An Architecture Comparison Framework for Software Project Visualization

Malte Mannott

Bachelor Thesis
March 27, 2025

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M. Sc. Niklas Rentz

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Communicating and visualizing changes in software projects are a time consuming and difficult task to accomplish with accuracy. Still, software projects grow in size and change every day and those changes have to implemented by developers for which accurate information is key.

OSGiViz is generated from SPViz, a generator for customized visualization tools, and OSGiViz is a visualizer, for software projects written with the OSGi framework. However, it lacks the ability to compare different models to each other. Versions of software, different states or source control could be compared.

This thesis implements such a comparison framework for OSGiViz with the possibility of abstraction into SPViz. The comparison framework enables developers to write and read OSGi models in a human-friendly and compact way and compare two models using new concepts, such as a visual comparison in a visual diff. This makes the visualization of differences easy and compact.

Acknowledgements

First of all I want to thank Prof. Dr. Reinhard von Hanxleden as the head of the working group. He not only welcomed me for this thesis as the end of my university studies, but also provided me with a working space that was available at all times. The nudges he gave, always sent me in the right direction.

I also want to express my gratitude to my supervisor Niklas Rentz, hopefully soon Dr. Niklas Rentz, for always helping and really working with me to give me a chance at writing a compelling thesis. He not only helped with writing this thesis, but also showed me how software development is done, outside of theoretical university exercises.

Thanks also go to Sören Domrös, Maximilian Kasperowski, Jette Petzold, Alexander Schulz-Rosengarten, and my fellow students that wrote their thesis at the same time and working group as me for the valuable ideas, feedback, and all around fun company.

Contents

List of Figures

Acronyms

SPViz	Software Project Visualization
OSGi	Open Services Gateway initiative
KIELER	Kiel Integrated Enviroment for Layout Eclipse RichClient
IDE	integrated development enviroment
KLighD	KIELER Lightweight Diagrams
DSL	domain-specific language
EMF	Eclipse Modeling Framework
OSGiViz	visualization for OSGi projects
XML	Extensible Markup Language
UML	Unified Modeling Language
ER	entity-relationship
oAW	openArchitectureWare
ELK	Eclipse Layout Kernel
VCM	visualization context model
ADL	architecture description language

Chapter 1

Introduction

Software projects grow in size every day. New features added to improve or widen the effectiveness, different dependency structures, bug fixes or security interventions are implemented by developers. The desired changes are often communicated in functional specification documents in the form of diagrams.

Currently the developers create Unified Modeling Language (UML)/entity-relationship (ER) diagrams by hand, use plugins for an integrated development enviroment (IDE) or external tools made specifically for reverse engineering or code execution tracing. UML/ER diagrams are used in class diagrams, sequence diagrams and state charts which not only show the current state but can display the future structure as well. Plugins visualize the program structure by recording code execution traces and displaying the way all parts of the structure work together. Another option is the usage of software reflexion models [MNS01] which map structure dependencies based on system artifacts. These are excellent tools for displaying the current project structure, yet they lack the ability to display a desired state. Most of the options available are without a concrete comparison framework and are therefore limited in their usability for representations in specification documents.

The visualization for OSGi projects (OSGiViz) [RDH20] introduces a way to visualize the software project structure implemented with the Open Services Gateway initiative (OSGi) [TV08] framework. The visualization works by displaying the dependency structure and hierarchy in the form of diagrams with directional arrows which are especially advantagous in presentations of dependencies between project parts. Diagrams display nested hierarchies by visually encapsulating bundles into their corresponding features. The existing visualization works as the basis of this comparison framework and displays the current state of the project in the comparison visualization.

This comparison framework enables developers to quickly distinguish the differences in the dependency structures. The ability to create comparisons is a fitting tool for representations in specification documents or progress reports and for that reason is highly beneficial. All this can be abstracted into Software Project Visualization (SPViz) and is thereby usable for all software architectures regardless of framework.

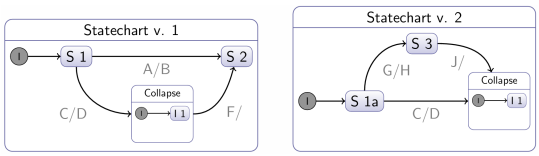
1.1 Problem Statement

The goal is to able to compare two different OSGi models via the current state of the project generated by OSGiViz and a model specifiable by humans. Currently the model generated

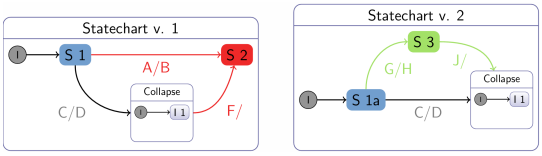
by OSGiViz is not specifiable by humans and I therefore have to introduce a new way of specifying a model. The model represented by the domain-specific language (DSL) works as the desired state and thereby the current state can be compared to the desired state. This enables developers to compare two states of the same project for progress reports or visualization in specification documents. I have to decide which layout is most practicle for the comparison and find a way to graphically compare two versions with each other.

1.2 Related Work

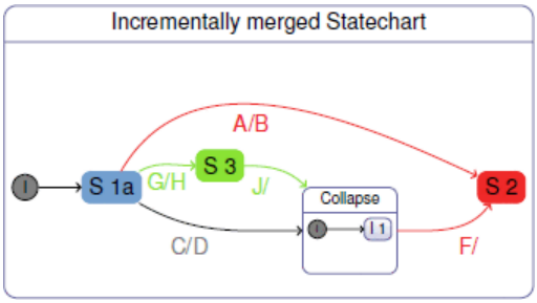
Arne Schipper et al. [SFH09] introduce multiple layouts for the visual comparison of two diagrams. Layouts differ from two diagrams next to each other with colors highlighting changes to a freely or incrementally merged visual diff which only displays one diagram with colors highlighting their differences aswell.



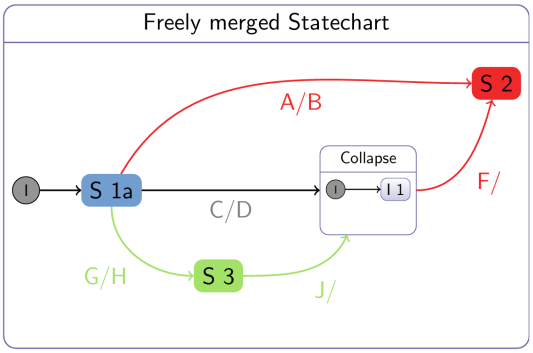
(a) Original Diagram



(b) Plain diff with red marking deletion, green addition and blue new position



(c) incrementally merged diff



(d) Freely merged diff

Figure 1.1. Layout diff by Arne Schipper et al. [SFH09]

Most important for us is the *plain* (Figure 1.1b) visual diff and its color scheme. Furthermore, implementation details in regard to finding differences and displaying the differences, in multiple ways, are described.

There are tools to generate code from UML diagrams and visualize code as a UML diagram as well. One such example is *GitUML*[1]. Fiddle, the tool behind *GitUML*, generates diagrams from source code and can generate code from diagrams too. The tool displays the structure of an entire codebase and can track changes over time and visualize them if needed. *GitUML*

[1]https://www.gituml.com/

supports adding new dependencies and classes during runtime which will work as a basis for this comparison framework.

Another option for reverse engineering source code is the *IntelliJ Idea*[2] plugin *AppMap*[3]. *AppMap* displays searchable diagrams from records of code execution traces. *AppMap* can locate dependencies and visualize them if needed. This works for any architecture.

Niklas Rentz et al. [RH25] implemented SPViz. SPViz developed at the Real-Time and Embedded Systems Group at Kiel University's Department of Computer Science introduced a way to visualize software project structures as diagrams using domain-specific languages (DSL). SPViz can generate visualizations for any project structure and the basis of this thesis, OSGiViz, is designed inside the SPViz tool.

1.3 Outline

Chapter 2 explains all the technologies and frameworks used in the making of this thesis. Chapter 3 discusses the various concepts on how the comparison should be visualized. Simple and complex examples are shown in this chapter aswell as different schemes of visualization. The implementation and its difficulties are shown in Chapter 4, while Chapter 5 evaluates the implementation completion with a look at the proposed goals. At last, Chapter 6 gives a brief conclusion and a look into future work.

[2]https://www.jetbrains.com/de-de/idea/
[3]https://appmap.io/

Chapter 2

Used Technologies

This chapter introduces the most important technologies necessary to understand the thesis implementation of this thesis. Each section contains a single framework or programming language.

2.1 Xtext

Xtext[1] is a framework used in the development of DSLs and programming languages. The framework is developed by the Eclipse Foundation as part of the Eclipse Modeling Framework (EMF). Xtext is a grammar language that not only implements itself but is well documented as well and can be used to describe the syntax of personalized programming languages or DSLs. The previously mentioned dialect of Java, Xtend, is implemented in Xtext. They are developed simultaneously to ensure their compatibility, which is important for multiple implementation steps of this thesis.

Xtext is used twice during the implementation of this thesis. Both times, Xtext is used to define a DSL and to integrate them into a software project.

First released as an openArchitectureWare (oAW) project by Efftinge et al. [EV06] it has since evolved and established itself as part of the open source Eclipse ecosystem. The complete user guide and examples can be found on the Xtext website[2].

2.2 EMF

The Eclipse Modeling Framework (EMF) is a framework used for modeling structured data. Code generation can be done with this framework as well. As part of the Eclipse modeling project, EMF is developed under the supervision of Ed Merks[3]. The modeling is done via, for example, the previously mentioned Xtext framework or Ecore and thereby abstracts from implementing models [Bud04]. EMF ensures the interoperability between all technologies under its hood, which makes this framework highly flexible and versatile in almost all modeling operations.

