Moving Transient Views from Eclipse to Web Technologies

Niklas Rentz

Master's thesis
November 15, 2018

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

Advised by
Dipl.-Inf. Christoph Daniel Schulze

Selbstständigkeitserklärung

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

Kiel,

Abstract

Understanding models with a textual and visual representation is easier and more efficient than with one of these representations alone. However, modeling diagrams by hand takes a long time and is a tedious task. Visualizing models from text automatically is a way to generate expressive visualizations quickly.

The KIELER project from Kiel University contains the KLighD framework to generate diagrams for specific languages inside an Eclipse IDE. The current trend for many applications is to move to web technologies to provide cloud services which have previously only been available on the desktop. Code development and modeling in IDEs can follow that trend to make both more accessible and faster to set up.

This thesis takes the idea of the KLighD framework out of KIELER and transfers it with new concepts into the web-based IDE Theia. It also presents new concepts needed when developing in a client-server environment and evaluates them in comparison to the approach by KIELER.

Acknowledgements

I would first like to thank Prof. Dr. Reinhard von Hanxleden as the head of the working group. Thanks to him I was able to step into the big new topic for this thesis as a way to complete my study at this university. He also provided me with good company, an office to work in, and always a helpful word during the last months.

I would also like to thank Christoph Daniel Schulze for supervising my thesis. He led me on the right track on how to work on this thesis and always had a good idea, when I was struggling with the next step. Furthermore, I thank him for helping with my writing and presentation style that improve the final quality of my work so much.

I would also like to thank Alexander Schulz-Rosengarten and Sören Domrös for being room colleagues who always helped when an immediate question came to my mind.

I also thank all other members of the staff and all students of the working group for their company and additional ideas and thoughts.

Finally, I would like to thank Moritz Eysholdt, Christian Schneider, and Miro Spönemann from TypeFox for their introduction to the technologies and their technical support around Theia and sprotty.

Brief Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Related Work . . . 3
- 1.3 Outline . . . 4

2 Used Technologies **7**
- 2.1 The Language Server Protocol . . . 7
- 2.2 Browser Technologies . . . 9
- 2.3 Server Technologies . . . 16

3 Concepts **21**
- 3.1 Overall Architecture . . . 21
- 3.2 Choice of Frameworks . . . 24
- 3.3 The View Model . . . 29
- 3.4 Communication . . . 34
- 3.5 Diagram Options View . . . 39
- 3.6 Rendering on the Client . . . 41

4 Implementation **43**
- 4.1 Server Implementation . . . 43
- 4.2 Client Implementation . . . 56

5 Evaluation **67**
- 5.1 Comparison Between the Web-Based Approach and KIELER . . . 67
- 5.2 Developing with Theia and Sprotty . . . 72
- 5.3 Completion of Proposed Goals . . . 73
- 5.4 Lessons Learned . . . 75

6 Conclusion **77**
- 6.1 Future Work . . . 77

Bibliography **83**

Lists **87**

Acronyms

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
DSL	Domain-Specific Language
ELK	Eclipse Layout Kernel
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
HTML	Hypertext Markup Language
ID	Identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JSON-RPC	JSON Remote Procedure Call
JSX	JavaScript XML
JVM	Java Virtual Machine
KEITH	Kiel Environment Integrated in Theia
KLighD	KIELER Lightweight Diagrams
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
LoC	Lines of Code
LSP	Language Server Protocol
MVC	Model View Controller
NETCONF	Network Configuration Protocol
npm	Node.js Package Manager
oAW	openArchitectureWare

Acronyms

RFC	Request for Comments
SCChart	Sequentially Constructive Statechart
SCCharts	Sequentially Constructive Statecharts
SCG	Sequentially Constructive Graph
SCL	Sequentially Constructive Language
SVG	Scalable Vector Graphics
SWT	Standard Widget Toolkit
UI	User Interface
URI	Uniform Resource Identifier
VSCode	Visual Studio Code
XML	Extensible Markup Language
YANG	Yet Another Next Generation

Chapter 1

Introduction

In the world of big software systems that are updated and improved more and more the simplicity gets lost and new technological advances do not fit in the systems anymore. Sometimes new features have to be implemented in overly complicated ways and take more time to develop than necessary. Staying with old technologies limits flexibility, but provides features that have been tested and verified to work correctly often and by many people. Starting to implement a very new software system and framework in contrast opens the possibility to utilize and test new technologies for their feasibility in usage and maintenance. Not every newly hyped technology will last very long, though, and so the choice for which ones to use can affect the rise or the fall of new developments.

For the Real-Time and Embedded Systems Group at Kiel University's Department of Computer Science, an urge to try to reimplement the core structure of their Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) project was present to benefit from some of the advantages that may come with new technologies and the new field that opens up for research topics. Since KIELER is the only implementation of the language Sequentially Constructive Statecharts (SCCharts), which is the main full programming language developed and used to research new concepts in synchronous languages by the working group [HDM+13], another implementation based on different frameworks with a different goal in mind is a good way to expand research and use cases of the synchronous language constructs around SCCharts. So migrating the main features of KIELER into a new Integrated Development Environment (IDE) that uses different new frameworks and concepts is a good way to improve working with parts of the KIELER project and to re-think design and implementation details of it.

An IDE based on new concepts is *Theia*, which is an IDE based on web technologies. That means it is not just a program to be downloaded, installed, and executed locally on one's computer, but instead is also designed to be hosted on a server. You can connect to that server with your browser and have an easy setup and a familiar user experience. This idea is already implemented in many applications and continues to grow in popularity. A publicly available and well-known implementation of this idea is Google Sheets[1] with a patent filed in 2004 by Rochelle et al. [RLK+15], which is a step from classic spreadsheet software such as Microsoft Excel to web technologies. However, also other applications, such as an environment to showcase and execute small code snippets for web development called JSFiddle,[2] have been growing on the web, with many of them used by a large number of users and even big

[1] https://www.google.com/sheets/about/

[2] https://jsfiddle.net/

companies to share ideas effortlessly. JSFiddle already implements some concepts of IDEs in web environments, but is not designed to be a framework for real development. Theia is an implementation of the idea to bring the known concept of IDEs to web technologies and is an option to elevate working with SCCharts and other languages from the KIELER project to web-based technologies.

Furthermore, supporting the Language Server Protocol (LSP) by Microsoft as a new web-based technology is a good way to start programming an IDE. When using language features as a service accepted by a well-defined API it allows easier maintenance for supporting many languages. The separation of tasks into a client and a server with the client running in a browser leads to easier platform and operating system independence, because all most commonly used browsers run on all main operating systems. Discussing why we chose Theia to be the new IDE is discussed later in the thesis.

The new IDE is not meant to replace KIELER, but should rather be seen as a project developed side by side with it. KIELER should still be used for developing with SCCharts, since all features of KIELER needed for expert development cannot be implemented from scratch in the new IDE during one semester worth of time. The new IDE can be used for teaching, where students can easily connect to the server and test the language without overloaded menus with many features they cannot even understand yet. Also, it can be seen as a new platform that can be used to work and research on for future work. New concepts for the used languages, diagrams, and automatic layout can be researched, for which just developing in Eclipse, which is based on a relatively old core technology, may only allow for a niche view on what can be reached with IDEs. Many new frameworks and concepts only working in a web environment can also be integrated and improved, which may cause advances in usability and diversity in use cases. So this may provide the working group with new topics and technologies for research. A new implementation also allows for internal design and implementation decisions to be reconsidered and changed, if these decisions turned out to be impractical in KIELER.

The project of migrating the KIELER IDE to a new platform is split up into two parts. One part is handling the language support features such as syntax highlighting, code completion, compilation, etc. written and implemented in the master's thesis of Domrös [Dom18]. The other part is this thesis, which is about adding transient views and diagram features to the new web-based environment. "Transient" in this case means that views are not kept and constantly updated, but are reconstructed each time a change in the model has happened.

1.1 Problem Statement

The problem statement can be concise: all features of KIELER concerning the rendering and modification of diagrams should be moved to web technologies. But this description is too brief to understand what needs to be done. Let me elaborate. KIELER has a diagram framework called KIELER Lightweight Diagrams (KLighD) that handles the steps needed to produce visualizations for arbitrary input models. There are also different interactions that

KLighD allows. A user can interact with the drawn diagram by invoking actions that cause the diagram to update. The user can also change options that allow them to change the diagram's appearance. These features for KLighD were first proposed and implemented by Schneider et al. [SSH13] and have to be adapted or reimplemented to work in the new environment.

Because we want a web-based IDE with code running on a backend server (doing things such as communicating with the data system and object models) and code running on the frontend application (doing things such as rendering a final diagram and interacting with it), the execution cycle of KLighD cannot be adopted as is. It has to be split up in a clean way that defines which parts are executed on the server or the client and how the communication between them should work. As programs running in web browsers are often written in JavaScript, the JavaScript Object Notation (JSON) is a natural choice to carry the messages between the client and the server.

SCCharts and many languages visualized with KLighD are written in a grammar language that is able to generate infrastructure using the web-based approach, so it is just natural to use this existing technology as an advantage.

To draw the diagrams in a web environment, we also need to settle for a way how they should be drawn. The rendering framework from KIELER cannot be reused in a web setting, because it is fixed to the Eclipse environment and is not reusable in other client software. Another framework that can be used in a web environment and is compatible to the communication of the IDE needs to be found and configured to imitate the previous behavior.

1.2 Related Work

A project very similar to what is presented in this thesis is the implementation of an IDE for the modeling language Yet Another Next Generation (YANG) in Theia called Yangster.[3] YANG is a data modeling language used for communication over the Network Configuration Protocol (NETCONF). It is similar to XML, but has its own grammar and defines some default structures, which describe how data is supposed to look. It can be structured in lists, modules, choice constructs, and many more and referenced and extended in other models. The RFC 6020 [Bjo10] describes everything that is possible to be modeled in the language.

TypeFox created Yangster to show a proof of concept, how the Theia IDE can be extended to create an environment that uses an Xtext Domain-Specific Language (DSL). The SCCharts DSL is also created with the help of Xtext, so the concepts of that idea can also be reused for this thesis. Yangster also uses the LSP and extends it with diagrams with the help of the diagramming framework sprotty [Köh17a]. They reach automatic translation from the textual representation of YANG models to a graphical representation rendered in any browser. Because that concept is very similar to what this thesis aims to implement, the implementation of Yangster can be used as a working basis that uses most concepts of Theia, sprotty, and the LSP. Yangster, however, only supports diagrams for this one specific language and implements the generic features and interaction defined by sprotty. This thesis shows how an entirely different

[3] https://github.com/theia-ide/yangster

view model of the KLighD framework that is able to model a wider variety of languages fits into sprotty and adds other interaction to the system.

Another example of a proof of concept for the technologies of sprotty, the Eclipse Layout Kernel (ELK) for automatic diagram layout, the LSP, and Xtext is the ELK live demo page created by Miro Spönemann.[4] It was created mainly to show the automatic layout for diagrams by ELK, but also conveniently uses web technologies that can be used for this thesis. However, it is not an IDE for any development and contains only basic visualizations of node-edge graphs for the ELK graph structure. The view model and renderings of KLighD handled by this thesis are far more complex and the full IDE also adds more interactivity than this demo page.

Schneider et al. [SSH12] describe how the concept of model-driven visualization is used to implement a transient view approach in Eclipse. They show the concept of fully automatic creation of a graph view model that is used to model a visual representation and can be rendered to a viewing framework used in Eclipse. Furthermore, they describe how automatic layout can be used in this process to reach visually appealing arrangements of the graph model. The implementation of the paper is currently used for diagram generation in the KIELER project. It can be seen like a template for this thesis for integrating a view generation to an IDE. However, this thesis uses a different IDE that is furthermore based on web technologies and therefore needs different concepts and a new implementation.

