online; accessed 07-12-2021].

[eco21b] Ecore. `https://wiki.eclipse.org/Ecore`, 2021. [Online; accessed 07-12-2021].

[emf21] EMF Compare Devloper Guide. `https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html`, 2021. [Online; accessed 27-11-2021].

[ext21] Eclipse Theia - Extension Generator. `https://www.npmjs.com/package/generator-theia-extension`, 2021. [Online; accessed 05-08-2021].

[FGW+15] Kleinner Farias, Alessandro Garcia, Jon Whittle, Garcia Chavez, Christina von Flach, and Carlos Lucena. Evaluating the effort of composing design models: a controlled experiment. *Software & Systems Modeling*, 14(4):1349–1365, 2015.

[Fin06] Kraig Finstad. The system usability scale and non-native english speakers. *Journal of usability studies*, 1(4):185–188, 2006.

[fuj21] Fujaba Tool Suite. `https://web.cs.upb.de/archive/fujaba`, 2021. [Online; accessed 30-05-2021].

[Gir06] Martin Girschick. Difference detection and visualization in UML class diagrams. *Technical university of Darmstadt technical report TUD-CS-2006-5*, pages 1–15, 2006.

[git21a] git. `https://git-scm.com`, 2021. [Online; accessed 30-05-2021].

[git21b] GitHub. `https://github.com`, 2021. [Online; accessed 30-05-2021].

[GKLE10] Christian Gerth, Jochen M Küster, Markus Luckey, and Gregor Engels. Precise detection of conflicting change operations using process model terms. In *International Conference on Model Driven Engineering Languages and Systems*, pages 93–107. Springer, 2010.

[GKLE13] Christian Gerth, Jochen M Küster, Markus Luckey, and Gregor Engels. Detection and resolution of conflicting change operations in version management of process models. *Software & Systems Modeling*, 12(3):517–535, 2013.

[gls21a] Eclipse GLSP Examples. `https://github.com/eclipse-glsp/glsp-examples`, 2021. [Online; accessed 05-08-2021].

[gls21b] Graphical Language Server Framework. `https://github.com/eclipse-glsp/glsp`, 2021. [Online; accessed 26-03-2021].

[goo] Google Forms. `https://www.google.com/forms/about`. [Online; accessed 26-01-2022].

[goo22] Jersey. `https://scholar.google.com/`, 2022. [Online; accessed 25-02-2022].

[gtr21] Google Trends. `https://trends.google.com/trends`, 2021. [Online; accessed 30-05-2021].

[GWKRM15] Manuel Gall, Günter Wallner, Simone Kriglstein, and Stefanie Rinderle-Ma. A study of different visualizations for visualizing differences in process models. In *International Conference on Conceptual Modeling*, pages 99–108. Springer, 2015.

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

[hyb] HybriDLUX. `https://hybridlux.wu.ac.at`. [Online; accessed 26-01-2022].

[ibm21] IBM Rational Software Architect. `https://www.ibm.com/products/rational-software-architect-designer`, 2021. [Online; accessed 30-05-2021].

[int] Intel. `https://www.intel.com/`. [Online; accessed 26-01-2022].

[inv21] InversifyJS. `https://github.com/inversify/InversifyJS/blob/master/wiki/container_modules.md`, 2021. [Online; accessed 05-08-2021].

[jer21] Jersey. `https://eclipse-ee4j.github.io/jersey/`, 2021. [Online; accessed 30-05-2021].

[jet] Jetty. `https://www.eclipse.org/jetty/`. [Online; accessed 26-03-2021].

[jku21] Johannes Kepler Universität Linz. `http://www.jku.at`, 2021. [Online; accessed 30-05-2021].

[jsf21] JSFiddle. `https://jsfiddle.net/m`, 2021. [Online; accessed 30-05-2021].

[jso21] JSON-RPC. `https://www.jsonrpc.org/specification`, 2021. [Online; accessed 30-05-2021].

[KDRPP09] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009.

[KHL$^+$10] Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, and Joern David. Comparing state- and operation-based change tracking on models. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 163–172. IEEE, 2010.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172. IEEE Computer Society, 2011.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. Dsmdiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007.