EMF is used during this thesis to describe and generate models, as well as accessing them. EMF connects different parts of the implementation by the ensured interoperability and thus

[1] https://eclipse.dev/Xtext/

[2] https://eclipse.dev/Xtext/documentation/index.html

[3] https://projects.eclipse.org/projects/modeling.emf.emf/who

allows us to work with different models that best fit each situation.

This framework was first released in 2003 as part of the open-source Eclipse ecosystem and has since received updates to this day. A guide can be found in the Eclipse wiki[4].

2.3 KLighD

KIELER Lightweight Diagrams (KLighD) is a framework used to generate graphical representations of models. Development is done by the Real-Time and Embedded Systems group at Kiel University as part of Kiel Integrated Enviroment for Layout Eclipse RichClient (KIELER).

KLighD works by synthesizing a model recursivly and implementing a specific synthesis for each and adding a layout with Eclipse Layout Kernel (ELK). The synthesis turns the source data of the model into KGraph instances, which describe the visual representation of the visual node.

KLighD is used during this thesis to synthesize a graphical representation of a model, allowing us to visualize and interact with it.

Current development is done by the Real-Time and Embedded Systems group at Kiel University, and the focus is now on the graphical representations. A guide and tutorial can be found on the KIELER GitHub wiki[5].

2.4 OSGiViz

The visualization for OSGi projects (OSGiViz) is a visualizer for software projects implemented with the OSGi framework. It cannot visualize the projects directly, but instead visualizes a textual representation that follows an OSGi model. The textual representation is then turned into a top-level OSGi project as a complex data structure, which is then visualized using this visualizer. The visualization is done with KLighD [RDH20].

OSGiViz is the basis of the entire implementation, and this thesis aims to enhance the current version of it.

OSGiViz was first developed by Rentz et al. [RDH20] as a proposal for the visualization of OSGi projects. OSGiViz is no longer in direct active development, as it is now part of SPViz. SPViz is able to generate OSGiViz and therefore the results of this thesis are to be abstracted into SPViz. The last non-generated version of OSGiViz and a user guide can be found on their GitHub[6].

[4]https://wiki.eclipse.org/
[5]https://github.com/kieler/KLighD/
[6]https://github.com/kieler/osgiviz?tab=readme-ov-file

2.5 SPViz

Software Project Visualization (SPViz) is a generator for customized visualization tools of arbitrary software architectures [RH25]. This tool generator uses two DSLs to firstly describe the structure and secondly to configure the visualization. The visualization style is derived from the previously mentioned OSGiViz. The important step is the abstraction from a visualizer for a specific architecture to a generator for visualizations for any architecture while keeping the simple dependency structure and bundle hierarchy of OSGiViz. SPViz generates the visualization in multiple parts.

1) A template for a generator that reads the project files and outputs a corresponding model file.

2) The visualizer that turns the model file into a graph with KLighD as the underlying graph framework

3) A language server for viewing the visualization in web environments.

A Maven build system completes the visualizer to a full-fledged Eclipse plug-in or executable.

SPViz generated the used version of OSGiViz on which this thesis is based on. That way it created the groundwork of nearly every implementation element. Every change will ultimately be abstracted into SPViz.

Developed by Rentz et al. [RH25] as an academic project, SPViz is already used in the industry for clarifying software structures and is available as an open-source library[7]. More info as well as a user guide can be found on the SPViz GitHub wiki[8].

[7] https://github.com/kieler/SoftwareProjectViz
[8] https://github.com/kieler/SoftwareProjectViz/wiki

Chapter 3

Concepts

This chapter discusses the design decisions made during this thesis and their impact. It introduces the ideas behind the implementation and aims to show the different approaches to problems that arose during the implementation. Two main problems had to be solved. Firstly the construction of a target architecture model that allows for handwritten modeling is introduced and discussed. The model is implemented as a DSL that has the same structure as the source architecture. After a model can be handwritten, the next step is to compare the two different versions of OSGi models, for which I introduce two methods and discuss which one more closely aligns with our goals. At first, this chapter explains the process behind the implementation of the desired state.

3.1 Desired State

Implementing the desired state is the first step to create the architecture comparison framework and the first hurdle of our framework. The desired state represents the project architecture as it could be without having to implement everything. It is not even necessary to implement a software skeleton. This is important because the generator for the current states needs implemented code to generate models out of. You can just write down what you want your project architecture to resemble. The written desired state has to conform to the same pattern of the OSGi model as the current state to ensure compatibility with the synthesis. The goal is a DSL that is capable of creating the same models as the OSGiViz generator but can be manually written without implementation needs.

The OSGi architecture model that the OSGiViz tool generated from SPViz as seen in Figure 3.1:

- OSGi *Project*: The all-encompassing root contains the project name and everything else.
- *Product*: A product comprises features and bundles.
- *Feature*: A feature consists of bundles.
- *Bundle*: A bundle incorporates the dependencies between itself and other bundles. A bundle also holds the service interfaces, service components and packages it is part of.
- *ServiceInterface*: A service interface holds the service components it is required by.
- *ServiceComponent*: A service component features the service interfaces it is provided by.

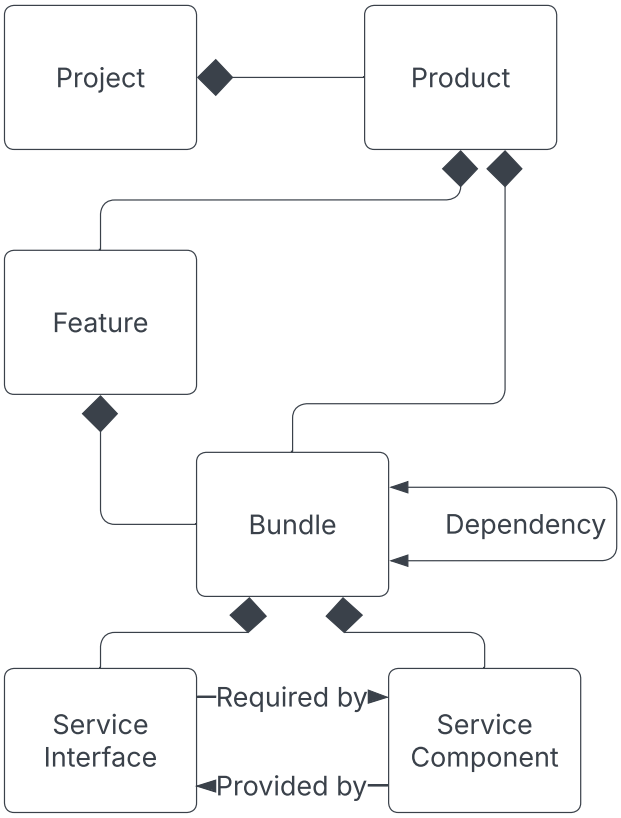


Figure 3.1. The OSGi architecture as a UML diagram

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <model:OSGiProject xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
       xmlns:model="de.cau.cs.kieler.spviz.osgi.model" projectName="languagedemo_slim">
3    <products ecoreId="Product_englishtranslator" name="englishtranslator"
         features="Feature_de.scheidtbachmann.osgi.language.runtime.feature
         Feature_de.scheidtbachmann.osgi.language.base.feature"
         bundles="Bundle_ch.qos.logback.classic Bundle_ch.qos.logback.core"/>
4    <features ecoreId="Feature_de.scheidtbachmann.osgi.language.runtime.feature"
         name="de.scheidtbachmann.osgi.language.runtime.feature"
         products="Product_englishtranslator" bundles="Bundle_ch.qos.logback.classic"/>
5    <features ecoreId="Feature_de.scheidtbachmann.osgi.language.base.feature"
         name="de.scheidtbachmann.osgi.language.base.feature"
         products="Product_englishtranslator" bundles="Bundle_ch.qos.logback.core"/>
6    <bundles ecoreId="Bundle_ch.qos.logback.classic" name="ch.qos.logback.classic"
         external="false" products="Product_englishtranslator"
         features="Feature_de.scheidtbachmann.osgi.language.runtime.feature"
```