The master's thesis by Domrös [Dom18], written parallel to this thesis, is about moving model-driven development from Eclipse to web technologies and synergizes with this thesis. Both implementations together form the basis of the new web-based IDE Kiel Environment Integrated in Theia (KEITH) that is similar to KIELER with new concepts. Domrös' part in this project is comprised of migrating the KIELER compiler features to the new IDE and changing the UI concepts that come from Eclipse and cannot be reused in the new environment. He also researched how the new product KEITH is built to form a local desktop application with Electron or as separate server and client software that can be used in a wide variety of new use cases. His work is isolated from my work though, as I move the transient view approach to generate automatic drawn diagrams from DSL models to the same framework in Theia, which can be merged into the work of Domrös. That way, both systems together form a bigger system with additional functionality that more closely matches the features in KIELER and even exceeds it in some aspects, such as the possibility to be executed in a browser.

1.3 Outline

Most of the main frameworks and technologies used for the implementation of this thesis are explained in Chapter 2. The information of that chapter is needed to better understand the chapters following that. Chapter 3 then discusses the choice of the frameworks and how they are used to generate interactive diagrams automatically from text source models in a web-based environment. Furthermore, it explains design choices made for the additional view showing options to modify the diagram generation. This concept's implementation

[4] https://rtsys.informatik.uni-kiel.de/elklive/

and details on faced problems are then presented in Chapter 4. Chapter 5 evaluates the performance and usability of the new concept and compares it to the implementation of KIELER. Finally, Chapter 6 concludes this thesis and gives an outlook on future work.

Chapter 2

Used Technologies

This chapter introduces the most important technologies that are used to simplify the implementation of this work. The next sections handle one framework or project each and explain what they are capable of and how their features can be used for my implementation. It is split up differentiating between technologies used on the client side with web technologies for implementing the IDE and on the side of the language server following the LSP, as explained in the next section.

2.1 The Language Server Protocol

The Language Server Protocol (LSP) is a core concept used for this implementation. It was initially designed for the IDE Visual Studio Code (VSCode) [Köh17c], but also got quickly adopted by many other tools, such as Eclipse, Vim and Sublime.[1] The LSP is an extensible protocol by Microsoft that is used to extract specific language support implementations from IDEs and development tools to an own program that can be used by any tool supporting this protocol. Without using the protocol, each IDE or development tool has to implement support for every language individually, which is a very time-consuming task and therefore every IDE has a limited set of supported languages. Table 2.1a presents an extract of a support matrix showing which IDE has support for which language.

If we assume to have n languages that need support by m tools, then there have to be $n \cdot m$ independent implementations to have each language supported by every tool. But when utilizing the LSP, only $n + m$ implementations have to be programmed. Table 2.1b shows the same languages as Table 2.1a and their availability of a language server, programmed or auto-generated by a tool such as Xtext (see Section 2.3.2), and Table 2.1c shows the same IDEs as Table 2.1a and their support for the LSP.[2] With language servers, every language only has to provide a language server and every IDE only has to support the LSP and every IDE has support for every language.

Of course this is not true for every feature that specific languages may have. The LSP provides a well-defined feature set that resembles what most programming languages, domain specific languages, markup languages, and others have in common. As the specification by Microsoft [Mic18b] states, features such as code diagnostics, auto-completion, find implementation or references, presenting documentation on hover, and many more are already

[1] https://langserver.org/

[2] A full list of officially supported IDEs and available language servers is hosted on https://langserver.org/.

Table 2.1. Language support without and with use of the LSP.

	Java	SCCharts	C	...
Eclipse	✓	✓	✓	
VSCode	✓	✗	✓	
NetBeans	✓	✗	✓	
⋮				

(a) Support matrix for IDEs and their languages.

	LSP Server
Java	✓
SCCharts	(✓)
C	✓
⋮	

(b) Extract of available language servers.

	LSP Client
Eclipse	✓
VSCode	✓
NetBeans	✗
⋮	

(c) Extract of IDE clients with LSP support.

supported. The protocol is not yet complete, as new features keep being added. During the year 2018 up until November, eight new minor versions have been published with most of them adding support for new features.

However, the protocol cannot handle specific requests needed for example for SCCharts, where a diagram view or the compilation to another language will be requested. The LSP needs to be extended by such specific requests. That extension has to be implemented by other frameworks like sprotty (see Section 2.2.4), the implementation of this thesis, or the implementation of the accompanying thesis by Domrös [Dom18].

An example communication of a language server with a development tool as presented by Microsoft is shown in Figure 2.1. After the tool and the server are connected and have exchanged their capabilities, the interaction depicted in the diagram starts. The user opens a document in the tool and the tool sends a notification containing that document, so the server knows about the current state of the document in the editor. After that the document gets changed, after which another notification gets sent with the identifier of the document (which the server has remembered) and the changes made to that document. The server now checks if the changes produce any warnings or errors, such as syntax errors or unused variables, and notifies the tool, so it can show those messages. Then another type of message is sent next as a request/response pair, where the user requests a “go to definition” of some object under the cursor in the tool, and the server responds by pointing to the location of the definition in the text document.

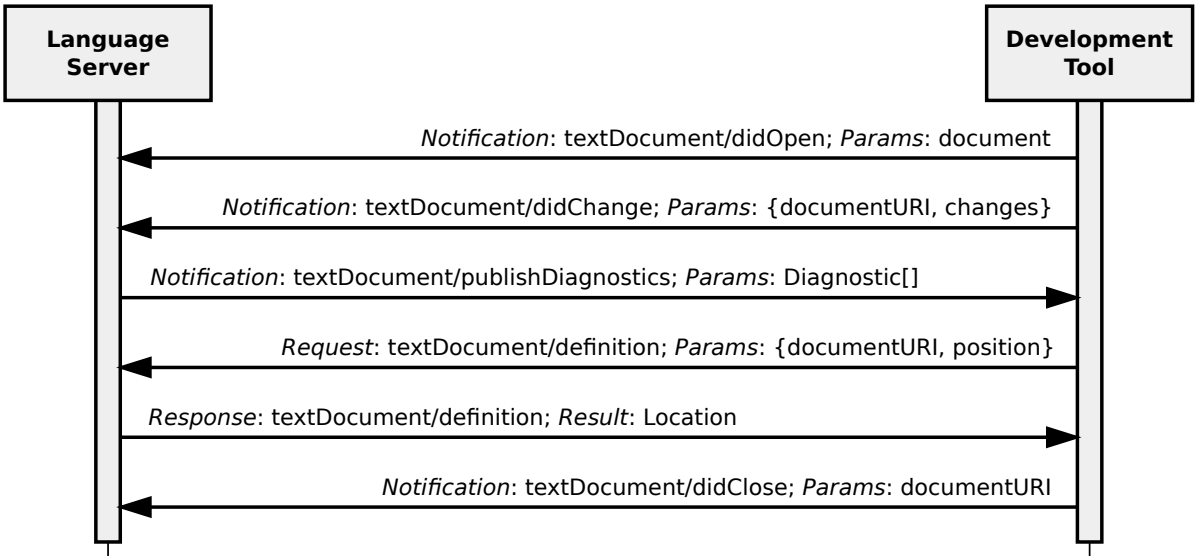


Figure 2.1. Example communication using the LSP [Mic18a].

To summarize, supporting the LSP is a good way to make a language re-usable in many IDEs without having to separately add support for each one.

2.2 Browser Technologies

The next sections explain the frameworks used mainly or exclusively on the client part of the implementation for this thesis.

2.2.1 TypeScript

First let us talk about the programming language used for the client part and some special features of that language. TypeScript is a superset of JavaScript, which is commonly used by nearly all websites on the internet for executing code in the browsers of their users. TypeScript can also be accessed easily, because it is open source and free to use.[3] All of the commonly used browsers support the execution of JavaScript code natively, which is why most websites use that technology, so they do not need the user to install some other non-standard software on their systems first just for viewing simple web pages. However, JavaScript is a "poor language" if larger applications should be developed and maintained, as mentioned by Bierman et al. [BAT14]. That is why Microsoft wanted to assist programmers by extending JavaScript to TypeScript, which compiles back to plain JavaScript code to run on any browser and any operating system that already supports it.

[3] `https://github.com/Microsoft/TypeScript`

The TypeScript language specification written by Microsoft [Mic16] explains most language features with examples in much detail, but it is not complete yet in its current version, as some sections are just filled with a to-do note to be replaced by actual documentation.

TypeScript adds classes and interfaces to be used like an object-oriented programming language and a static type system to catch type mismatches not only at runtime, but already at compile time or even while writing the code in an IDE. TypeScript has access to all JavaScript libraries and can therefore still run into runtime type errors that cannot be statically analyzed by the TypeScript compiler. However, this access and the fact that TypeScript is a superset of JavaScript simplifies migrating systems to TypeScript.

For this thesis, the TypeScript language is used to extend the Theia IDE to alleviate being executable as a desktop or a browser application and being able to use the infrastructure of defined types and objects provided by Theia, because it is also implemented in TypeScript.

2.2.2 JSX

One of the yet undocumented features in TypeScript's language specification is JavaScript XML (JSX), which is also an extension for JavaScript. JSX is an open standard to write XML-like code in JavaScript in a natural and readable way. It was first defined with the JavaScript framework *React* by Facebook [Hun13] to deal with user interfaces with changing content. The specification for JSX is published and maintained by Facebook [Fac14] as well.

Since then React did not stay the only implementation of the framework; another one used for this thesis is *snabbdom-jsx*.[4] React is the default implementation used for JSX though.

Listing 2.1 shows an example of how JSX may be used in TypeScript code. The example is taken from the diagram options view to be explained in Sections 3.5 and 4.2.3. It creates an HTML element to show a box with different options to click based on a given object defining those options. The box is defined in line 2 as an HTML `fieldset` with an attribute named `key` with the value that is evaluated by TypeScript for reading the object `option.name`. Its children are another HTML tag `legend` and some elements created by the `renderChoiceValue`-function for each possible option value. The code represents the resulting HTML closely, as the tag names and attribute names follow the same syntax in both JSX and HTML. The only difference is that the values that have to be evaluated by TypeScript are surrounded by curly brackets. This way, JSX allows to construct XML-like tree structures with attributes that are embedded in JavaScript code in a natural way with exchangeable data. Side effects that should be executed when interacting with the elements are also represented in the code. Looking at line 17, each input field will call another function `onChoice` when clicked, which may change the data given to the displayed options and will re-render them in that case. That is a feature of React to define and modify data views easily through natural looking code using the JSX standard. The alternative to JSX would be to use the `React.createElement` function with typical TypeScript/JavaScript syntax.

Listing 2.2 shows the example from Listing 2.1 translated to plain JavaScript. Here every

[4] `https://github.com/snabbdom-jsx/snabbdom-jsx`

```
renderChoice(option: SynthesisOption): JSX.Element {
   return <fieldset key = {option.name}>
      <legend>{option.name}</legend>
      {option.values.map(value =>
         this.renderChoiceValue(value, option)
      )}
   </fieldset>
}

renderChoiceValue(value: any, option: SynthesisOption): JSX.Element {
   return <div key = {value}>
      <input
         type = "radio"
         id = {value}
         name = {option.name}
         defaultChecked = {value === option.currentValue}
         onClick = {e => this.onChoice(option, value)}
      />
      <label htmlFor = {value}/>
      {value}
   </div>
}
```

Listing 2.1. Example code using JSX.