[mav21] Apache Maven. `https://maven.apache.org`, 2021. [Online; accessed 30-05-2021].

[MD94] Jonathan P Munson and Prasun Dewan. A flexible object merging framework. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 231–242, 1994.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[MGH05] 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*, pages 204–213, 2005.

[MM15] Sonja Maier and Mark Minas. Recording, processing, and visualizing changes in diagrams. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, Atlanta, GA, USA, October 18-22, 2015*, pages 131–135. IEEE, 2015.

[MNSD17] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 467–478. IEEE, 2017.

[mof05] Meta Object Facility (MOF) Specification. `https://www.omg.org/spec/MOF/ISO/19502/PDF`, 2005. [Online; accessed 30-05-2021].

[mof14] Information technology - Object Management Group Meta Object Facility (MOF) Core. `https://www.omg.org/spec/MOF/ISO/19502/PDF`, 2014. [Online; accessed 30-05-2021].

[mon21] Monaco Editor. `https://microsoft.github.io/monaco-editor`, 2021. [Online; accessed 05-08-2021].

[NER01] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of systems and software*, 58(2):171–180, 2001.

[nod21] Node Fetch. `https://github.com/node-fetch/node-fetch`, 2021. [Online; accessed 05-08-2021].

[npm21] npm. `https://www.npmjs.com`, 2021. [Online; accessed 30-05-2021].

[NSC+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 54–64. IEEE, 2007.

[ode21] TOP ODE Index. `http://pypl.github.io/ODE.html`, 2021. [Online; accessed 30-05-2021].

[OWK03] 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*, pages 227–236. ACM, 2003.

[pap21] Eclipse Papyrus. `https://www.eclipse.org/papyrus`, 2021. [Online; accessed 05-08-2021].

[poe21] Poetry. `https://python-poetry.org`, 2021. [Online; accessed 30-05-2021].

[Por05] Ivan Porres. Rule-based update transformations and their application to model refactorings. *Software & Systems Modeling*, 4(4):368–385, 2005.

[RAB+07] Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. Models in conflict-detection of semantic conflicts in model-based development. *MDEIS@ ICEIS*, 7:29–40, 2007.

[RB01] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

[rep21] Replit. `https://replit.com`, 2021. [Online; accessed 30-05-2021].

[res21] RESTful API. `https://www.redhat.com/en/topics/api/what-is-a-rest-api`, 2021. [Online; accessed 30-05-2021].

[S+00] Richard Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SG08] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the sidiff framework. In *Companion of the 30th international conference on Software engineering*, pages 947–948, 2008.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[spa21] Sparx Systems. https://www.sparxsystems.eu/, 2021. [Online; accessed 30-05-2021].

[spr21a] Eclipse Sprotty. https://www.eclipse.org/community/eclipse_newsletter/2018/october/sprotty.php, 2021. [Online; accessed 05-08-2021].

[spr21b] sprotty. https://github.com/eclipse/sprotty, 2021. [Online; accessed 26-03-2021].

[spr21c] Sprotty. https://github.com/eclipse/sprotty/wiki, 2021. [Online; accessed 05-08-2021].

[spr21d] Sprotty Architectural Overview. https://github.com/eclipse/sprotty/wiki/Architectural-Overview, 2021. [Online; accessed 27-11-2021].

[sta21] StackOverflow. https://stackoverflow.com, 2021. [Online; accessed 30-05-2021].

[sub21] Apache Subversion. https://subversion.apache.org, 2021. [Online; accessed 30-05-2021].

[SZN04] Christian Schneider, Albert Zündorf, and Jörg Niere. Coobra-a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments*. Citeseer, 2004.

[TELW14] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13(1):239–272, 2014.

[the21] Theia - Cloud and Desktop IDE. https://www.theia-ide.org/doc/index.html, 2021. [Online; accessed 26-03-2021].

[Tou06] Antoine Toulmé. Presentation of emf compare utility, position paper at eclipse summit. *Esslingen, Germany, October*, pages 11–12, 2006.

[tuw21] Technische Universität Wien. https://www.tuwien.at, 2021. [Online; accessed 30-05-2021].

[vsc21] Visual Studio Code. https://code.visualstudio.com, 2021. [Online; accessed 30-05-2021].