```
        connectingDependencyBundles="Bundle_ch.qos.logback.core"/>
7     <bundles ecoreId="Bundle_ch.qos.logback.core" name="ch.qos.logback.core"
        external="false" products="Product_englishtranslator"
        features="Feature_de.scheidtbachmann.osgi.language.base.feature"
        connectedDependencyBundles="Bundle_ch.qos.logback.classic"/>
8   </model:OSGiProject>
```

Listing 3.1. A small example for an OSGi model in XML.

Listing 3.1 shows the generated text for a small example I implemented. It is a simplified version of an example OSGi model from the examples repository.[1] The example consists of:

- The overarching project: `languagedemo_slim`
- The product: `englishtranslator`
- Two features: `de.scheidtbachmann.osgi.language.runtime.feature` and `de.scheidtbachmann.osgi.language.base.feature`
- Two bundles: `ch.qos.logback.classic` and `ch.qos.logback.core`

Every artifact has its *ecoreId* and name specified. The product contains the features and bundles, the features contain their respective bundles and their parent. The bundles contain the dependency to another bundle and their parent. The redundancy in the Extensible Markup Language (XML) format is everywhere. Starting from the top in line 1 from Listing 3.1, I do not care for the version of XML nor the encoding. The model is already specified by the file extension. Both *ecoreId* and name are always specified as seen in line 3 from Listing 3.1, even though they contain the same information. The *ecoreId* can be generated directly from the name and a prefix of the artifact type. The features do not have to mention their parent artifact nor do the bundles like they do in line 4.2 from Listing 3.1. The connection between two bundles can be implemented with only one bundle specifying the connection.

The current implementation mentions both sides, like in lines 6.3 and 7.3.

The improvements are shown later. Now I take a closer look at the existing visualization.

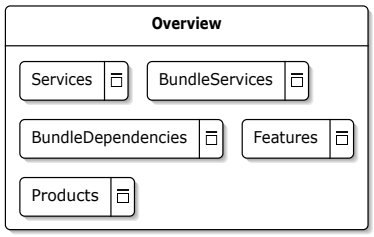


Figure 3.2. Default look of every visualization.

[1] https://github.com/kieler/osgiviz/tree/master/examples/languagedemo

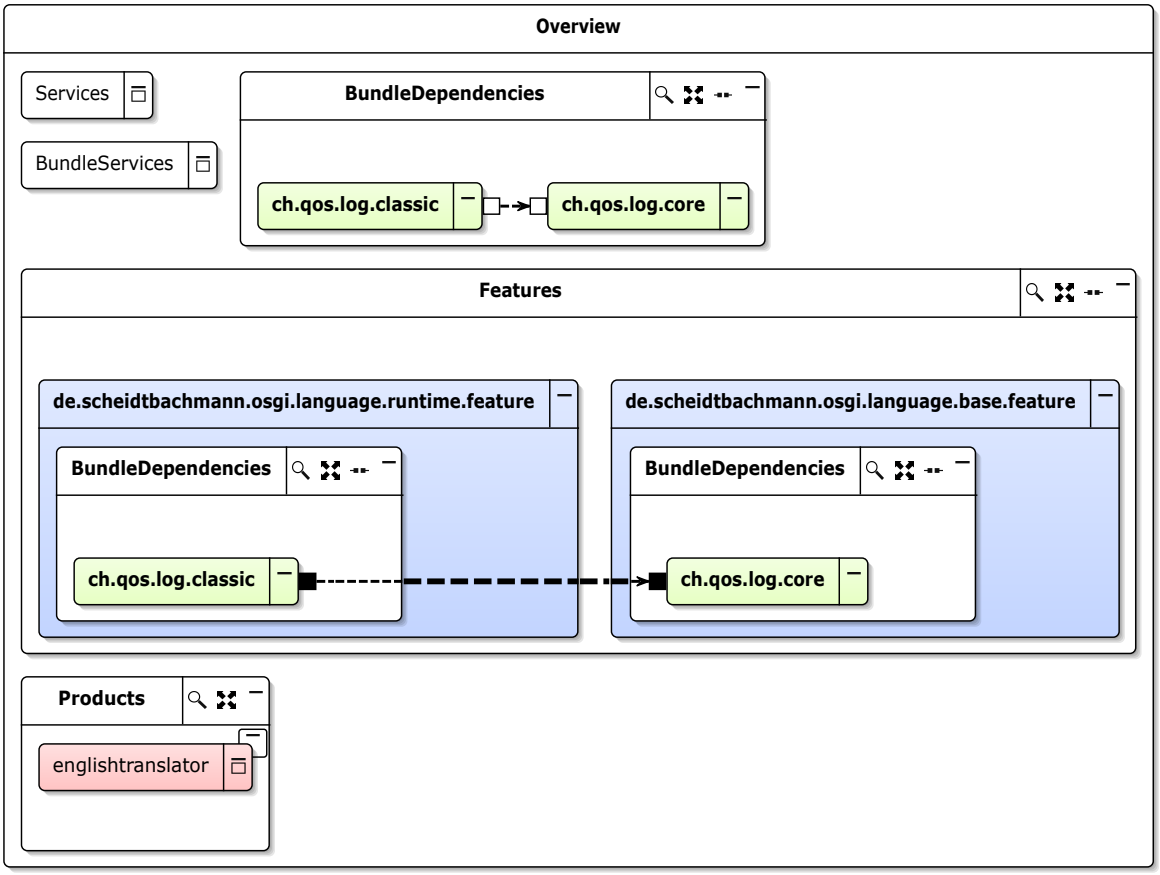


Figure 3.3. View of the XML example with available artifacts shown.

▻ *Services* shows the services described in this model.

▻ *BundleServices* shows the bundles with their services embedded in them.

▻ *BundleDependencies* shows the bundles and their dependencies.

▻ *Features* shows the features.

▻ *Products* shows the products.

Figure 3.3 is one possible visualization of Listing 3.1. As our example does not contain services, *Services* and *BundleServices* are closed. *BundleDependencies*, *Features* and *Products* are opened with dependencies enabled. Until now, I introduced the already existing infrastructure for visualizing OSGi models using OSGiViz. How do I improve this to write our own desired state as text by hand? The structure has to follow the structure from Figure 3.1 since that way, I can reuse the already implemented OSGi synthesis to synthesize the resulting object from our DSL.

```
1  OSGi languagedemo_slim_target {
2     Product englishtranslator {
3        Features: ['de.scheidtbachmann.osgi.language.runtime.feature',
4           'de.scheidtbachmann.osgi.language.base.feature']
5        Bundles: ['ch.qos.log.classic','ch.qos.log.core']
6     }
7     Feature 'de.scheidtbachmann.osgi.language.runtime.feature' {
8        Bundles: ['ch.qos.log.classic']
9     }
10    Feature 'de.scheidtbachmann.osgi.language.base.feature' {
11       Bundles: ['ch.qos.log.core']
12    }
13    Bundle 'ch.qos.log.classic' {
14    }
15    Bundle 'ch.qos.log.core' {
16    }
17    'ch.qos.log.classic' Dependency To 'ch.qos.log.core'
18 }
```

Listing 3.2. First structure for OSGiDSL implementing Listing 3.1.

First, I need a structure for our DSL. Listing 3.2 is the first attempt at a structure for the OSGiDSL. This structure removes a lot of redundant information. The XML structure is gone and replaced with human writable text. There are no parents mentioned in child artifacts, and only the name is written down since the *ecoreId* can be generated. The containment is implemented as individual lists with square brackets inside the curly brackets of an artifact. Depencies between bundles are shown by writing them as *sourceBundle* `Dependency To` *targetBundle* as seen in line 17. Further improvements can be made by adding the dependencies to the bundle containments.

```
1  OSGi languagedemo_slim_target {
2     Product englishtranslator {
3        Features: ['de.scheidtbachmann.osgi.language.runtime.feature',
4           'de.scheidtbachmann.osgi.language.base.feature']
5        Bundles: ['ch.qos.log.classic','ch.qos.log.core']
6     }
7     Feature 'de.scheidtbachmann.osgi.language.runtime.feature' {
8        Bundles: ['ch.qos.log.classic']
9     }
10    Feature 'de.scheidtbachmann.osgi.language.base.feature' {
11       Bundles: ['ch.qos.log.core']
12    }
13    Bundle 'ch.qos.log.classic' {
14       Dependency To 'ch.qos.log.core'
15    }
16    Bundle 'ch.qos.log.core' {
17    }
```

```
18  }
```

Listing 3.3. Second structure for OSGiDSL implementing Listing 3.1.

Listing 3.3 improves on the dependency redundancy from Listing 3.2. Bundle dependencies are now written as references inside the curly brackets of the source bundle. Only the bundle the dependency comes from articulates the dependency and thus removes the bidirectional references. This structure enables us to write the same model architecture with 567 characters, which is down a lot from the 1447 characters used in the XML version from Listing 3.1. There are still improvements to be implemented. For example the indentation stemming from the `OSGi` curly brackets in line 1 is plain annoying. The artifact types should also be lowercase.

Introducing the second and final version of the proposed DSL:

```
1   projectName languagedemo_slim_target
2   product englishtranslator {
3      features: ['de.scheidtbachmann.osgi.language.runtime.feature',
4            'de.scheidtbachmann.osgi.language.base.feature']
5      bundles: ['ch.qos.logback.classic',
6            'ch.qos.logback.core']
7   }
8   feature 'de.scheidtbachmann.osgi.language.runtime.feature' {
9      bundles: ['ch.qos.log.classic']
10  }
11  feature 'de.scheidtbachmann.osgi.language.base.feature' {
12     bundles: ['ch.qos.log.core']
13  }
14  bundle 'ch.qos.log.classic' {
15     dependency to 'ch.qos.log.core'
16  }
17  bundle 'ch.qos.log.core'
```

Listing 3.4. Final structure implementing the example from Listing 3.1.

The outer project curly brackets were removed and all uppercases were replaced with lowercases as types in other languages are always lowercase. The need to write curly brackets if the artifact is empty was also removed.

The visualization of our OSGiDSL looks identical to Figure 3.3. The identical look is exactly what I wanted to achieve and I have succeeded in implementing a human writable DSL that is capable of representing the same architectures as the OSGiViz generator.

The desired state contains nothing specific from OSGi, which is by design as the goal is to abstract everything into SPViz. SPViz should be able to generate everything that is needed to handwrite a desired state for any software architecture.

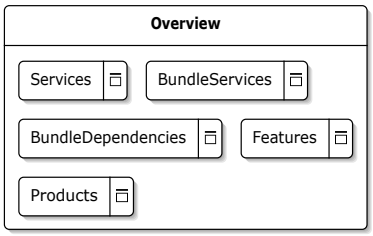


Figure 3.4. Default look of visualization

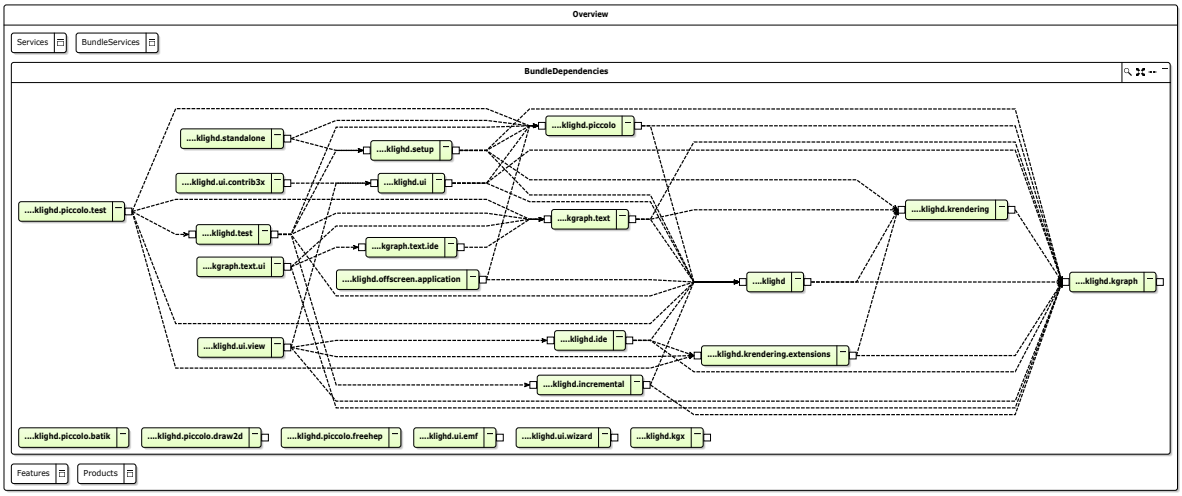


Figure 3.5. KLighD bundle dependencies visualized with OSGiViz

3.2 Differences in Visual Comparison

This section introduces different choices for displaying changes and the reasoning behind the final selection, but also the colors chosen to represent different kinds of changes.

Let us first take a look at what is already in place.

OSGiViz is already able to visualize a single model as seen in Figure 3.3. Figure 3.4 displays the result of the OSGi synthesis and the recursive subsyntheses. The visible subsynthesis are *Services*, *BundleServices*, *BundleDependencies*, *Features* and *Products*. These subsynthesis are nodes in their own regard and recursively call their subsynthesis. This creates the tree of nodes the visualizer displays at the end. The visualizer can display any OSGi model. Some examples are shown below with KLighD[2] and the language demo[3].

I want to create a structure with which I can compare two models. This would be used to compare old and new versions, different branches in source control, or a current state and a desired state of an architecture. I have to expand the capabilities of OSGiViz to display their

[2]https://github.com/kieler/KLighD
[3]https://github.com/kieler/osgiviz/tree/master/examples/languagedemo

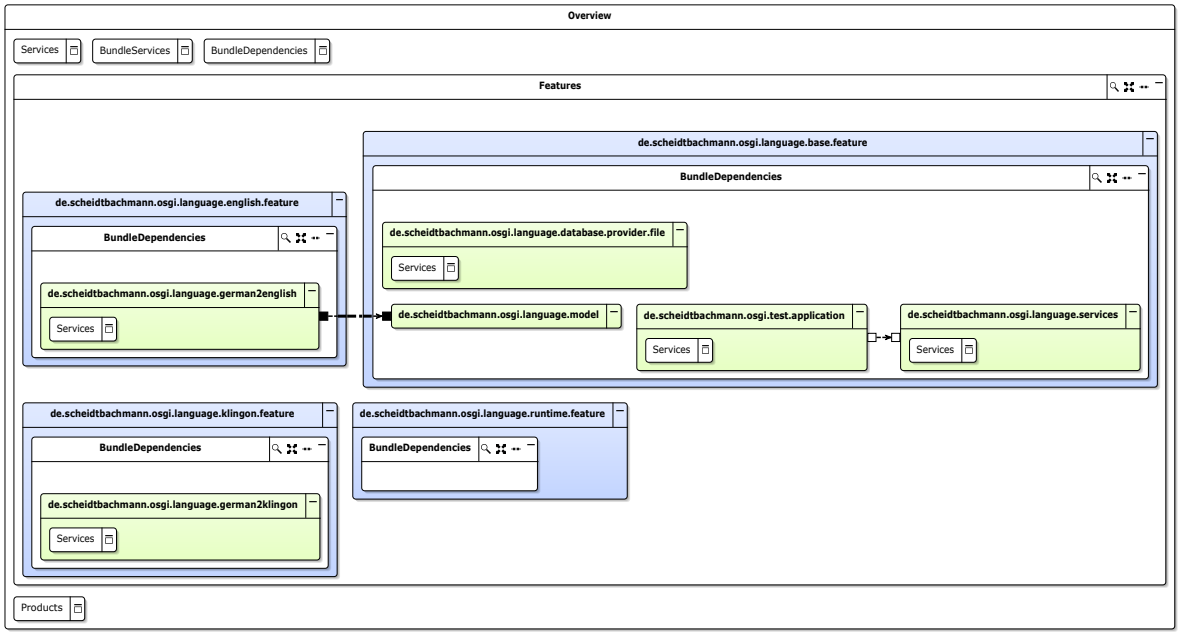
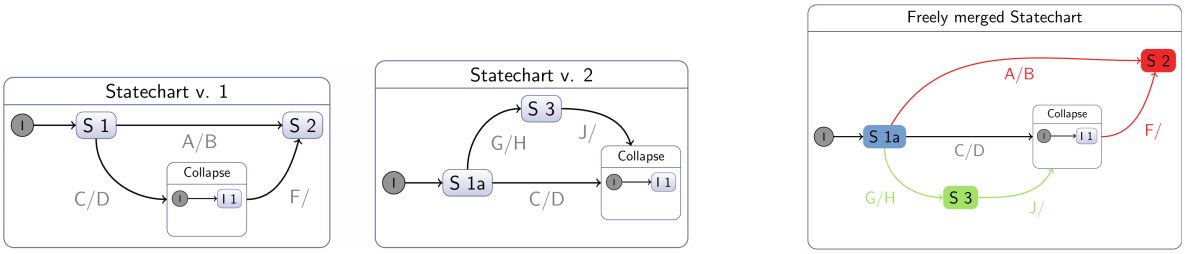


Figure 3.6. Language demo features with bundle dependencies visualized by OSGiViz



(a) Original Diagram

(b) Freely merged diff with red marking deletion, green addition and blue movement

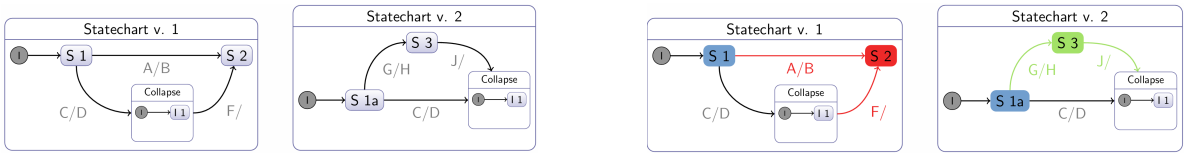
Figure 3.7. Comparison example of the freely merged diff by Arne Schipper et al. [SFH09]

differences in a single merged graph or as two models side by side to achieve this comparison. I have to choose between those two options as implementing both would not be in scope for this thesis.

3.2.1 Freely Merged Differences

The first option is a freely merged diff That way, both states are compared and merged into a single graph, which then shows the overall structure of both states with the differences highlighted in color.

This example between Figure 3.7a and 3.7b provided by Arne Schipper et al. [SFH09]



(a) Original Diagram

(b) Plain diff with red marking deletion, green addition and blue movement

Figure 3.8. Comparison example plain diff by Arne Schipper et al. [SFH09]

shows the two original diagrams, which are in our framework comparable to the current state as *Statechart v. 1* and the desired state as *Statechart v. 2*. The result is then merged into a freely merged diff as shown in Figure 3.7b. All nodes are incorporated into the final merged diff and the differences are highlighted in color.

For example, the node *S 2* is missing in the desired state and therefore is highlighted in red, while the node *S 3* is added in the desired state, is therefore colored in green. The node *S 1* in the current state and the node *S 1a* in the desired state fulfill the same purpose. The only difference is the connection to other nodes. As a result, the nodes are merged and the resulting node is colored blue. Completely unaffected nodes stay the same color as they are neither merged nor deleted or added.

The resulting graph is very compact, takes up little space, and additions or deletions are easily visible. Yet the computation of a merged graph is complicated. Simple examples, as shown above, are visualized easily and are pleasant to look at. However, more convoluted examples, especially with nested layouts, get very labyrinthine and therefore unreasonably complex and difficult to understand. Another downside is the recognizability of the resulting graph after examining the known current state. With many changes the merged diff changes the graph so much it becomes unrecognizable and it takes a long time to understand the changes made to the original architecture.

Due to these downsides, I introduce another way of displaying differences in the following.

3.2.2 Plain Differences

The plain diff is the comparison of two graphs without merging them. Two graphs are placed side by side, and the differences are highlighted with color. It does not require another synthesis for creating a singular merged graph and the plain diff is therefore a lot simpler to implement.

Figure 3.8 shows that both the current state, *Statechart v. 1*, and the desired state, *Statechart v. 2*, are just displayed again on the right side; however, the nodes that got changed are colored. The current state contains the node *S 2* while the desired state got rid of it, hence, the node *S 2* is colored red. Looking at the desired state, it contains the node *S 3* while the current state does not. Therefore, it was added to the desired state and thus is colored green. The node *S 1* is the result of the nodes *S 1* and *S 1a* merging, but the connections change from the current to the desired state, and that being the case, is colored in blue.

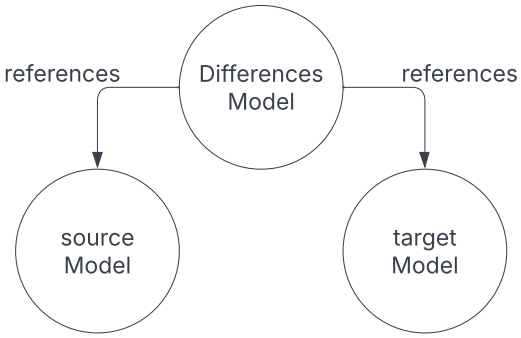


Figure 3.9. Differences architecture

This way of displaying the graphs is simpler then the freely merged diff. I am able to print the graphs as they are synthesized and just have to color the changes nodes accordingly. This eases the implementation and fits the recursive synthesis as I can show the diff synthesis as a graph and the plain diff graph is visualized. The side-by-side comparison is user-friendly, and no introduction is needed to aid understanding.