HTML element was translated to the above mentioned function `React.createElement` with the tag name as its first argument, the attributes in an object defined as key value pairs as its second argument, and the child elements as its remaining arguments. The code layout and indentation help finding which parts belong where, but in general this is far less readable than it was before. The different way of defining functions and the lack of types here are due to this code being translated from TypeScript to JavaScript as well.

In general, JSX and its implementation with React does not add any functionality to JavaScript and therefore TypeScript, but is an extension to facilitate writing code for interfaces for websites.

2.2.3 Theia

The main framework used is the Theia IDE,[5] an open source IDE developed and maintained by `TypeFox` with their source code hosted on a public GitHub repository.[6] It is in an early development phase and receives new updates regularly, but does not yet have as full a feature set as many other IDEs.

Since its first commits in February 2017 Theia was developed with some design decisions in mind, one of them being that it should be implemented with a modern look and feel based

[5] `https://www.theia-ide.org`

[6] `https://github.com/theia-ide`

```
renderChoice = function (option) {
   return React.createElement("fieldset", { key: option.name },
      React.createElement("legend", null, option.name),
      option.values.map(function (value) {
         return this.renderChoiceValue(value, option);
      })
   );
};

renderChoiceValue = function (value, option) {
   return React.createElement("div", { key: value },
      React.createElement("input", {
         type: "radio",
         id: value,
         name: option.name,
         defaultChecked: value === option.currentValue,
         onClick: function (e) { return this.onChoice(option, value); }
      }),
      React.createElement("label", { htmlFor: value }),
      value
   );
};
```

Listing 2.2. Translated JavaScript code from the code in Listing 2.1 without JSX.

on Microsoft's IDE VSCode, with the code split up into a frontend and a backend [Eff17]. In this way, Theia can be started locally as a desktop application like most common IDEs, or as a web-based application with the backend running on a server that is then accessed via a frontend application loaded into the user's local browser. Instead of using the browser, Theia can also be started as an Electron app so it does not need a dedicated browser installation to work. An Electron app bundles a Node.js runtime to execute the backend code of a web application with a Chromium browser to execute the frontend code of a web application. That way, the Theia Electron app can be installed and executed as a single program on multiple platforms[7] with its GUI provided by Chromium. Web-based applications are not a new concept and Theia is one of many IDEs to leverage it, but Theia is special with the additional concepts built into its core.

Other key design decisions for Theia are support for many languages using the LSP, as described in Section 2.1, and being extensible on both the front- and backend, so that a new extension can be developed and installed without needing to rebuild the whole application every single time. Theia itself is built from extensions such as an editor view, Git support, workspace and file system managers, and many more, all implemented in TypeScript (see Section 2.2.1). That confirms that extensions are a core part of Theia's design and can support

[7] https://github.com/electron/electron

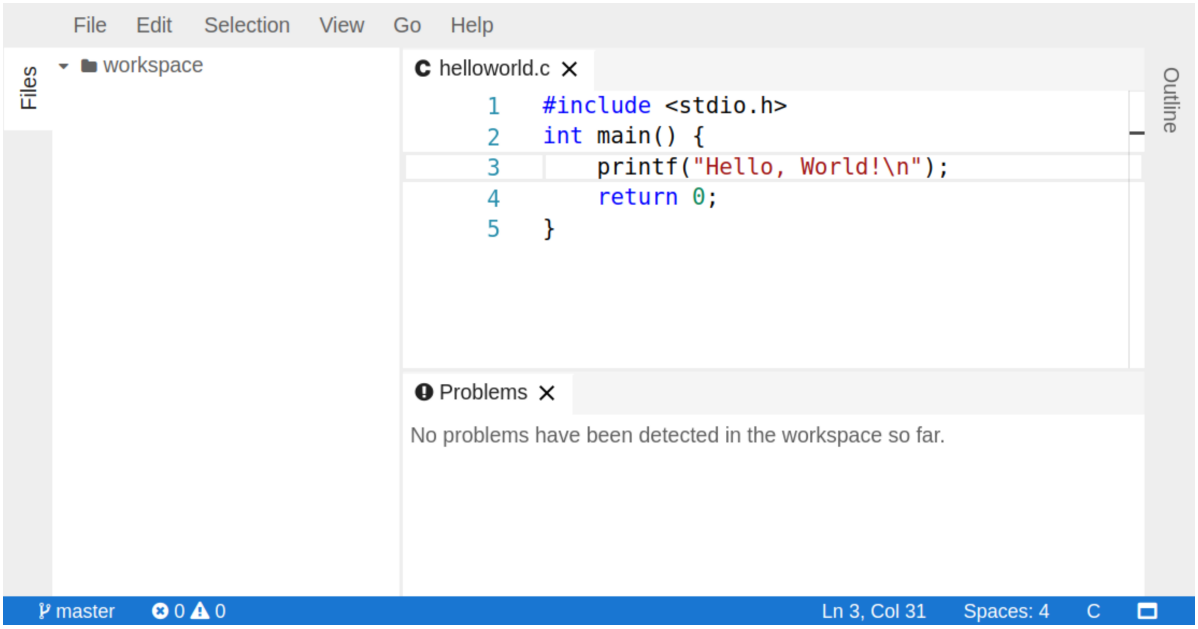


Figure 2.2. Theia's user interface.

a wide variety of functionality, which makes Theia a good candidate to base this thesis' implementation on.

Figure 2.2 shows the user interface of Theia while programming a simple Hello World example in C. It has some of the default extensions activated, such as the text editor, file system access, and syntax highlighting. Language-specific functionality has to be added in the form of extensions to Theia, so that it behaves like an IDE a developer might want to use.

2.2.4 Sprotty

Next to Theia, sprotty is an open source[8] web-based framework that is used for generating diagrams. It was designed to enable showing and working with diagrams in Theia, but developed as an independent framework to use for any application using diagrams. sprotty is also being developed by TypeFox. Therefore it has good compatibility with the concepts used for Theia and already has an integration with Theia to be shown in a widget.

Köhnlein explains the basic functionality and its use in a blog post [Köh17b] and writes a bit more in depth about the combination with Theia, the LSP, and Xtext in another blog post [Köh17a].

For rendering diagrams, sprotty uses Scalable Vector Graphics (SVG), which are supported by most modern browsers, yet, according to the developers, sprotty works best with the Chrome browser. It can be used as a client-only application that holds a model that can be rendered with some interactions with the user. But the more complete way also includes a

[8] https://github.com/theia-ide/sprotty

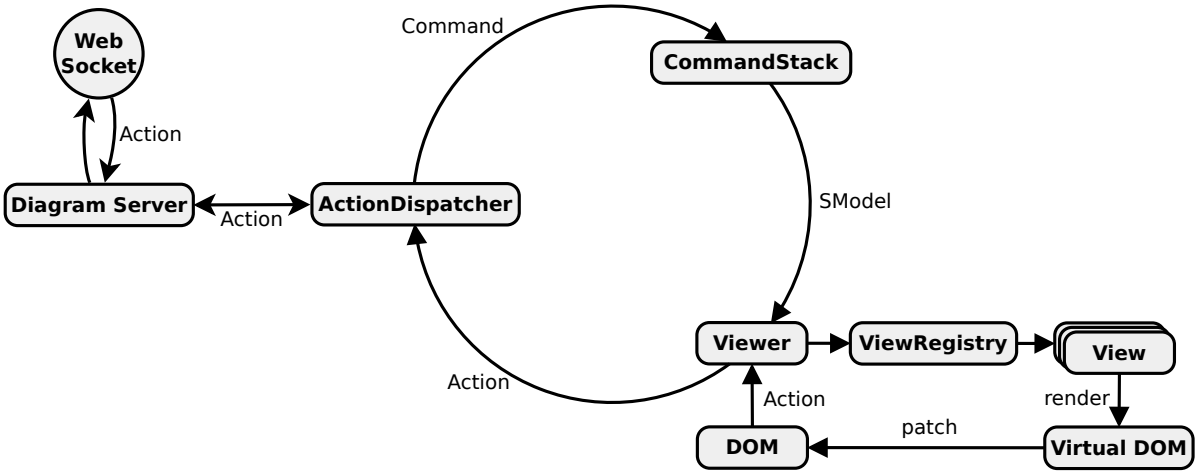


Figure 2.3. sprotty's client architecture: flow of information and the virtual DOM. Diagram inspired by presentation slides for sprotty [KS17].

server that maps an underlying model to diagram elements and handles more resource and memory intensive workloads that should not be executed on the client.

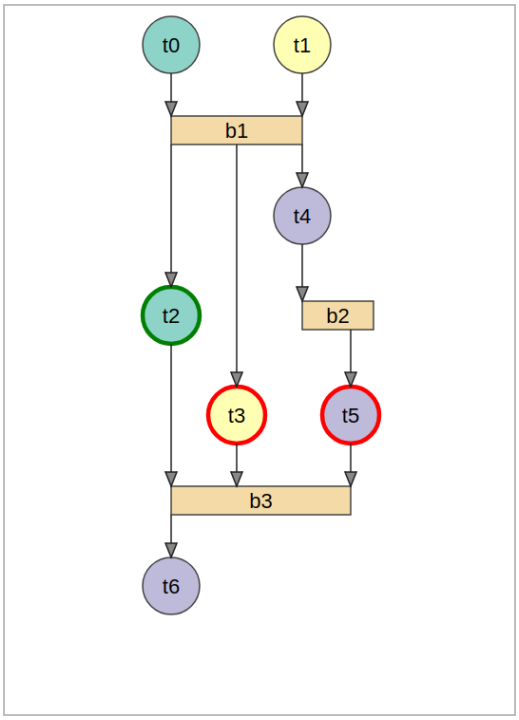
The architecture on the client side does not follow the often used Model View Controller (MVC) concept [Ree79], but rather a modern reactive framework, where information flows in a circle between three components. Figure 2.3 shows this architecture. Messages to and from the server or within sprotty are *actions* that contain some kind of operation, such as requesting a diagram model or setting the bounds of some model elements. Internally in sprotty they are then translated into commands to alter the diagram model state. The sprotty model (`SModel`) can be translated to different views registered in the *view registry*. These different views are comparable to an HTML Document Object Model (DOM) that is able to create a web page view like browsers would show it, or a tree view of all tags, their attributes, and children. Regardless of which sprotty view is configured to be used, they all create an SVG that could directly update the DOM of the SVG shown on screen, but that way the entire model would get changed, which requires a complete new rendering. By just changing a virtual DOM that is not the model rendered to the screen, it can patch only modified parts of the model into the actual DOM to optimize performance.

sprotty's client side and the server communicate their actions via JSON Remote Procedure Call (JSON-RPC) notifications. This extends the LSP by adding one more message to the protocol, the `diagram/accept` message. Usually, every method in the LSP has its own dedicated message in the protocol, but sprotty instead has just one message with a polymorphic action as a parameter, so that users of sprotty can add custom actions "without having to fiddle with the details of JSON-RPC," as Köhnlein puts it [Köh17a].