[Wes10] Bernhard Westfechtel. A formal approach to three-way merging of emf models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 31–41, 2010.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, 2005.

[yeo21] Yeoman. `https://yeoman.io`, 2021. [Online; accessed 05-08-2021].

Appendix

Table 1: Benchmarking results for diagrams with 50 elements and 10 changes

Nr.	**Computed comparison - Backend**	**Received comparison - Frontend**	**Rendered comparison**	**Trans-mission (in ms)**	**Rendering (in ms)**	**TOTAL (in ms)**
1	1645290229375	1645290229382	1645290230401	7	1019	1026
2	1645290235296	1645290235303	1645290236299	7	996	1003
3	1645291320221	1645291320228	1645291321230	7	1002	1009
4	1645291329286	1645291329293	1645291330267	7	974	981
5	1645291333759	1645291333765	1645291334776	6	1011	1017
6	1645291337431	1645291337438	1645291338446	7	1008	1015
7	1645291345386	1645291345392	1645291346393	6	1001	1007
8	1645291349586	1645291349593	1645291350602	7	1009	1016
9	1645291353829	1645291353836	1645291354817	7	981	988
10	1645291358085	1645291358092	1645291359080	7	988	995
11	1645291366519	1645291366526	1645291367547	7	1021	1028
12	1645291383003	1645291383010	1645291384008	7	998	1005
13	1645291386987	1645291386993	1645291387990	6	997	1003
14	1645291395145	1645291395151	1645291396155	6	1004	1010
15	1645291408830	1645291408835	1645291409847	5	1012	1017
16	1645291645118	1645291645124	1645291646099	6	975	981
17	1645291648891	1645291648897	1645291649896	6	999	1005
18	1645291653035	1645291653040	1645291654063	5	1023	1028
19	1645291661402	1645291661408	1645291662415	6	1007	1013
20	1645291674415	1645291674421	1645291675427	6	1006	1012
21	1645291700603	1645291700609	1645291701616	6	1007	1013
22	1645291705325	1645291705331	1645291706340	6	1009	1015
23	1645291723702	1645291723708	1645291724700	6	992	998
24	1645291727693	1645291727699	1645291728711	6	1012	1018
25	1645291736346	1645291736352	1645291737372	6	1020	1026
26	1645373033807	1645373033813	1645373034803	6	990	996

Nr.	Computed comparison - Backend	Received comparison - Frontend	Rendered comparison	Trans-mission (in ms)	Rendering (in ms)	TOTAL (in ms)
27	1645373037477	1645373037482	1645373038510	5	1028	1033
28	1645373044342	1645373044348	1645373045331	6	983	989
29	1645373047597	1645373047603	1645373048590	6	987	993
30	1645373051065	1645373051074	1645373052070	9	996	1005
31	1645373054443	1645373054449	1645373055415	6	966	972
32	1645373057626	1645373057633	1645373058608	7	975	982
33	1645373061154	1645373061160	1645373062121	6	961	967
34	1645373064611	1645373064617	1645373065585	6	968	974
35	1645373067913	1645373067919	1645373068848	6	929	935
36	1645373071545	1645373071550	1645373072510	5	960	965
37	1645373075083	1645373075091	1645373076040	8	949	957
38	1645373078829	1645373078834	1645373079800	5	966	971
39	1645373082386	1645373082393	1645373083395	7	1002	1009
40	1645373085808	1645373085813	1645373086790	5	977	982
41	1645373092867	1645373092872	1645373093835	5	963	968
42	1645373096395	1645373096401	1645373097349	6	948	954
43	1645373100267	1645373100272	1645373101226	5	954	959
44	1645373104040	1645373104046	1645373105006	6	960	966
45	1645373110831	1645373110837	1645373111833	6	996	1002
46	1645373114564	1645373114569	1645373115564	5	995	1000
47	1645373117993	1645373117999	1645373118981	6	982	988
48	1645373122103	1645373122109	1645373123102	6	993	999
49	1645373125534	1645373125540	1645373126515	6	975	981
50	1645373128764	1645373128769	1645373129744	5	975	980
AVERAGE				6,14	988,98	995,12
MINIMUM				5	929	935
MAXIMUM				9	1028	1033
STANDARD DEVIATION				0,83		22,41