There are a few downsides to this comparison style. The more nodes are compared at a time, the more they are scaled down. This is true for all styles, nonetheless, the shrinkage is doubled here, as there are two graphs displayed at a time. This, at times, creates the need of zooming into the left or right side, which then loses the ability to compare the states, as only a part of one state is shown. This style is more sensitive to structural changes, as the structure of both states should be nearly identical to easily and quickly identify the changes.

In the end, I decided to use the plain diff. Mainly for its ease of implementation and simple readability compared to the difficult implementation of a freely merged diff and its reading difficulties for small changes.

Now that I have decided on the way I want to display the diff, the next section discusses how the plain diff is realized for OSGiViz.

3.2.3 Differences Model

The main idea is to expand on the recursion by introducing a new model that sits at the top of the tree of nodes, which then combines two models by calling their syntheses as subsyntheses. Figure 3.9 shows the connections between the model files. The OSGi diff model is specified by a DSL and serves as the connection between two OSGi models. The diff model contains a reference to the *sourceModel* and to the *targetModel*. Both states follow the same OSGi model structure and can therefore be synthesized by the same synthesis. They can also be swapped anytime by changing the referencing order in the differences model. I disregard coloring the differences for now.

The resulting visualization looks like this:

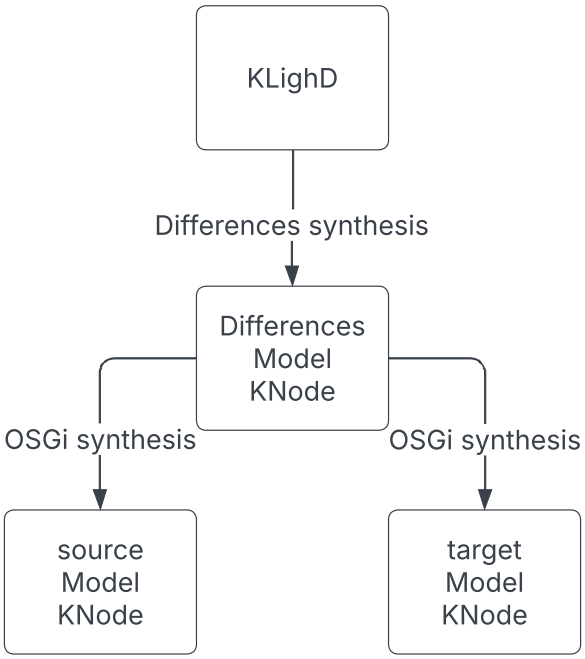


Figure 3.10. Synthesis architecture

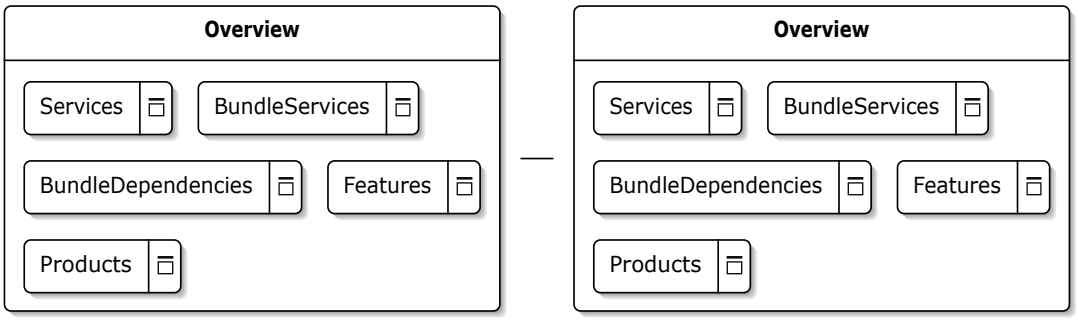


Figure 3.11. Diff model visualization default look

Figure 3.11 shows the visualization of the proposed idea from Figure 3.10. Figure 3.11 shows two models side by side, but in reality it is just one diff model that has two OSGi models as subsyntheses. The default view is identical to Figure 3.4. It just shows the two underlying models side by side as the goal is the visualization of a plain diff, which calls for a side by side visualization. Opening the features reveals that both sides have the same

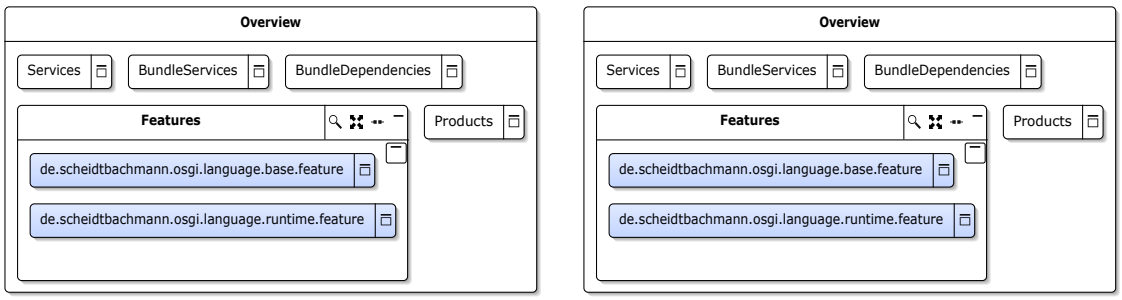


Figure 3.12. Diff model visualization with features

features. The two models work in sync if it is possible. Every action on one side is also done on the other side if the node exist on both sides. This is an important feature so that the user does not need to look for changes twice when interacting with the diff model.

Now that I have a functioning visualization capable of displaying two models at the same time, I can discuss how to visualize the differences between the models in the most fitting way.

3.2.4 Colors

Colors are crucial for finding the changes between the current and desired state easily with the human eye. The possible changes are done when writing the desired state, and they are:

1. The addition of any artifact is defined by writing it in the desired state when it is not part of the current state.

2. The deletion of any artifact is shown by not including it in the desired state.

3. The movement/merging of any artifact has several possibilities to be shown. The dependencies can change or the overarching node can change, but it itself stays the same.

Striking and recognizable colors are essential to the user experience. Firstly, colors that are already used when coloring the nodes in the visual graph or colors closely resembling them are off limits as there is no differentiating them. This creates a problem as the color for nodes is chosen not randomly but by fixing the value and saturation in the Hue, Saturation, Value, or in short HSV color space, and only circling around the hue. Circling around the hue means increasing the hue untils it is at 360° and then starting again at 0. This happens in every shortening intervalls to ensure the colors are unique Those colors are generated during the generation of the visualization tool by SPViz and are then used during the synthesis. This means I need to differentiate the generated colors in general by, for example, increasing the saturation of our colors.

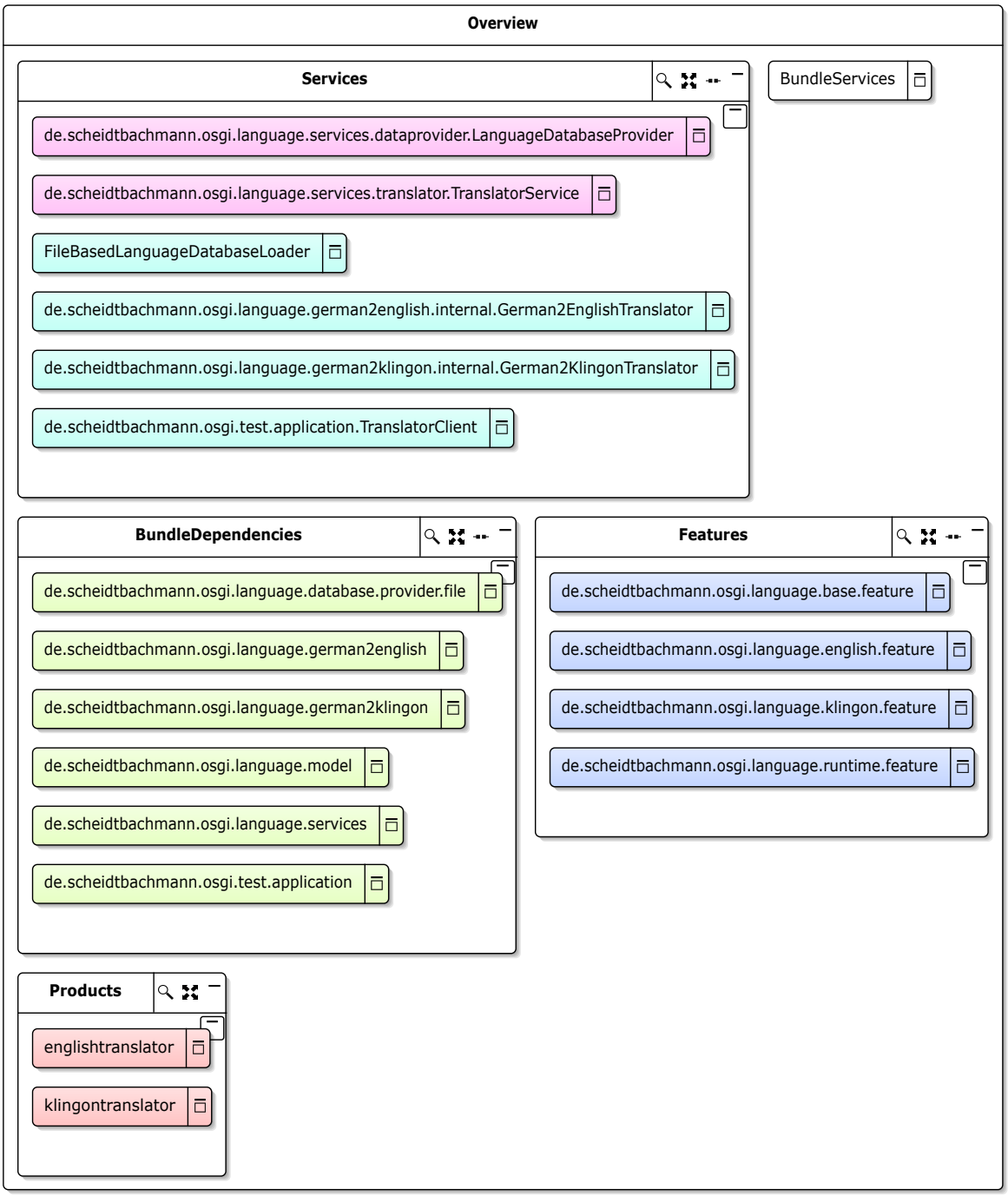


Figure 3.13. Language demo visualized with all subnodes opened

Figure 3.13 shows the first five colors to be generated for artifacts.[4]

[4]https://github.com/kieler/osgiviz/tree/master/examples/languagedemo

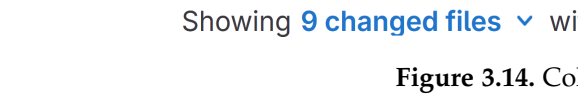


Figure 3.14. Color usage in GitLab

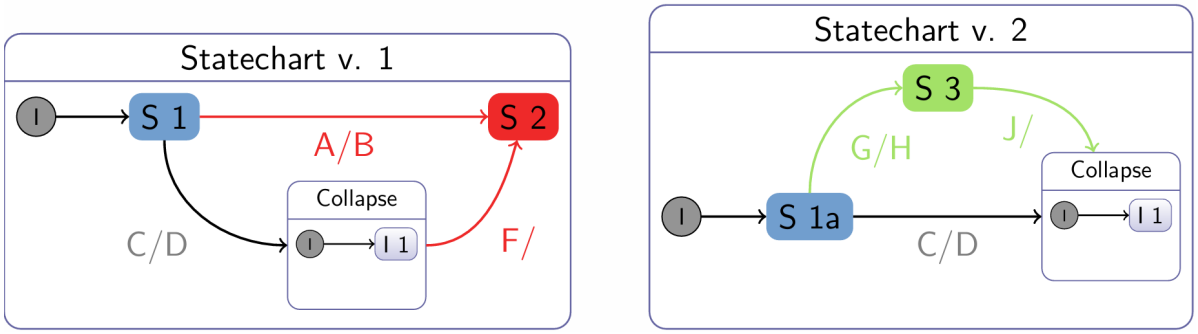


Figure 3.15. Color usage by Arne Schipper et al. [SFH09]

I also want fixed colors for fixed meanings. For the abstraction into SPViz to work, a color that has a special meaning should stay the same, regardless of visualized architecture. This renders generated colors for visualizing changes unusable as different architectures incorporate a different amount of colors and the colors used for visualizing changes would change when the architecture changes.

I at first identified what colors are applied in real-world contexts for the possible changes. The first example is the color usage of GitLab to display differences in textual files:

Figure 3.14 is a screenshot from a commit issued during this thesis and shows the header of the changes from the commit. It uses the color green for addition, red for deletion, and blue for identifying the changed files. GitHub uses this color scheme as well.

A second example is the color usage of the previously mentioned related work from Arne Schipper et al. [SFH09].

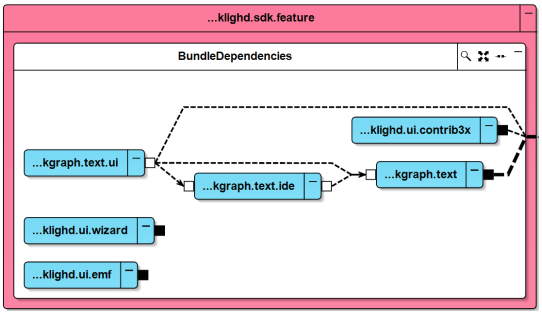
Figure 3.15 shows the deletion in red, the addition in green, and even the movement of a node in blue.

The use of red as deletion and green as addition is widespread, and there are countless comparison tools available that use red and green in this way. Most important is the usage in GitLab[5] and GitHub[6], as they are widely used in the world of software engineering and their color usage is therefore well known. The color for the movement of a node is a more difficult choice. Arne Schipper et al. [SFH09] introduced blue as the color of choice for showing movement of a node in Figure 3.15, which is why blue is our choice for showing movement.

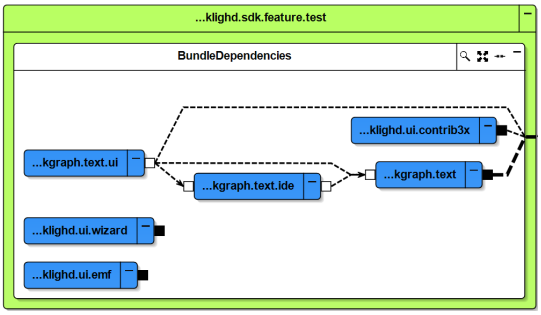
I need to ensure that my colors stick out by increasing the saturation. The HSV colors that are generated always use a value of 1, a gradient of 0.12-0.24 and the hue is circled around. My colors always use a value of 1, a saturation of 0.62 for addition (green), 0.52 for deletion (red), and 0.52/0.80 for movement (blue). The difference in this case is 0.4-0.7 in saturation,

[5] https://about.gitlab.com/company/

[6] https://github.com/about



(a) Part of a *current state* showing removal and change in parent



(b) Part of a *desired state* showing addition and change in parent

Figure 3.16. Comparison example using addition, deletion, and movement with the plain diff