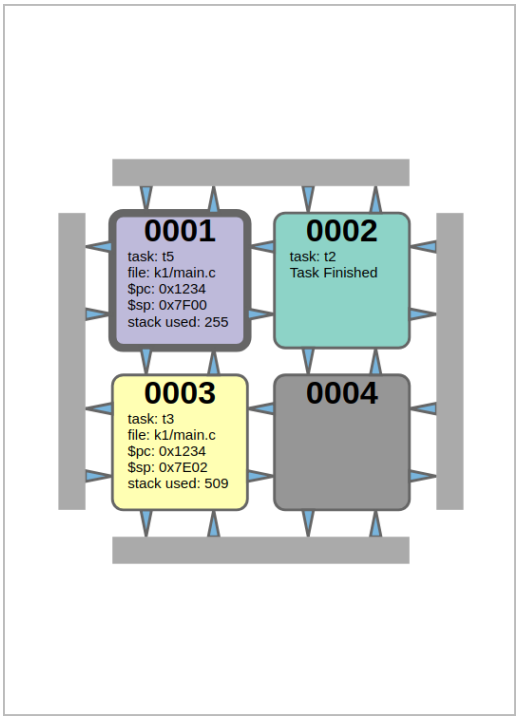
An example application of two sprotty views generated for a single source model can be seen in Figure 2.4. Figure 2.4a displays an editor of some DSL generated by Xtext for defining the flow of control for a parallel algorithm. The algorithm consists of tasks and barriers

```
29  barrier b3: join t2, t3, t5
30  then task t6: execute k2
31
32  step 1 {
33      core 1 runs t0 { $pc=0x1234 $sp = 0x
34      core 2 runs t1 { $pc=0x1234 $sp = 0x
35  }
36
37  step 2 {
38      core 1 finished t0 { $pc = 0xFFFF }
39      core 2 runs t1 { $pc=0x1234 $sp=0x7E
40  }
41
42  step 3 {
43      core 2 runs t2 { $pc=0x1234 $sp=0x7E
44      core 3 runs t3 { $pc=0x1234 $sp=0x7D
45      core 4 runs t4 { $pc=0x1234 $sp=0x7E
46  }
47
48  step 4 {
49      core 2 finished t2 { $pc = 0xFFFF }
50      core 1 runs t5 { $pc=0x1234 $sp=0x7F
51      core 3 runs t3 { $pc=0x1234 $sp=0x7E
52  }
```

(a) Editor with the algorithm's source model.



(b) The algorithm's structure as a sprotty diagram.



(c) The algorithm's core and memory allocation as a sprotty diagram.

Figure 2.4. Example application with two sprotty views of a parallel algorithm with server communication. The application runs on sprotty's demo page.[a]

[a] `http://sprotty-demo.typefox.io/examples/index.html`

defining flow dependencies between the tasks and an allocation of cores and memory for each task. The server then generates diagrams for both sprotty views shown in Figures 2.4b and 2.4c (in this case for Step 4 of the algorithm defined in the editor). Clicking on different objects in the editor or the flow diagram then sends an action to the event cycle and requests new diagrams from the server, depending on which element and step in the execution should be highlighted.

2.3 Server Technologies

I will now introduce the technologies and frameworks used on the server side of the LSP for the implementation of this thesis.

2.3.1 Xtend

The main programming language used for the server side implementation is the language Xtend, a Java dialect that compiles to Java code. Xtend is an open source project initially implemented for the openArchitectureWare (oAW), a model-driven software development platform that now is a sub-project of Eclipse as mentioned by Efftinge et al. [EV06]. Since then oAW has moved away from Eclipse, while Xtend and Xtext (see next section) are still developed by the Eclipse Foundation in the Eclipse Modeling Framework (EMF).[9]

Xtend aims to make coding for the Java Virtual Machine (JVM) easier and the code more comprehensible while keeping the performance of the code as it would have been just coding in Java. For example, it simplifies the creation and interaction with different collections, automatically infers types for variables and methods, and treats everything as an expression. This way the results of if-else expressions can be assigned to variables, similar to how the ternary operator works. Generating arbitrary code from Xtend is also simplified by the addition of template expressions. With them, strings are constructed and customized in a readable fashion. Also, extension methods can be defined and then called on any object, as if it was part of the object's type. Another very important aspect is that Java and Xtend are completely interoperable, as Java libraries can be used in Xtend natively and *vice versa*. For more information I refer the reader to Xtend's official documentation.[10]

Xtend is used to extend the language server, one of the core parts of the implementation, as described in Chapter 4.

2.3.2 Xtext

Xtext is a grammar language used for developing DSLs and programming languages. Languages can be defined in a grammar that is well documented with examples and tutorials,[11] and Xtext produces infrastructure needed to work with that language. For example, Xtend is

[9] https://wiki.eclipse.org/OAW

[10] https://www.eclipse.org/xtend/documentation/index.html

[11] http://www.eclipse.org/Xtext/documentation/index.html

written in Xtext, while the code generator for Xtend is written with Xtext's generated tooling support. Since both projects are developed together, this is a proof that both systems work and are usable as they are targeted. Xtend uses many of Xtext's features to make it a working language. For example, Xtext aids generating code from a domain model parsed from the grammar. In this case, Xtend needs to be translated to Java code which can be executed then. This is also described in the book about Xtext and Xtend by Bettini [Bet16]. Next to the code generation Xtext also helps working with content assist in editors for Xtext languages, showing an outline of the model, scoping of references between models, interpreting or compiling the model, and much more.

The important aspects for this project are that Xtext is able to create a parser and linker to generate a browseable EMF `EObject` as well as an executable language server following the LSP for every defined DSL. A model to work with the language and a language server to communicate with editors is already generated by Xtext and only needs to be extended for further use as all used DSLs for this project have an Xtext grammar. An extension to the language server generated by Xtext for sprotty is already implemented as well[12] to make the beginning for working on the language server easier.

The first version published by Efftinge et al. [EV06] was written "xText" and published as an Eclipse oAW project, similar to Xtend. It is established open source technology in the Eclipse world that is being adopted by larger companies[13] and developed and supported mainly by TypeFox [Eys17] and Itemis [Bün18]. A full user guide for version 1.0 was published by Behrens et al. [BCE+08] and is maintained online on the websites of Eclipse since then.[14]

2.3.3 Gson

Gson[15] is an open source Java library developed by Google to serialize Java objects to JSON and deserialize JSON objects back to Java. It is able to take any Java object, regardless of whether it is a primitive variable, a complex object with deep hierarchy, or some self-defined class, and turn it into a human- and machine-readable JSON representation. It is useful for providing an API to other programs written in languages other than Java and is even better for communicating with JavaScript programs, since it is the natural form of writing JavaScript objects. Gson is also configurable for any object or field to serialize them differently. This is an essential feature that makes using Gson over one of the many other JSON serialization libraries the best choice for this project.

When using serializations like Gson, it is important to notice that working with serialized objects is not the same as in Java. In Java the objects are instances of some class which also defines methods and describes behavior, such as a *car* being able to reduce its *speed* by *braking*. The JSON representation of such a car would only include the valued field speed, as

[12] `https://github.com/theia-ide/sprotty/tree/master/server/xtext-diagram`

[13] Adopters, such as Google, Siemens and Bosch are listed on Xtext's main page `http://www.eclipse.org/Xtext/index.html`.

[14] `http://www.eclipse.org/Xtext/documentation/index.html`

[15] The name Gson is a mix of Google and JavaScript Object Notation (JSON).

braking is a process that changes the value of the speed and cannot be described as a single value. So serialized Java objects only contain their fields and not their methods, which would help to fully describe the behavior of the objects. JSON objects as the basis of sending data from one system to another leaves the implementation of the behavior up to both systems independently.

Since Gson is an open source project, the documentation and a user guide can be found online on their GitHub pages.[16]

2.3.4 Eclipse Layout Kernel

The Eclipse Layout Kernel (ELK) is a framework started for the KIELER project under the name of KIELER Infrastructure for Meta Layout (KIML) by Spönemann [Spö15] to provide algorithms and an infrastructure to generate automatic layouts for graph structures of nodes with ports and edges. As the name suggests it has become an Eclipse project[17] and is also used for projects outside of KIELER. Ptolemy[18] and sprotty (see Section 2.2.4) are projects that use ELK for automatic layout of their diagrams.

ELK is based on an own graph structure and provides two ways to connect with the framework. On the one hand it provides an interface for viewers to register a *diagram layout connector* that is able to map between the view model used by the viewer and the layout model used by ELK's layout algorithms. On the other hand it also provides an interface for implementations of layout algorithms to register a *layout provider*.

With that structure, any registered layout provider can be used by any diagram layout connector over the common graph structure, the *ELK graph*. Instead of just providing a generic layout for every call, ELK can be configured with layout options that the algorithms define and use to change some aesthetics in the layout, such as a default spacing between nodes and edges, layout directions, and so on.

This means that ELK provides easy and configurable access to users wanting to have automatic layout on their view models and layout developers can quickly implement new algorithms and test them in different use cases.

2.3.5 KLighD

KIELER Lightweight Diagrams (KLighD) is a framework to translate any data structure in Java to a view model that can be rendered easily. The framework was introduced by Schneider et al. [SSH13] to efficiently create interactive diagrams with little time investment. To reach this goal, KLighD manages multiple steps for generating a view. It is based on the `KGraph` structure to provide an interface for defining models as a graph structure with rendering information (`KRendering`) and data for positioning the graph elements (`KLayoutData`). To get a `KGraph` representation, a synthesis from the source data structure to a `KGraph` instance has

[16] `https://github.com/google/gson/blob/master/UserGuide.md`

[17] `https://www.eclipse.org/elk/`

[18] `https://ptolemy.berkeley.edu/ptolemyII/index.htm`

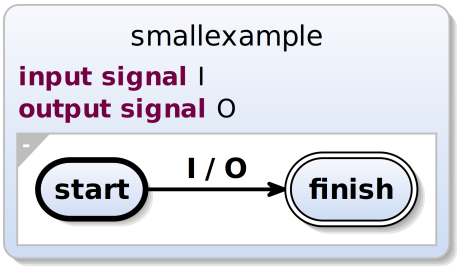


Figure 2.5. The diagram visualization of a small example SCChart.

to be provided that describes the visual representation of each source object. The visual representation for a node for example contains text, circles, rectangles, specific shapes, and an area to contain the hierarchic children of that node. The synthesis for a structure such as a tree with a root, children, and numbered leafs could translate each element to nodes with different shapes, with each element connected by a directed edge defining the parent-child relationship of each tree element. A more complex synthesis that translates every part of the language SCCharts (see Section 2.3.6) shows the power and diversity that is possible with `KGraphs`.

The synthesis only provides the graph structure with rendering information. KLighD then calculates the size of each element and calls ELK's automatic layout to get their final positioning. The `KGraph` elements are then handed over to the Piccolo2D graphics framework [BGM04] to be rendered on screen based on the positioning and rendering information provided.

KLighD is also able to evaluate different options to change the resulting graph model through options provided by the synthesis and handle actions associated with interacting with different diagram elements. For both options and actions KLighD may need to rerun the layout by ELK or even execute the synthesis again to generate a model fitting the new options.

As KLighD is the main framework that is needed for the automatic views in KIELER it needs to be adapted in a web-based environment. More details on KLighD and the new approach are in Chapters 3 and 4.

2.3.6 SCCharts

The Real-Time and Embedded Systems Group at Kiel University's Department of Computer Science develops KIELER as a platform and IDE to host many projects that are being worked and researched on. Sequentially Constructive Statecharts (SCCharts), a visual safety-critical language defined by von Hanxleden et al. [HDM+13], is one of the bigger projects of this group. It is a dialect of StateCharts and is supposed to be used in combination with the transient views generated by KLighD, to have real-time feedback of the modeled diagram while working in a textual editor.

A small example SCChart is pictured in Figure 2.5. It describes the behavior of a state machine that is waiting for an input `I` to happen and returns the output `O` once and then finishes execution. Being able to display SCCharts, such as the one in the example, will be

used as a benchmark for this project. Because SCCharts already uses most of the features for diagram generation by KLighD, it is a good way to measure how well the implementation matches the requirements of displaying a diagram correctly.