Table 2: Benchmarking results for diagrams with 250 elements and 50 changes

Nr.	Computed comparison - Backend	Received comparison - Frontend	Rendered comparison	Trans-mission (in ms)	Rendering (in ms)	TOTAL (in ms)
1	1645293017371	1645293017397	1645293019831	26	2434	2460
2	1645293024665	1645293024697	1645293026998	32	2301	2333
3	1645293030587	1645293030618	1645293033029	31	2411	2442
4	1645293045201	1645293045230	1645293047700	29	2470	2499
5	1645293052184	1645293052216	1645293054443	32	2227	2259
6	1645293058604	1645293058628	1645293060817	24	2189	2213
7	1645293086917	1645293086942	1645293089432	25	2490	2515
8	1645293094090	1645293094115	1645293096360	25	2245	2270
9	1645293108194	1645293108221	1645293110472	27	2251	2278
10	1645293116759	1645293116786	1645293119073	27	2287	2314
11	1645293123302	1645293123330	1645293125531	28	2201	2229
12	1645293131113	1645293131142	1645293133335	29	2193	2222
13	1645293138335	1645293138362	1645293140638	27	2276	2303
14	1645293144752	1645293144778	1645293147113	26	2335	2361
15	1645293151996	1645293152025	1645293154389	29	2364	2393
16	1645293159278	1645293159306	1645293161578	28	2272	2300
17	1645293165697	1645293165728	1645293168021	31	2293	2324
18	1645293171885	1645293171912	1645293174262	27	2350	2377
19	1645293382037	1645293382062	1645293384522	25	2460	2485
20	1645293387904	1645293387932	1645293390271	28	2339	2367
21	1645293406423	1645293406453	1645293408814	30	2361	2391
22	1645293431569	1645293431595	1645293433909	26	2314	2340
23	1645293468770	1645293468795	1645293471233	25	2438	2463
24	1645293474725	1645293474751	1645293477171	26	2420	2446
25	1645293513898	1645293513925	1645293516329	27	2404	2431
26	1645293520273	1645293520298	1645293522738	25	2440	2465

Nr.	Computed comparison - Backend	Received comparison - Frontend	Rendered comparison	Trans-mission (in ms)	Rendering (in ms)	TOTAL (in ms)
27	1645293526643	1645293526668	1645293529040	25	2372	2397
28	1645293532289	1645293532314	1645293534764	25	2450	2475
29	1645293538453	1645293538480	1645293540951	27	2471	2498
30	1645293545093	1645293545122	1645293547562	29	2440	2469
31	1645293550890	1645293550918	1645293553329	28	2411	2439
32	1645293556684	1645293556713	1645293559124	29	2411	2440
33	1645293563132	1645293563160	1645293565624	28	2464	2492
34	1645293569200	1645293569227	1645293571684	27	2457	2484
35	1645294242688	1645294242717	1645294244783	29	2066	2095
36	1645294248953	1645294248984	1645294251200	31	2216	2247
37	1645294254766	1645294254794	1645294256836	28	2042	2070
38	1645294260870	1645294260895	1645294263087	25	2192	2217
39	1645294266421	1645294266448	1645294268513	27	2065	2092
40	1645294272655	1645294272680	1645294274731	25	2051	2076
41	1645294278353	1645294278382	1645294280512	29	2130	2159
42	1645294283569	1645294283592	1645294285642	23	2050	2073
43	1645294289569	1645294289594	1645294292014	25	2420	2445
44	1645294295748	1645294295774	1645294298202	26	2428	2454
45	1645294301821	1645294301846	1645294303976	25	2130	2155
46	1645294307469	1645294307494	1645294309605	25	2111	2136
47	1645294313705	1645294313732	1645294315795	27	2063	2090
48	1645294319443	1645294319468	1645294321531	25	2063	2088
49	1645294325128	1645294325150	1645294327220	22	2070	2092
50	1645294330730	1645294330754	1645294332904	24	2150	2174
AVERAGE				26,98	2289,76	2316,74
MINIMUM				22	2042	2070
MAXIMUM				32	2490	2515
STANDARD DEVIATION				2,29		144,85