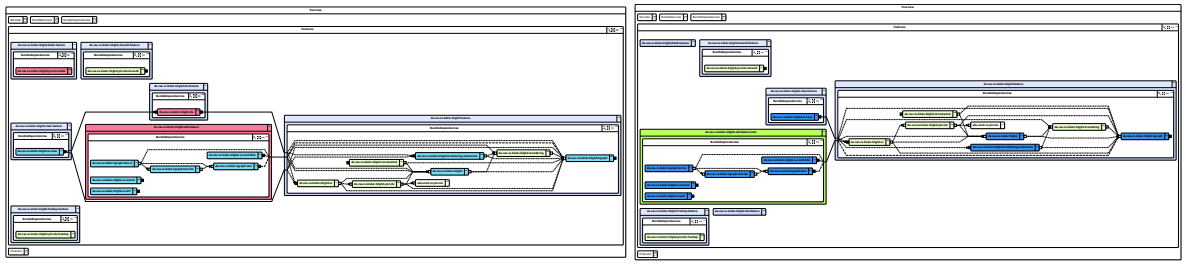


Figure 3.17. Comparison of KLighD feature architecture with some changes

which ensures that even though red, green, and blue are generated by SPViz and used by OSGiViz, the colors I chose, are different enough for any misunderstandings to happen.

Figure 3.16 shows the colors in action during a comparison. The colors are higher in saturation than the colors previously presented in Figure 3.13. I also use two different kinds of blue, to indicate, which state I are currently looking at.

Figure 3.17 displays two models. The left one, the current state, was generated by the OSGiViz generator while the model on the right, the desired state, was handwritten by us with some changes to demonstrate the usage of color. This now connects the two core concepts of this thesis. I can display two models and compare them using the plain diff and I can manually write a model that gets compared to an existing model. This enables this framework to visualize version control, source control and compare the current state of an architecture to a desired state that has yet to be implemented.

Chapter 4

Implementation

This chapter features the implementation details regarding the previously discussed concepts. The structure of this chapter resembles the previous one, and therefore it starts with the implementation of the OSGi diff model, followed by the core concepts of the last chapter, the desired state and the plain diff.

4.1 OSGi Differences Model

The OSGi diff model is implemented in Xtext and it specifies the structure of an .osgidiff file. The structure is defined as the connection of the current state as the *sourceModel* and the desired state as the *targetModel*. The model references are strings containing the file paths and those references are saved in an OSGiDiff object. These file paths are then used to load the *sourceModel* and the *targetModel* during the OSGi diff synthesis if they were not previously loaded to reduce redundant runtime. The OSGi diff synthesis then calls the synthesis for the loaded models and adds the respective other model to the synthesis properties as well as the knowledge whether the synthesis is for the *sourceModel* or the *targetModel*. This is used later to identify differences and categorize them accordingly.

4.2 Desired State

The desired state is implemented in Xtext and it specifies the contents of an .osgidsl file. The grammar was first generated from the OSGi ecore model and was then adjusted as seen in Chapter 3.1. The section also brought the generation of the *ecoreId*. The *ecoreId* is now implemented during the linking process and combines the artifact type prefix with the name of that artifact. For example, a *bundle* with the name *new.example.test* generates a bundle with the *ecoreId Bundle_new_example_test*.

The bidirectional references, also mentioned in Chapter 3.1, are implemented during the same linking process as well. A check for a bundle dependency is done and if it is true, then a reference from the target bundle is added. This is necessary, as the bidirectional references are very useful for handling the model during, for example, the synthesis. Even though EMF has bidirectional lists that update each other, Xtext is bugged and this issue is circumvented by implementing the bidirectional references during the linking process[1].

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=282486

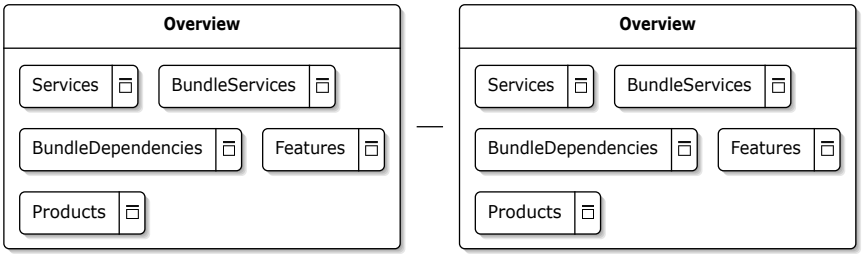


Figure 4.1. *sourceModel KNode* on the left and *targetModel KNode* on the right

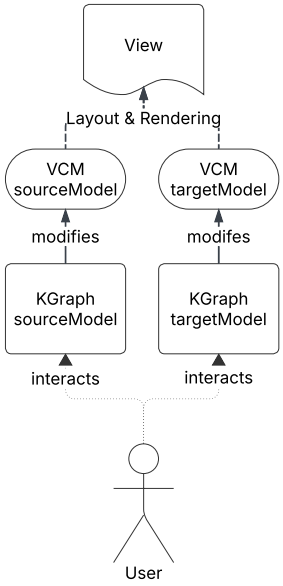


Figure 4.2. User interaction workflow

4.3 Plain Differences

The plain diff is the specific visualization of the OSGi diff model. First, an invisible Polyline connecting the *KNodes* generated from the model synthesis is added to ensure the layout of the models is consistent. This solution solves the possibility of unordered nodes, because the the order is not specified to the layout algorithm in any other way. The left graph always shows the *sourceModel* and the right graph shows the *targetModel*. Figure 3.11 shows the connection in the middle, visualized for this figure, and the *sourceModel* on the left and the *targetModel* on the right.

Differentiating the models when implementing new features was the first hurdle.

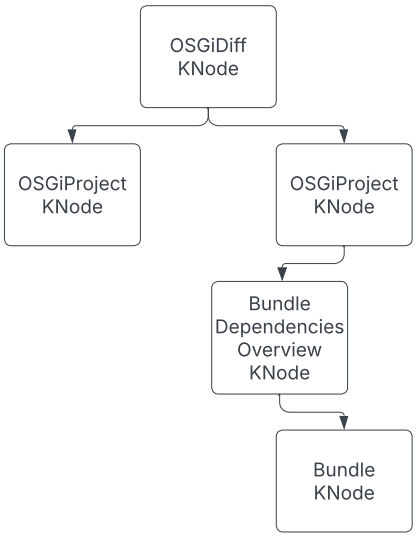


Figure 4.3. Example of partial tree of nodes

4.3.1 Visualization Context Model Handling

One of the biggest problems was the handling of the visualization context model (VCM). The VCM contains all currently visualized nodes and their status. Figure 4.2 shows the handling of the interaction as it is now implemented. Previously interacting with a node from either side of the *KGraph* resulted in an error as at first there was only one VCM, thus both sides showed the same model, and after implementing two, one for each *KGraph* side, the program still had to clarify which of the two is responsible.

The correct root VCM, in the original OSGiViz was chosen by "walking" up the tree of nodes to the root. Figure 4.3 shows part of a KGraph tree with a path from the top node to a node on the bottom which can be interacted with.

This KGraph tree was created by the KLighD synthesis that was used on the model file. The synthesis build the KGraph by creating *KNodes* for the different ways I want to visualize the software architecture and incorporates the model elements as children to the *KNodes* at the top. The structure of the OSGi model still remains as the model elements have the same structure in the KGraph tree. For example a feature still incorporates bundles. The synthesis also relates the VCM element to the corresponding *KNode*, so that the references back to the VCM and with that to the bundle from the original model can be drawn. The root VCM for all elements from one side of the tree in Figure 4.3 is the same, as they belong to the same model. Finding the root VCM is important for the program to know which of the OSGiprojects the user wants to interact with.

After finding the corresponding *OSGiProject KNode*, from which the program gets the corresponding VCM, the VCM is modified.

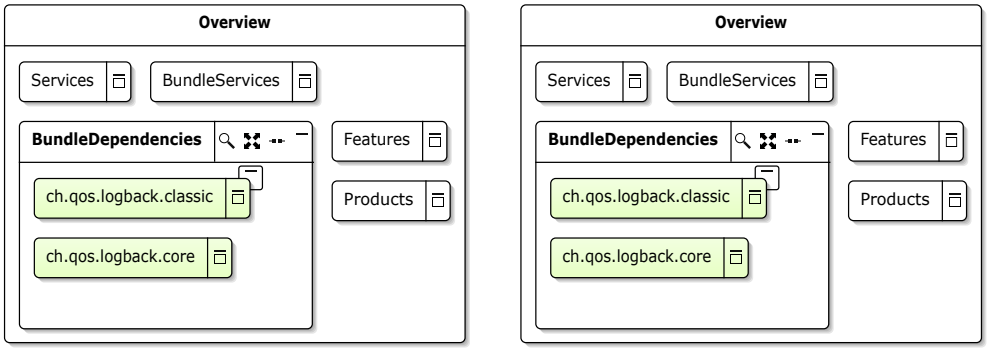


Figure 4.4. Opening one side opens both side

Actively handling the VCM also enables us to implement new features. When interacting with a node in either state, the corresponding node in the other state should act as well, if it exists. Otherwise, only the clicked-on node reacts. This works by searching for the respective VCM on the other side and modifying it in the same way. The corresponding VCM is searched for by going up the tree of nodes again and adding each step to a list, reversing it and checking all resulting options for the original *ecoreId*.

This is without a doubt a huge improvement for clarity and ease of comparison. A few examples are:

- ▻ Expanding a node like in Figure 4.4, opens the node on both sides.