Chapter 3

Concepts

Before looking into the implementation, this chapter presents the concepts, ideas, and discussions about how the whole implementation and its parts should work in different levels of detail. Since the implementation is split up into a server and a client part, it shows the concepts of each part and of the communication between them. At first this chapter will look at the framework KLighD that is used for the generation of diagrams in KIELER and what that approach translates to when using web technologies.

3.1 Overall Architecture

Figure 3.1 shows how the framework KLighD used in KIELER works internally and which steps are required to generate and update diagrams. KLighD gets a domain model that it should

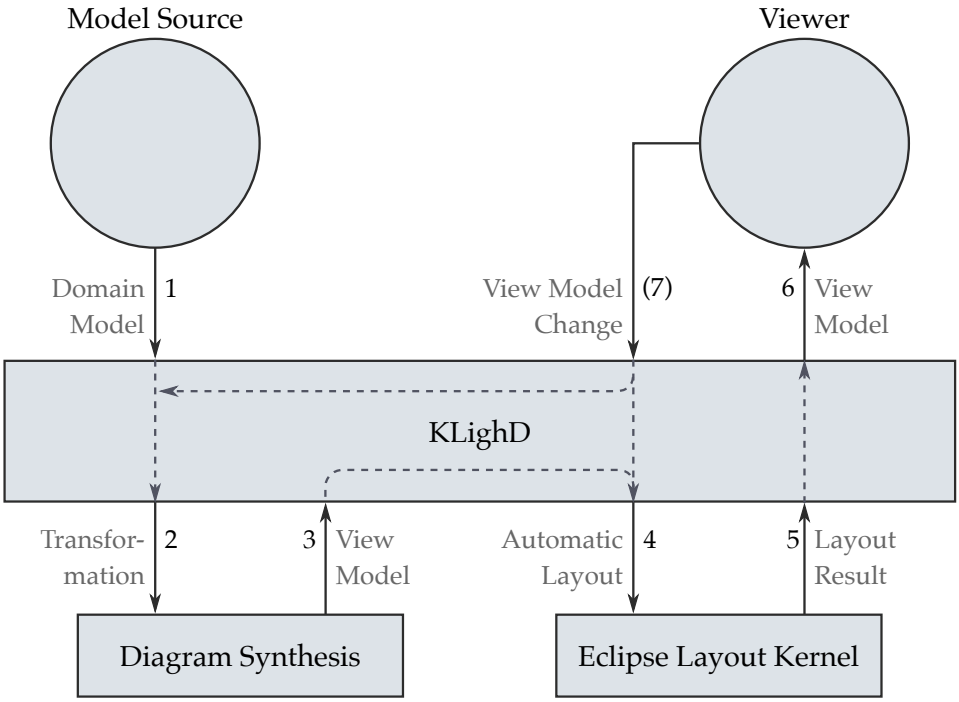


Figure 3.1. KIELER Lightweight Diagrams: the road from a textual model to a drawn diagram. This diagram is provided courtesy of Dipl.-Inf. Christoph Daniel Schulze.

draw in Step 1. The domain model is a Java object model describing the objects that the text is describing and is usually parsed with a grammar. In Step 2 it looks for a diagram synthesis that is able to transform the model into an internal view model that KLighD knows how to render. If that exists, Step 3 returns said view model. It usually contains no information on the positioning of the view model elements yet, so KLighD invokes the automatic layout provided by ELK in Step 4 and attaches the result in Step 5 to the internal view model. This updated view model then gets transferred to a viewing framework that knows how to draw the view model in Step 6.

After that step, the request to KLighD to draw a diagram is complete, but user interactions that may change the view are also accounted for. KLighD allows options to be defined and changed, which may cause the diagram synthesis to return a different view model. So upon change of such an option in Step 7, KLighD needs to re-run the process from Step 2. Defined actions in the view model may instead cause another of the multiple renderings of an element in the view model to become active such that the view model stays the same, but the active rendering might have changed. In that case Step 7 just needs to call ELK again from Step 4 in the diagram. Then a new layout needs to be calculated because the rendering in the view model might have a different sized representation.

A concept similar to this will be implemented in another environment based on web technologies. The model source will be stored somewhere on a server and the viewer will be on a client that a user can interact with to call for and interact with diagrams. The concept of a server and a client should definitely be implemented, so that the client can even run on some lower-end hardware, such as laptops of tablets, and the server can do the more complex computations. Also, the client should stay as lightweight as possible to minimize the memory footprint on the browser and the loading time of the page.

By taking a look at the frameworks used by KLighD, we can analyze which of them can be reused in a web-based environment. The diagrams drawn by KLighD come from arbitrary object models within the EMF or for us often from languages with a grammar written in Xtext. Xtext is able to generate a domain model from a text source and is not bound to Eclipse, so the first part can also be used in the web. The view model and the transformation to that view model is a little more difficult as it requires a diagram synthesis, a diagram layout connector, and compatibility to the viewing framework in the web to be programmed. Currently KLighD uses the `KGraph` model that the domain model is translated to. The `KGraph` can be layouted with ELK and can be drawn with the graphics framework Piccolo2D with a Standard Widget Toolkit (SWT) implementation in Eclipse. Because of implementations for both frameworks handling the `KGraph`, Piccolo2D and its implementation to render `KGraphs` cannot be reused in the web environment because the implementation is specific to Eclipse. So the decision on the view model informs what has to be done with the synthesis and the layout and if they can be reused.

Figure 3.2 shows an abstract view on an idea to what a counterpart to KLighD could look like in a web environment with a server and a client. The server and client (or browser) both need access to the model source. The client needs it to show it in a text editor, so the user

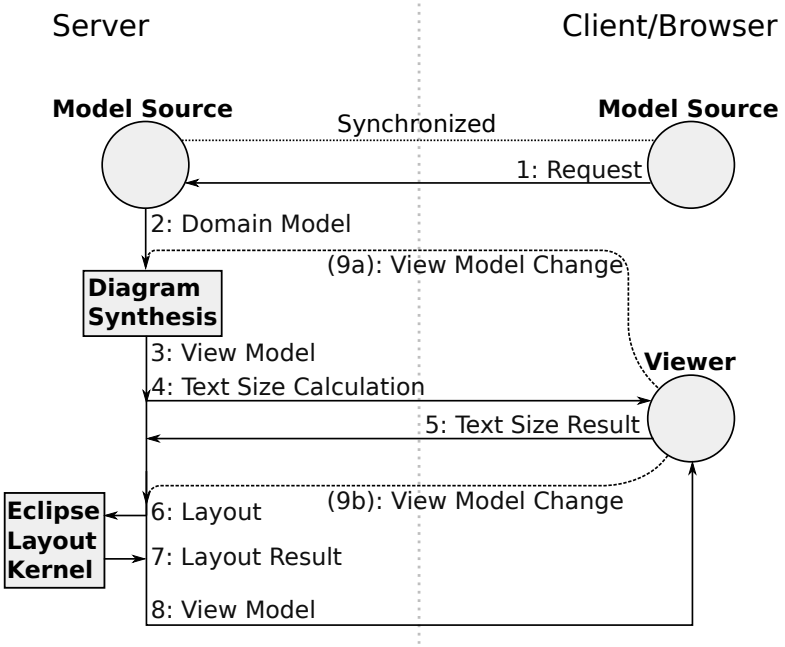


Figure 3.2. The architecture for diagram visualization in a client-server environment.

can view and modify it, and the server needs access to it to generate the domain model and the view model. Both sides need to be synchronized, so that the diagram generation always provides views matched to the source that the user can see.

In Step 1 of the new architecture the user requests a diagram view for the model source shown in their browser. Since the generation should not happen on the client side, the request is forwarded to the server that knows which model source the request belongs to.

Step 2 translates the source code to an internal domain model understood by the server, similar to Step 1 from Figure 3.1. Since this is an arbitrary data structure defined by the domain model, it still needs to be translated to a view model that defines the visual structure of the model.

This is the same principle from KLighD, in the sense that a diagram synthesis creates a view model from the domain model in Step 3. If the domain model and the view model stay the same as they were in KLighD, the same syntheses can be used and do not need to be rewritten.

The next steps are new and caused by splitting the framework into a server and a client side. The view model contains rendering information with absolute or relative sizes for all shapes, but for texts it only contains hints on the used font, size etc. but not specific widths and heights. In the next step KLighD can just try to render the texts in its viewing framework and use the size of those renderings as estimations of the size the text will require when rendered on the client. However, the text size estimation from the server does not exactly represent how big the text will really be rendered on the browser. The server and client could run on separate machines with different operating systems and the server does not know

which fonts are supported by the client. Even if it did, it could not know whether the fonts were really rendered the same way on both machines. Implementations of fonts can differ on different operating systems and the way they are rendered is not consistent in different browsers. As a result Step 4 requests the viewer in the browser to calculate the sizes it needs to render all texts in the view model.

Step 5 then is the answer of the browser with the resulting bounds of all texts. The result then gets integrated in the view model, so the server can now compute exact sizes for each view model element needed for the next step.

Now the flow is back at a point that has a counterpart with the behavior of KLighD. Step 6 hands the view model over to ELK to generate an automatic layout of the graph structure of the view model. The diagram synthesis already generates a model describing the renderings of each individual graph element, but not how these renderings should be arranged on a drawing plane.

In Step 7 the result of ELK's layout again gets mapped back to the view model, which now is ready to get drawn, since every position, size, and the overall look of the graph elements are stored in the view model.

As the 8th and last step the view model has to be transferred back to the viewer on the client side which then handles the creation of a drawing.

When the user now interacts with the view, they can do so in two different ways. The *first* way is through changing options defined by the diagram synthesis from Step 3. Every synthesis can define a number of options that change either how the synthesis itself should generate the view model (such as an option between a fancy colorful view model and a simple black and white one), or how the layout should be configured (such as an option which controls the spacing between nodes and edges). When such an option is changed, the process needs to be started again either from Step 2 so the synthesis can generate a new view model (Step 9a), or it needs to be started again from Step 6 and just recalculate the layout on the view model still stored on the server with the new layout options (Step 9b).

The *second* way is through triggering actions defined during the diagram synthesis. For example, the SCChart from Figure 2.5 has a minus sign in the top left corner of the white region containing the states. When double-clicking on that corner, the region containing the states should collapse and not show the child states anymore. That causes the size of the surrounding rendering to update, so ELK is called again by starting the process at Step 6 (caused by Step 9b again).

3.2 Choice of Frameworks

As the introduction of this chapter and Chapter 1 explained, the new implementation of the IDE next to KIELER should fit in the trend of using web technologies and thus should be separated into a server and a client part. However, just starting to program everything from scratch is way too much work, especially when frameworks for IDEs, drawing diagrams, and the communication between the server and the client side already exist. Also, the LSP

significantly reduces the required effort. As the languages SCCharts, KGraph, Lustre, SCL, SCG, and others used in KIELER are modeled within the EMF and have an Xtext grammar, they all are already able to generate a working language server following the LSP that automatically handles many language features in any IDE supporting the protocol as well. So we are looking for a web-based IDE supporting the LSP in the best case.

3.2.1 Why Theia?

Looking at the list of IDEs or programming tools supporting the LSP,[1] there already is a good choice to pick the next environment to host the functionality of KIELER.

The list states full support of the LSP by *Eclipse*. However, there is no need to discuss that option, since it already is the basis of KIELER.