Table 3: Benchmarking results for diagrams with 500 elements and 100 changes

Nr.	Computed comparison - Backend	Received comparison - Frontend	Rendered comparison	Trans-mission (in ms)	Rendering (in ms)	TOTAL (in ms)
1	1645371333144	1645371333194	1645371336135	50	2941	2991
2	1645371339912	1645371339955	1645371342736	43	2781	2824
3	1645371347303	1645371347361	1645371350178	58	2817	2875
4	1645371354908	1645371354946	1645371358177	38	3231	3269
5	1645371362563	1645371362599	1645371365264	36	2665	2701
6	1645371369395	1645371369442	1645371372177	47	2735	2782
7	1645371376922	1645371376961	1645371379624	39	2663	2702
8	1645371384294	1645371384331	1645371387610	37	3279	3316
9	1645371392019	1645371392062	1645371394746	43	2684	2727
10	1645371399685	1645371399721	1645371402345	36	2624	2660
11	1645371406516	1645371406554	1645371409277	38	2723	2761
12	1645371413820	1645371413860	1645371416525	40	2665	2705
13	1645371420800	1645371420837	1645371423515	37	2678	2715
14	1645371427118	1645371427155	1645371429745	37	2590	2627
15	1645371433738	1645371433786	1645371436563	48	2777	2825
16	1645371440578	1645371440616	1645371443293	38	2677	2715
17	1645371471869	1645371471909	1645371474589	40	2680	2720
18	1645371478198	1645371478240	1645371481012	42	2772	2814
19	1645371484788	1645371484829	1645371487538	41	2709	2750
20	1645371493161	1645371493205	1645371496202	44	2997	3041
21	1645371500296	1645371500339	1645371503360	43	3021	3064
22	1645371516419	1645371516474	1645371519514	55	3040	3095
23	1645371523894	1645371523946	1645371527000	52	3054	3106
24	1645371532028	1645371532074	1645371535127	46	3053	3099
25	1645371823056	1645371823094	1645371826222	38	3128	3166
26	1645371830249	1645371830287	1645371833269	38	2982	3020

Nr.	Computed comparison - Backend	Received comparison - Frontend	Rendered comparison	Trans-mission (in ms)	Rendering (in ms)	TOTAL (in ms)
27	1645371837124	1645371837161	1645371840272	37	3111	3148
28	1645371844444	1645371844494	1645371847494	50	3000	3050
29	1645371851501	1645371851557	1645371854682	56	3125	3181
30	1645371858734	1645371858794	1645371861905	60	3111	3171
31	1645371881346	1645371881399	1645371884539	53	3140	3193
32	1645371902906	1645371902949	1645371906103	43	3154	3197
33	1645372343474	1645372343532	1645372346659	58	3127	3185
34	1645372350606	1645372350672	1645372353627	66	2955	3021
35	1645372357687	1645372357751	1645372360628	64	2877	2941
36	1645372365848	1645372365894	1645372368810	46	2916	2962
37	1645372372999	1645372373044	1645372375824	45	2780	2825
38	1645372379676	1645372379724	1645372382336	48	2612	2660
39	1645372386209	1645372386256	1645372388954	47	2698	2745
40	1645372392810	1645372392853	1645372395519	43	2666	2709
41	1645372400468	1645372400519	1645372403310	51	2791	2842
42	1645372406927	1645372406971	1645372409794	44	2823	2867
43	1645372423568	1645372423610	1645372426685	42	3075	3117
44	1645372430764	1645372430820	1645372433899	56	3079	3135
45	1645372437955	1645372438001	1645372441004	46	3003	3049
46	1645372460830	1645372460879	1645372463842	49	2963	3012
47	1645372487105	1645372487168	1645372490319	63	3151	3214
48	1645372517333	1645372517369	1645372520112	36	2743	2779
49	1645372524003	1645372524041	1645372526826	38	2785	2823
50	1645372530468	1645372530506	1645372533329	38	2823	2861
AVERAGE				45,66	2889,48	2935,14
MINIMUM				36	2590	2627
MAXIMUM				66	3279	3316
STANDARD DEVIATION				8,12		194,85