- ▻ Clicking on a node highlights the node and its connections in blue.
- ▻ Showing the connections of a node.
- ▻ Hiding a node.

4.3.2 Color

Implementing the changing colors for the nodes is the next topic. I introduced three different colors: red for deletion, green for addition, and blue for movement. Of course, if it is not any of the three, the node should keep their default color.

The other OSGi model is passed down the syntheses, because I need it to compare the artifacts to. As mentioned in Chapter 3, a node that exists in the *sourceModel*, but not in the *targetModel* should be colored red and green if its the other way around. The implementation checks for each artifact if it exists in the other model.

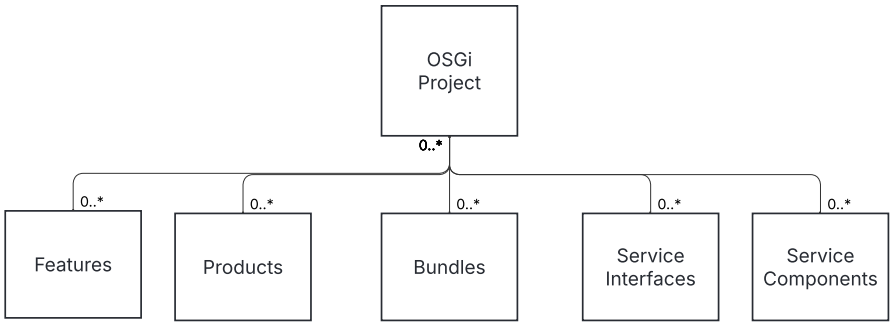


Figure 4.5. Subset of the OSGi class structure

This works because the model has direct access to the artifacts as shown in Figure 4.5. For example, a bundle with the *ecoreId Bundle_new_example_test* checks the existence of a bundle with the same *ecoreId* by accessing the *Bundles* class from the OSGi model and comparing the *ecoreId* to each bundle. If the bundle exists it is returned for later use. This works for both red and green by checking which model the node belongs to afterwards. The check for which model a node belongs to is analog to Figure 4.3.

Blue is implemented after the checks for green/red are positive. This guarantees that both models contain this artifact. Blue had two meanings, as mentioned in Chapter 3. The first implementation works by checking the parent of an artifact like a bundle.

A bundle can return its parent and is therefore able to compare its parent to an existing parent of the corresponding node in the other model. If the parents are not the same the color is set to blue. The second implementation works by accessing the respective target bundle and comparing the incoming and outgoing dependencies. If they are not the same the node is colored blue. The lighter blue and the darker blue are decided on by again checking, which model this node belongs to.

Chapter 5

Evaluation

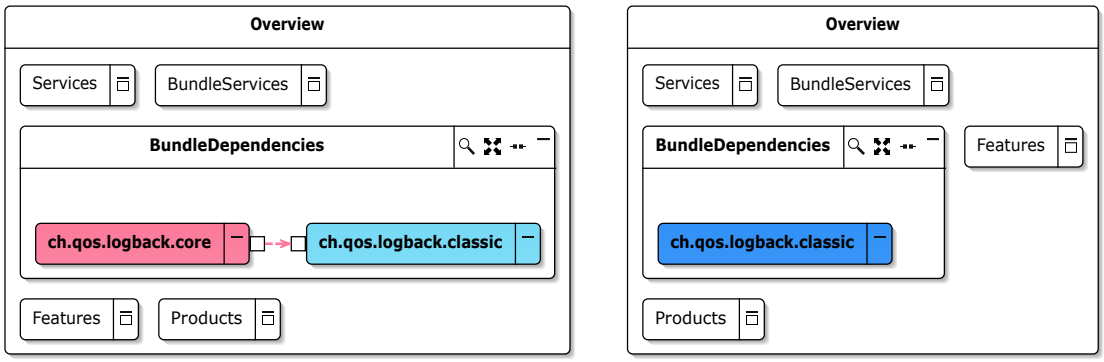


Figure 5.1. A small example diff derived from the language demo example

This chapter discusses possible questions that could be asked to evaluate the effectiveness of this comparison framework and evaluates the completion of the goals I set for myself at the start of the semester. Furthermore, this chapter describes the lessons I learned during the duration of this thesis.

5.1 Possible Evaluation Questions

Conducting a survey is out of scope for this thesis, but I still propose and discuss possible questions for future work after this thesis. The goal of the questions should be to figure out how effective the comparison framework is at letting users spot differences and ease of use.

I propose 3 different kinds of questions. The first kind of question would be questions about the meaning of the colors. A questionnaire would show how small examples diffs. with isolated changes at first and increase the complexity gradually. A low complexity example would be Figure 5.1. There is only one node removed and everything is readable without zooming. Figure 5.2 displays a very complex example which would be shown near the end of the questionnaire.

The second kind of question would be about the visibility of the changes. The amount of changes would increase and the questionnairewould for example ask on a scale of 1 to 10 how easy, how many, the changes were spotted in what time frame and did they find all changes.

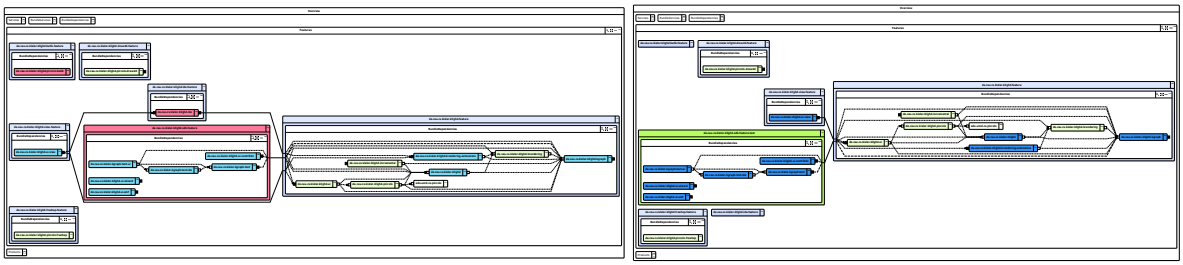


Figure 5.2. A very complex example derived from KLighD

The third kind of question would be implementing a model from a diff. The *sourceModel* would be given and the final diff and the objective would be to change the *sourceModel* into the needed *targetModel* to produce the given diff. The complexity would have to be shallow and this question could only be asked to people already familiar with programming, but this framework would also only be used by programmers. This would work with Figure 5.1, but not with Figure 5.2.

5.2 Completion of Proposed Goals

At the start of the semester, after getting the task for this thesis, I set myself goals to achieve during the implementation. Hard requirements are required for this thesis and have to be met at the end. Soft requirements are not required, nonetheless add quality of life features to round off this implementation.

5.2.1 Hard Goals

The overarching goal was to create a comparison framework for OSGiViz with the possibility of abstraction into SPViz. The comparison for OSGiViz is implemented, and the abstraction is possible, yet not implemented.

The first hard goal for this thesis was to create a way of describing an OSGi model by writing it as text and then having it displayed in the same way the generated OSGi model is. Chapter 3 has most of a section designated to the evolution of my DSL.

Furthermore, I had to create or choose a visualization style. The first idea I had was to sketch different possibilities and compare those possibilities. A few weeks in, I was introduced to the work of Arne Schipper et al. [SFH09] which then served as the basis for the comparison styles mentioned in this thesis and where I ultimately picked the plain diff style from.

The last requirement was the implementation of the chosen comparison style in OSGiViz. This encapsulates two objectives: Firstly, a way to visualize two models next to each other and secondly, to color the differences. The first objective was done fairly quickly by implementing a small-scale DSL which connects the two models into one and displays them. The second objective was more difficult to implement but was also done easily, after discovering where

the necessary changes belong.

Every hard goal has therefore been achieved, and the assessment of the obstacles at the start of the semester, was correct.

5.2.2 Soft Goals

The first soft goal was interactivity between the two models, which has been partly achieved by the synchronization of actions between the models. Every action that was implemented in OSGiViz is synchronized but the options sidebar has not yet been implemented which is an addition of significant use.

Another proposed soft goal was the implementation of other comparison styles. This has not happened, but the concept for the freely merged diff exists and is detailed clearly in its own subsection in Chapter 3.

As the title of this thesis suggests, the abstraction into SPViz was always something that will happen, but not as part of this thesis. Suggested as a soft goal at the start of this semester, I discovered quickly that this was outside of scope for me. Nonetheless, attention was spent on the implementation as to not use OSGi-specific code that would prevent the abstraction.

5.3 Lessons Learned

This was the first time that I worked on a larger software project by myself and there is a lot I learned during the implementation of this thesis.