A first real candidate is *IntelliJ*.[2] It is an IDE where the community version is open source. The full professional version is not accessible for free and the LSP plugin was developed externally, though,[3] so working with that could cause problems that may not be solved by the developers of IntelliJ. Also, IntelliJ is just a classic desktop application, so a client-server architecture was not a design choice there. Other known editors that also support the LSP are *Vim*,[4] *Sublime*,[5] and *Emacs*.[6] All of them are desktop apps with externally added support for the LSP via plugins and are stated as not really being IDEs in the internet.[7,8] Vim and Emacs both have a keyboard based navigation concept, which is not a very modern style and some, especially younger people, are not used to that style of using software. That is especially because most other default text editors have a different concept and their shortcuts are very different. They are also not meant to provide graphical frameworks for my use case. Sublime has the additional drawback that it is not open source, so support for it and the implementation can be more difficult than it is for open source software, where one can still look up what is happening in the source code when facing problems if nothing else helps.

Another honorable mention is the text editor *Monaco*.[9] It is the first editor in this list based on web technologies. However, it is not meant to be an IDE, but was pulled out of the source code of the text editor of VSCode to run separate from that IDE in any web application. Since it is just a web-based text editor, its features can be used for any other IDE as well, if they base the text editor functionality on Monaco. And looking at VSCode's editor functionality it seems fast and modern with many features, of course at least the ones defined by the LSP.

Speaking of *VSCode*,[10] it also supports the LSP because of the Monaco editor. VSCode is also

[1] https://langserver.org/

[2] https://www.jetbrains.com/idea/

[3] https://github.com/gtache/intellij-lsp

[4] https://www.vim.org/

[5] https://www.sublimetext.com/

[6] https://www.gnu.org/software/emacs/

[7] https://css-tricks.com/turn-sublime-text-3-into-a-javascript-ide/

[8] https://www.quora.com/Why-do-so-many-programmers-prefer-vi-or-Emacs-to-using-an-IDE

[9] https://github.com/Microsoft/monaco-editor

[10] https://code.visualstudio.com/

open source[11] and supports the LSP by design, which can be seen since both the LSP and VSCode are developed parallel to each other by Microsoft and since the support of language servers is a documented feature [Mic18c]. It also has good extension support and a very new and modern look and feel. VSCode is newer in the field of IDEs than most other mentioned ones and has the goal to be an environment to host support for as many languages as possible. To achieve that, extension support is also built in as a big feature. Once VSCode sees you working on a document with a file extension that you have no extension installed for, it automatically suggests to go ahead and install the extension for that language, so you will get the best possible comfort while developing. This is a good candidate to get out of the Eclipse environment. However, there is one disadvantage when using VSCode. It is using web technologies inside a desktop application, but has no intention to run in a backend server separated from its frontend, so the application still has to be downloaded and executed instead of running on some website in any browser.

There is also a big range of IDEs that use a web-based approach already to provide IDE support in the browser and in the cloud and are either commercial, free, or open source. The commercial products, such as *AWS Cloud9* by Amazon[12] and *PaizaCloud* by Gino,[13] can be excluded from the discussion immediately, as the new framework used in the future should be available free of charge for anyone to use. Then there is the open source cloud IDE *Eclipse Che*,[14] which is backed by a server and can be hosted anywhere or leased on websites such as Codenvy.[15] However, because one of the goals is to move away from Eclipse towards a completely new platform to create ways to research new and different concepts, staying on the same basic platform with Eclipse does not fit that criterion. There is another open source web-based IDE called *Light Table*.[16] It was developed as a full IDE running in a Chrome environment to develop and view web content with instant feedback with JavaScript and Clojure, the language Light Table itself was developed in. It was funded as a project on Kickstarter[17] and developed by a team with that money. What it is missing, though, is support for many languages, especially for DSLs with Xtext and the LSP.

The list of tools supporting the LSP contains one last candidate for an IDE that also uses web technologies and seems to be crafted just perfectly for our use case, and that is *Theia*.[18] Theia is an IDE built with web technologies and can be started as a typical desktop application or in a web browser with a backend server. It was designed with VSCode in mind and is called "VSCode in the cloud" by the developers of Theia [Eff17]. It also uses Monaco as its text editor and has LSP support as one of its key design features. Extension support is also viewed as a number one priority. Almost every feature of Theia is realized through an extension bundled

[11] https://github.com/Microsoft/vscode

[12] https://aws.amazon.com/cloud9/

[13] https://paiza.cloud/en/

[14] https://www.eclipse.org/che/

[15] https://codenvy.com/

[16] http://lighttable.com/

[17] https://www.kickstarter.com/projects/ibdknox/light-table

[18] https://www.theia-ide.org/

into the Theia application. That includes the text editor Monaco, access to the file system, Git support, the views showing problems and errors, and many more. Like in VSCode, the design is kept simple with a modern look and feel. Another big plus for Theia is the support for the diagram framework *sprotty* (see next section) that is designed to work with Theia and supports arbitrary data to be visualized easily.

So Theia has support for the important parts used in the diagram generation in KIELER: On the server side there can be Xtext generated language servers following the LSP, which is natively supported by Theia and the Monaco editor, and a viewer supporting some view model that will draw a diagram within Theia. However, Theia also has some drawbacks, which is why it is the best choice with some restrictions. Theia is still in an early access phase and does not support as many features as other IDEs do. Therefore the user experience can suffer, as for example a debug functionality is still missing (although it is currently being worked on). Nevertheless, I think Theia is a good choice to try the new development on, because it is open source and still in development to improve in the future and because of the wide range of use cases possible when executed in any browser.

This is by far not a complete list of IDEs that could be tested to start a new implementation of KIELER. Taking an IDE that supports the LSP is very valuable though, because that saves much work for developing that support in the first place. The thesis written by Domrös [Dom18] especially would have a much harder time even getting basic language support for our specific languages. Furthermore, since his and my work should work together, that choice applies to me too. Also, the LSP is a good standard to work on for better language support for everyone in my opinion, because it makes reusability of implementations between tools very easy and allows IDEs to quickly support a wide variety of languages by just implementing the LSP API once.

Because the implementations of Domrös and me put together form the basis of a new IDE based on Theia that bases its functionality on the KIELER IDE, the new IDE gets the name Kiel Environment Integrated in Theia (KEITH), which should be reminiscing of the old name Kiel Integrated Environment for Layout Eclipse RichClient (KIELER), but with Theia instead of Eclipse as its platform.

3.2.2 Working with Sprotty

When using Theia and the LSP with Xtext generated language servers, also working with sprotty for generating diagrams is a very convenient choice, because sprotty was designed in part for exactly that use case.

Like the LSP, sprotty also communicates via JSON-RPC over the gap between the server and the client. And because there already is a language server holding the domain model that should be visualized, and the client already listens to that server to update its views, sprotty just sits right on top of that architecture. It extends the LSP by the `diagram/accept` method with a parameter containing an action as mentioned in Section 2.2.4. Actions sent to the language server are not executed on the client, because they need additional information that the server with the complete domain model can calculate. For example, the `RequestModelAction` is sent

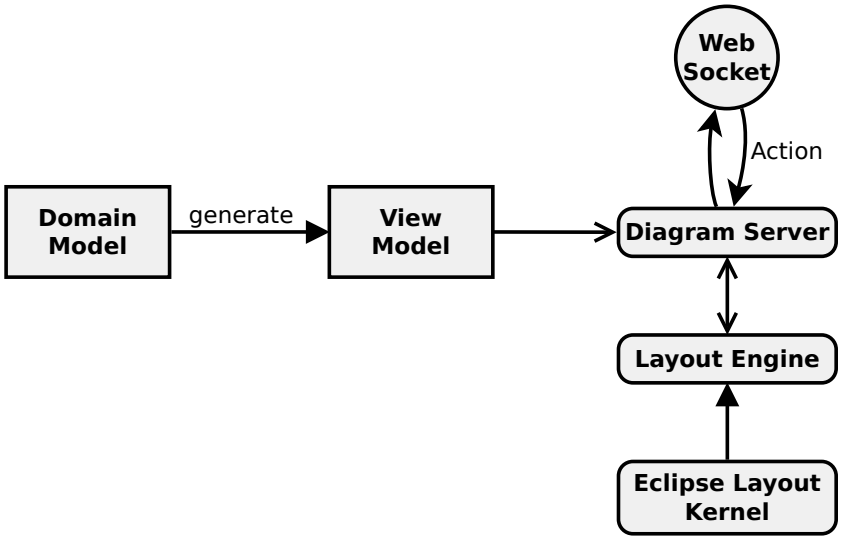


Figure 3.3. sprotty's server architecture. Shows a generic way to allow a diagram server to handle a view model and a layout engine with communication of actions to a server. Diagram inspired by presentation slides for sprotty [KS17].

by sprotty's client side to the server to generate a view model, which can then be rendered on the client side. Since this is a generic action needed for all projects that want to visualize their DSL generated by Xtext and use sprotty and the LSP, the `RequestModelAction` and some other actions are already implemented by sprotty. A diagram server supporting that language is called to handle this action and generate a diagram.

It is configurable by Guice modules via dependency injection. That means, the specific modules extending sprotty's architecture are started by the generic sprotty code without direct knowledge of the specific classes when starting the language server. sprotty will ask Guice for a diagram server implementing sprotty's diagram server interface and it will provide an instance of a more specific class configured with Guice at the start of the server. The same concept is used for more modules used by sprotty on both the server and the client side to make the code easier to understand and maintain. Section 4.1.1 explains this further.

sprotty also provides generic code for the view model transformation of any domain object to its view model (the `SGraph` model) and functionality to automatically layout the `SGraph` on the server and client via ELK, the same framework used by KIELER and KLighD. Figure 3.3 shows the approach on the server side. The view model generation, the layout engine, and even the diagram server itself from the figure can be extended and changed to meet the requirements of the application.

Now the architecture of sprotty looks quite similar to the architecture of KLighD, with their generation of a view model from the domain model, their inclusion of ELK and handing the view models over to a rendering framework. The difference between the sprotty and KLighD view models and how they are both indispensable is explained in the next section.

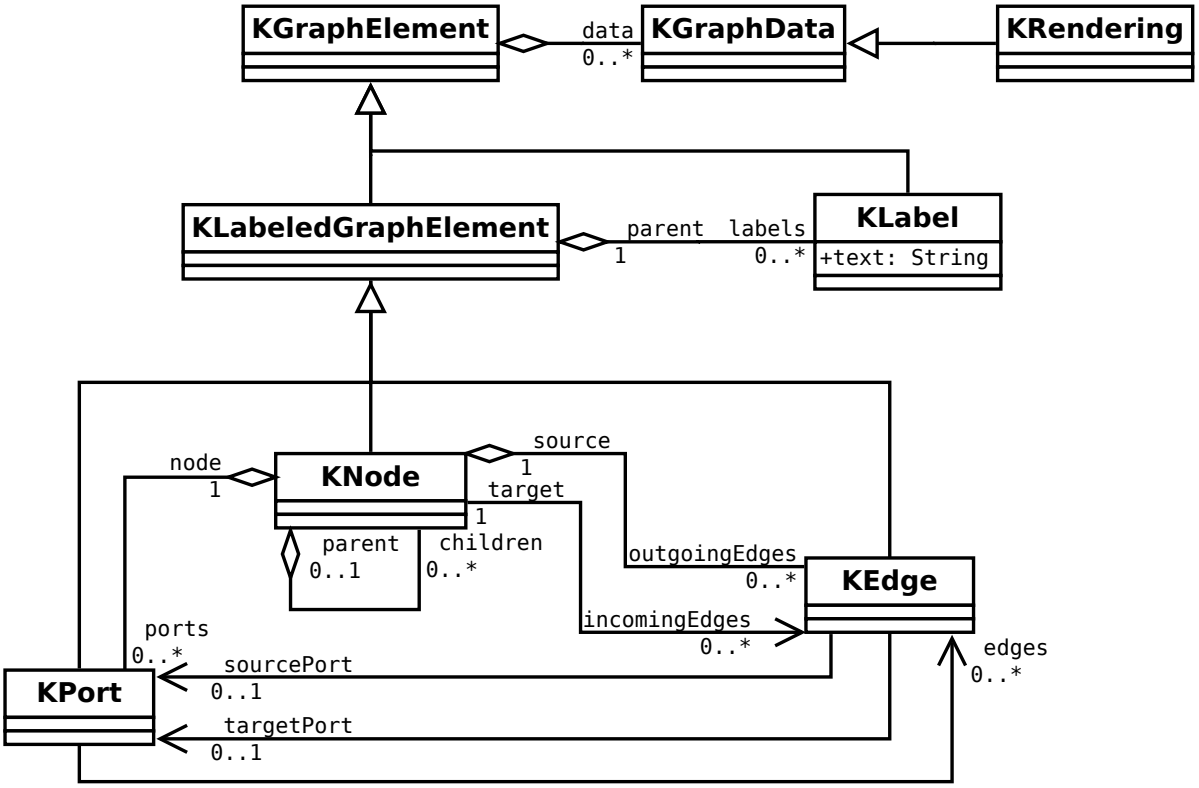


Figure 3.4. The KGraph view model.

3.3 The View Model

To stay with the architecture of KLighD and to have many reusable parts it would be good to have the KGraph model as the view model. To transition to the new architecture with sprotty, it would be good to have the SGraph model as the view model. Looking at their structure shows similarities, but also makes the fundamental differences in their design visible.

Figure 3.4 shows a class diagram of the KGraph model. KGraphs are made up of *nodes* (KNode) connected by *edges* (KEdge), where *ports* (KPort) connected to nodes can be the anchor points of the edges. The nodes are modeled as the main objects, as they contain their attached ports, further nodes as their children, and all outgoing edges. All elements know which other elements they contain and in which elements they are contained. Ports and edges are additionally linked if they are connected to each other. All these elements are KLabeledGraphElements and can contain *labels* with an identifying text. With that added, they all form KGraphElements.

These graph elements map nicely to the graph structure used by ELK and can therefore be easily mapped to and from ELK's layout structure to calculate positions of all KGraphElements by a diagram layout connector. The layout data is then stored in all elements directly in fields

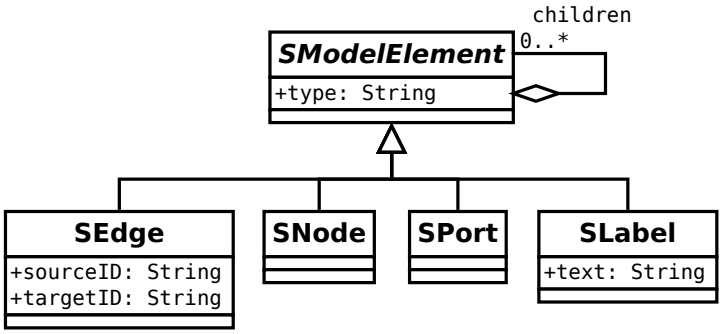


Figure 3.5. A simplified `SGraph` view model on the Java side.

not shown in the diagram (edges have bend points to define their layout while nodes, ports, and labels extend another class `KShapeLayout` to store their position and size directly).

The visualization is then attached to each `KGraphElement` in their `data` field as a `KRendering`, so the specific look of each element in this model is directly attached to each element separately. For the visualization and the automatic layout calculated by ELK (here also called *macro layout*) each node should already have a defined width and height, so the macro layout is able to position the nodes and route edges around them without intersecting nodes and edges. So the bounds have to be calculated first by the so-called *micro layout*, which evaluates the attached `KRendering` and remembers the resulting size for the macro layout.

The `KGraph` view model should not be extended; every piece of information needed for the visualization is either defined by or can be calculated with the `KRendering`s and then a viewing framework can render the model. As the `KGraph` model is defined as a Java object model, it cannot yet be understood on the client side by the sprotty viewer, since it is implemented in TypeScript and therefore would first need a JavaScript representation.

The `SGraph` model by sprotty has a different concept, though. Figure 3.5 shows a simplified version of the `SGraph` model implemented for the JVM. Like the `KGraph` model, the `SGraph` model is also made up of nodes, edges, ports, and labels, representing the same structure. As the names suggest, they are the `SNode`, `SEdge`, `SPort`, and `SLabel`, respectively. However, here all elements extend from the `SModelElement` class and have the only connection in a parent-child aggregation, where every model element can contain other model elements. If a node for example has labels and some ports, they are all `children` of that element. On the TypeScript side the connections are more restrictive, but the general concept is the same. Restrictions such as not allowing labels to be children of labels and ports to only be children of nodes would take effect here. That is the same concept as in the `KGraph` model, although edges are handled differently. For `KGraph`, two nodes connected with each other with an edge would be modeled as two `KNode` siblings with the `KEdge` as the outgoing edge in one of the nodes, where siblings are elements that share a common parent. For `SGraph`, though, it would be modeled as three siblings: two `SNodes` and an `SEdge`.

sprotty's concept of the `SGraph` view model is that each model element with a conceptually different representation in the rendering should extend one of the `SGraph` model classes and

define a unique `type` for that element. This way, the visualization framework should define a rendering just based on the specific class and the `type` field of the element. Looking at the sprotty example diagram from Figure 2.4b there are different node types—the circular task node type and the rectangular barrier node type—that extend the `SNode` with different `types`. The viewer then is programmed to generate SVG circles with a fixed size from the task nodes and SVG rectangles with sizes given from the server layout. Cascading Style Sheets (CSS) are then used to add an easy to configure styling to each element to define their color, border line width, text fonts, and so on.

The `SGraph` model has a diagram layout connector in ELK for generating a layout as well. The layout is also stored directly in the graph elements. However, the micro layout and rendering is handled in a different way than it is in `KGraph`s. There are no rendering information attached to estimate sizes of nodes. That is because the classes `SEdge`, `SNode`, `SPort`, and `SLabel` are supposed to be extended for each use case. The extensions then contain additional fields to provide the visualization with additional data used only for that kind of diagram. The `type` string in each element tells the visualization which rendering to use. Prerendering each element before the macro layout is the micro layout here and provides ELK with the bounds it needs.

So for sprotty models the SVG view has to be programmed afresh for each use case in addition to an extension to the `SGraph` structure, whereas for `KGraph` models only a new synthesis to `KGraph` has to be implemented with the rendering information attached. However, the sprotty model and the serialization is implemented and working in both the Java and the TypeScript world, so that working with that model would be a smaller hassle in that sense. That means that both view models require a different approach when used.

3.3.1 Diagrams 100 % Sprotty Style

Now let us assume that we ignore the `KGraph` view model and only use the `SGraph` model for visualizing an SCCharts model. Since SCCharts is a language written with Xtext, there already is a language server that sprotty can build on and the domain model gets generated on the server. For the view model, the `SGraph` model needs to be extended for each possible type of state and transition, for regions and comments, and not least for all options applicable to and actions executable on SCCharts. So the handling of said options and actions, which is already implemented for the `KGraph` model by KLighD, has to be reimplemented and added to the `SGraph` model so that the viewing framework can consider them.

Next, a diagram generator needs to be implemented to translate the domain model to the new defined extension of the `SGraph` model. Because the SCCharts language is quite complex and is still actively being extended, this translation would be a big part and would need to be updated every time something changes in the language, as already happens with the synthesis from SCCharts to `KGraph`. In fact, this translation would look quite similar to the `KGraph` synthesis, except that no explicit rendering information would be attached, but hints for the viewing framework would get added to the sprotty model.

As the next step a micro layout mainly determining the width and height of each view

model element with dimensions, just like the micro layout by KLighD calculates for KGraphs, would need to be implemented. Because of the problem with text sizes, as mentioned earlier in this chapter, the micro layout cannot be calculated on the server alone, since it can only guess sizes of texts the rendering framework would use. So a micro layout with either one or two parts would need to be implemented. With only one part, one would need to implement sending the whole model to the client to estimate all view model element sizes. With two parts, an option is to estimate the text sizes on the client with all texts contained in the model and the micro layout for the remaining elements on the server, which also needs to be implemented first. The single part approach in this case seems more convenient, because splitting the single task of the micro layout on two different machines seems unnecessary and because the viewing framework should know how to render the model and therefore know the resulting dimensions of the renderings anyway. That is also the approach suggested by sprotty: to have the client render every element in a hidden view and tell the server the sizes of every model element. That comes with the cost of a bigger network footprint though, because the whole model would be sent instead of just the collection of used texts. Then the macro layout by ELK can be applied, which is conveniently already implemented for SGraph models.

After that the whole model would need to be sent again to the client where a specific implementation of the SVG rendering of every part of the SGraph model of SCCharts would need to be programmed. So in short, using only the SGraph view model will require programming

- a new SGraph extension on the server and the client,
- a diagram generator to translate the domain model to that new extension,
- a new micro layout,
- a translation to SVG,
- and handling of options and actions on the server and the client.

And with that only one new language is supported and the process has to be repeated for every language known to KLighD that should also be visualized with sprotty. So reusing as much of KLighD as possible will make the implementation work easier and simplify the maintenance required to keep sprotty views and interactions updated relative to the implementation in KIELER.

3.3.2 Diagrams 100 % KLighD Style

Ignoring sprotty's view model SGraph completely will not really work, because the interfaces for communication between client and server and sprotty's viewing framework expect SGraph models and will not work without them. However, since the view models KGraph and SGraph look so similar and since the SGraph model is meant to be extended, elevating the concept of KGraphs to the SGraph model is the closest one can get to reuse the KGraph and KLighD's functionality with sprotty.

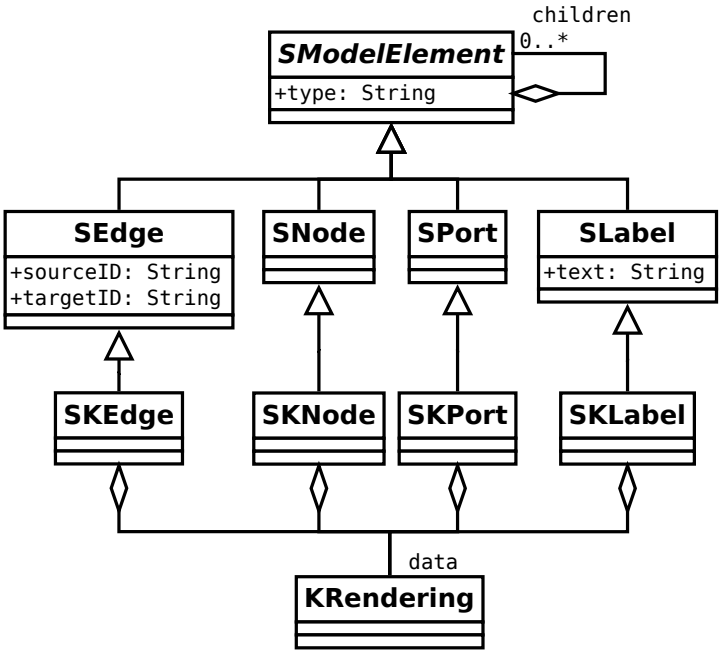


Figure 3.6. An extension on the `SGraph` model to represent the `KGraph` called `SKGraph`.