The biggest hurdle in the beginning was the size of the project I had to change and adjust. The sheer amount of concepts and technologies was staggering. KLighD, ELK, EMF and, by extension, Xtext, were already in use by OSGiViz and were thereby required for me to understand before I could start changing the code according to my needs. There were problems that were so far out of scope for me that I had no choice but to rely on my supervisor. An example would be to get OSGiViz running in Eclipse on my machine. This seemingly rudimentary task turned out to be a multi-day problem, which I had no chance of solving in a reasonable time. Problems like these in turn made this thesis more difficult but also more interesting, and I gained deeper knowledge about subjects I had little investment in earlier.

The programming language used for OSGiViz, XTend, was a new language for me. Yet as a Java dialect and my prior knowledge of Java, it turned out to be a pleasant change and made me appreciate dynamic typing.

My programming skills in general improved greatly during this thesis, not only in terms of writing code, but also debugging, thinking about, and understanding concepts, and inventing new ones. The debugging feature was of significant help when beginning this thesis and trying to understand the software project in front of me.

The last thing I want to talk about is time management. Setting fixed time windows for the implementation was highly difficult for me at the start, as I had little knowledge about the

project and the way it works. Things take time and often longer than anticipated, especially when writing this thesis in English, with German being my native language. Though, writing this thesis nurtured my interest in the academic process and gave me a greater understanding of working in an academic setting.

5.4 Collaborations

The results from this thesis were presented to representatives from Scheidt & Bachmann GmbH[1] and they are pleased with the final look of this comparison framework. SPViz is currently used by them and they are looking forward to work with this comparison visualization.

Scheidt & Bachmann GmbH is not the only one using SPViz and the results were presented to some other users as well and the overarching response was very positive.

[1] https://www.scheidt-bachmann.de/de/

Chapter 6

Conclusion

This thesis shows a way to create a comparison framework for OSGiViz and the possibility of abstraction into SPViz. It introduces new concepts for writing and comparing models and discusses them assidouosly. The possibility of abstraction into SPViz means, that this framework can also be used for any given software structure, which increases the value of the results from this thesis tremendously.

Comparison frameworks however, are never finished, and neither is this one. There is still room for improvement in regards to the mentioned soft goals and even more possibilities beyond those soft goals.

The results can ultimately be used in real-world applications, similar to how SPViz is currently being used. They can also be used by other students to play around with and expand upon during new theses.

6.1 Future Work

This comparison framework can be expanded upon in a number of ways. Some ideas are presented here:

Abstraction into SPViz: The abstraction into SPViz is the goal for the future. Abstracting expands the usability of this framework like no other contribution will and is very important for the future of this framework.

Writing the model as an ADL: An architecture description language (ADL) is a way of describing software architectures similiar to the DSL I implemented [Cle96]. They are commonly used to generate code from an abstract description [RAB+04]. The structure of the architecture is given by a concrete implementation of the DSL and the artifacts would be predefined per type. This would make it possible to generate code from the model that you write down with the OSGiDSL. Thereby speeding up the process of converting the theoretical software structure into the finished product. The generation would be a software skeleton without concrete implementation details.

Different diffs.: Different kinds of diffs like the freely merged diff mentioned in Chapter 3 would widen the possibilities of expressing the changes made to a structure. This feature is not necessary for this framework to function but would provide some diversity.

More Interactivity: Adding the support of the option sidebar would greatly improve the usability and grant this framework the same functionality as a singular displayed model.

Different Color Schemes: Right now the dominant colors in use are red, green and blue. Yet the most common color vision deficiency is red-green [Sim16]. This makes some users unable to differentiate changes which in turn makes this framework unusable for those developers. An option for switching between different color schemes can be implemented. Some examples schemes would include brown-purple or pink-blue.

Bibliography

[Bud04] Frank Budinsky. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[Cle96] P.C. Clements. "A survey of architecture description languages". In: *Proceedings of the 8th International Workshop on Software Specification and Design*. 1996, pp. 16–25. DOI: 10.1109/IWSSD.1996.501143.

[EV06] Sven Efftinge and Markus Völter. "oAW xText: a framework for textual DSLs". In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 118. 2006.

[MNS01] G.C. Murphy, D. Notkin, and K.J. Sullivan. "Software reflexion models: bridging the gap between design and implementation". In: *IEEE Transactions on Software Engineering* 27.4 (2001), pp. 364–380. DOI: 10.1109/32.917525.

[RAB+04] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. "Archc: a systemc-based architecture description language". In: *16th Symposium on Computer Architecture and High Performance Computing*. 2004, pp. 66–73. DOI: 10.1109/SBAC-PAD.2004.8.

[RDH20] Niklas Rentz, Christian Dams, and Reinhard von Hanxleden. "Interactive Visualization for OSGi-based Projects". In: *2020 Working Conference on Software Visualization (VISSOFT)*. 2020, pp. 84–88. DOI: 10.1109/VISSOFT51673.2020.00013.

[RH25] Niklas Rentz and Reinhard von Hanxleden. "SPViz: a DSL-driven approach for software project visualization tooling". In: *Proceedings of the 20th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - IVAPP*. INSTICC. SciTePress, 2025, pp. 967–974. ISBN: 978-989-758-728-3. DOI: 10.5220/0013356800003912.

[SFH09] Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. "Visual comparison of graphical models". In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*. 2009, pp. 335–340. DOI: 10.1109/ICECCS.2009.15.

[Sim16] Matthew P Simunovic. "Acquired color vision deficiency". In: *Survey of ophthalmology* 61.2 (2016), pp. 132–155.

[TV08] Andre L.C. Tavares and Marco Tulio Valente. "A gentle introduction to OSGi". In: *SIGSOFT Softw. Eng. Notes* 33.5 (Aug. 2008). ISSN: 0163-5948. DOI: 10.1145/1402521.1402526.