Figure 3.6 illustrates a simple extension to the `SGraph` model to contain `KGraph`s with the rendering information of the `KRendering` still included in the model, such that the rendering framework of sprotty is able to draw each element as it was intended by the `KGraph` diagram synthesis. The node, edge, port, and label classes of KLighD are just translated to subclasses of the node, edge, port, and label classes of sprotty, called `SKNode` for the sprotty version of a `KNode`, `SKEdge` for the sprotty version of a `KEdge`, and so on. If the `KGraph` model is now translated to this `SKGraph` model before transmission to the client, sprotty can work with it.

Regardless of for which language a diagram should be rendered, then, as long as KLighD knows a synthesis from the domain model of that language to `KGraph`, the diagram generation should work. Let us still assume we want to visualize an SCCharts model using almost only the `KGraph` view model. Xtext again already generates the language server that sprotty can work with and generates the domain model on the server. The `KGraph` view model is now easily provided by the already existing synthesis and is enriched by the `KRendering` data that describes exactly how the SCChart looks like in KIELER as well.

The micro layout here also first needs communication with the client for at least the calculation of text sizes. One option is that only the text sizes are calculated on the client first and the rest of the micro layout is then handled on the server by KLighD using the already calculated sizes of all texts. The other option is that the entire micro layout is calculated on the client. However, to send the entire model over, it first needs to be translated to the `SKGraph` model described in Figure 3.6. A diagram generator needs to be implemented to transform the `KGraph` model into an `SKGraph` model. That should be a relatively easy transformation, since only the four graph elements need to be translated from one to the other kind while

keeping the tree structure of the parent-child associations intact. Some more attention has to be given to the edges, because the outgoing edges of a node in the SGraph model are not contained in that node, as they are in the KGraph model, but are siblings within their mutual containing node. This SKGraph can then be sent to the server for calculating the micro layout for each element. Here the first option seems easier, because the server already knows how to do the micro layout for KGraphs. Whether the macro layout by ELK then is executed on the KGraph or the SKGraph does not really matter, because both are already supported by ELK. The SKGraph needs to be generated anyway before transferring the model to the server and a mapping between the elements of the KGraph and the SKGraph needs to exist to apply the layout results and later handle any options changed and actions requested by the client.

Then a translation from the SKGraph or, more specifically, the contained KRenderings to SVG needs to be implemented on the client. Since the micro layout with the positions and sizes of all KRenderings is already calculated and since the KRenderings describe shapes that map to SVG nicely, this is just a translation task with an easy mapping for each element. Also, the styling of diagrams in this case is not implemented in CSS, as it is usual for sprotty views, but is directly mapped from the KStyles contained in every KRendering to SVG instead.

Using this approach will require programming

- the SKGraph extension on the server and client,
- a simple diagram generator to translate from KGraph to SKGraph,
- a translation from SKGraph to SVG,
- and handling of options and actions on the client only.

The last point only really needs to be implemented on the client, because the handling of actions and options is already implemented on the server by KLighD for the KGraph model. Furthermore, with this approach not only SCCharts, but every Xtext language that KLighD knows a diagram synthesis for can also be viewed in sprotty with this concept without needing to reimplement anything. This is the generic approach that the KGraph view model aims at to have a single view model to render all kinds of different diagrams with the help of KLighD.

Of these two approaches (do the diagram visualization in sprotty only with specific SGraphs or with the generic SKGraphs based on KGraph), the second one seems to be easier to implement and be more generic for the use of arbitrary languages and will therefore be used.

3.4 Communication

The messages between sprotty's server and client parts are sent over the language server via JSON-RPC with the diagram/accept message defined as a new message by sprotty. sprotty defines the use cases of where to calculate the micro and macro layout of all elements in

their wiki.[19] sprotty distinguishes between three ways in which a layout for diagrams can be generated:

- on the server only,
- on the client only,
- or on the server and the client.

Layout on the server only means that the micro and macro layouts have to be computed on the server. This setup was first planned for `KGraph` diagrams. KLighD is already capable of calculating the micro layout for any given rendering, so no additional communication with the client was thought to be needed. Also, ELK is able to work with the macro layout of `KGraph`s on the server as well. The layout can almost be calculated on the server alone and reuse all parts of KLighD. However, just because of a single part of the rendering (the texts), this is not possible as KLighD does not know how other machines and programs render texts.

Layout on the client only is also supported by sprotty, but would require adapting to the JavaScript version of ELK, *elkjs*.[20] elkjs is in the open source part of KIELER called *OpenKieler* and works very similar to the ELK implementation running in Java on the server. Furthermore, doing the layout on the client alone would require an additional implementation of the micro layout of `KGraph`'s `KRendering`s and would contradict the concept that the steps with higher computational cost, such as the layout, should rather be outsourced to the server.

For that reason the layout split up onto the server and client is the way this project will take. Yet the defined steps by sprotty are still not quite what we need. Looking at the sequence of messages needed to generate a diagram defined by sprotty's client-server protocol makes this a little clearer.

In the following, the letters S and C represent the *server* and the *client*, respectively, and $S \rightarrow C$ or $C \rightarrow S$ indicates that the server sends a message to the client and *vice versa*:

1. $C \rightarrow S$ `RequestModelAction`
2. $S \rightarrow C$ `RequestBoundsAction`
3. $C \rightarrow S$ `ComputedBoundsAction`
4. $S \rightarrow C$ `SetModelAction` or `UpdateModelAction`

The first message is sent by the client once the user requests a diagram for a text source opened in Theia and contains only the URI identifying the same text source on the server and the type of diagram requested, so the correct sprotty diagram server is addressed.

The server will then create the `SGraph` view model and will issue the micro layout request via the `RequestBoundsAction`. In this action, the server sends the complete view model and expects to receive back an answer from the client in the form of a `ComputedBoundsAction`. The

[19] https://github.com/theia-ide/sprotty/wiki/Client-Server-Protocol

[20] https://github.com/OpenKieler/elkjs

client then renders the complete model in sprotty's hidden renderer and sends that action back with new calculated bounds and alignments.

The bounds and alignments reference the elements they belong to via the IDs, which got associated with the elements during the `SGraph` model generation. The alignments are only used when the complete layout is calculated by the client alone. In that case the alignments are used to position the elements relative to the origin point while drawing. As a response to the server, this is usually not needed though, but because both concepts use the same message interface it is included here.

In a last step the server computes the macro layout with ELK respecting the bounds just given to the elements and sends the complete model with positions and bounds back to the client via the `SetModelAction` if the diagram is drawn a first time or via the `UpdateModelAction` for every subsequent requested diagram for the same source model.

This is not the exact message protocol applicable to `SKGraph`s. Because the complete micro layout on the client is not necessary, as KLighD takes its own part in that and because the macro layout should not be computed on the `SKGraph`, but on the `KGraph`, the `RequestBoundsAction` and the `ComputedBoundsAction` need to be modified.

My proposed new message protocol is the following:

1. $C \rightarrow S$ `RequestModelAction`
2. $S \rightarrow C$ `RequestTextBoundsAction`
3. $C \rightarrow S$ `ComputedTextBoundsAction`
4. $S \rightarrow C$ `SetModelAction` or `UpdateModelAction`

This is similar to sprotty's original message protocol and the `RequestTextBoundsAction` and `ComputedTextBoundsAction` are named so similar to sprotty's original messages, because they have the same purpose, but are handled in a new way to fit the diagram generation and layout with KLighD.

After the `RequestModelAction` requests the server to generate a diagram, not the complete `SKGraph` view model is sent back to the client for micro layout, but only a special smaller `SKGraph` just containing elements for every text together with its rendering and styles. Each element in that smaller graph just has the text as its rendering with the `KStyles` attached to the text rendering from the original graph. The styles are CSS-like stylings such as the text size and font which all will affect the size of the rendered text on the client. That way, the `RequestTextBoundsAction` can also call the hidden rendering on the client and render all texts, without needing to implement views for the texts drawn inside sprotty's hidden rendering and while keeping the message much smaller. In that smaller graph containing only texts, each text is held in its own element with its own ID. Because sprotty reads the size of each element from the hidden renderer and because each rendering can contain multiple texts, it is necessary to strip each text from the original elements to estimate the size of each text individually. That means a single width and height per element from the main `SKGraph` model would not suffice and the default sprotty style micro layout messages cannot be used here.

However, the client will not send back the bounds and alignments, but just the bounds of all elements (and therefore all texts) in the `ComputedTextBoundsAction`. Because the message is different from sprotty's micro layout, it does not need to follow the interface to also include alignments. A system with only client layout will never need the `ComputedTextBoundsAction`, because the client alone does not face the problem of needing to message a different machine that knows the bounds.

When the server now receives the `ComputedTextBoundsAction` containing the text IDs and the bounds, it maps those bounds back to the texts from the original `KGraph` model so the remaining micro layout and the macro layout can be calculated on that model.

The last message in the protocol however stays the same. It sends the whole `SKGraph` with positions and sizes over to the client for rendering.

Some new messages for Steps 9a and 9b from the overall architecture in Figure 3.2 need to be defined in addition to the exchange of messages for generating diagrams. These messages are sent when the user interacts with the model and causes actions from the diagram synthesis to be performed, or when the user changes options that influence the way the diagram should be drawn. Both actions and options are defined in the diagram synthesis, but their behavior is not sent to the client with the view model. Since the options and actions can be arbitrary and defined specifically for a single DSL, they cannot be executed on the client alone. An ID pointing towards the action that should be performed and the activation condition is attached to every rendering in the view model. The implementation of the action and how it should modify the view model exists only on the server, because KLighD was only ever used on the JVM. Thinking about the decision that most processing for the rendering should be done on the server, and because all action implementations should not be reimplemented on the client, if avoidable, the actions and the corresponding change of the view model should be executed on the server. This thought also applies to the options. They also should change the model on the server, which should be resent to the client to update the rendering there. For actions, a new message containing the ID of the action and the rendering it originates from needs to be sent to the server. This could be sent as a sprotty action named `PerformActionAction`. Note that this is a sprotty action containing an action from KLighD's diagram synthesis, hence the awkward name. After receiving this message the server would perform that KLighD action on the original `KGraph` model and restart the diagram generation process on some different entry point, eventually sending an `UpdateModelAction` back to the client. This handling of KLighD actions is not included in the implementation for this thesis, so it is a part for future work.

What is already included is a similar handling of the options defined by the `KGraph` diagram synthesis. The options are not attached to any part of the view model itself, so they are not sent to the client through the default diagram generation process. Therefore the client needs to ask the server for the available options, so it can display them to the user. So when the initial `RequestModelAction` is sent from the client to the server, an additional message named `keith/getOptions` needs to be sent over the LSP. That message contains an argument with two fields:

uri — The URI pointing towards the resource for which the available options should be sent back. This should be the URI of the text document's resource, for which the `RequestModelAction` is also issued.

waitForDiagram — A boolean indicating if the `RequestModelAction` was issued at the same time. If that is the case, the diagram options may not be available yet and the response message would not contain any options. Another request would have to be sent again later. To avoid that, this option causes the response to be delayed until the diagram generation can provide this request with diagram options.

The response of the server to the `keith/getOptions` message contains a list of all options generated by KLighD's diagram synthesis and is sent back to the client.

This message is separated from the diagram generation because the diagram options view is also separated from the main diagram view. That way, both views have their own messages and stay independent of each other with respect to the messages. Furthermore, only when the options view is opened in the browser, the data for that view is requested by the browser. For more details on the diagram options view, see Section 3.5.

When the user now changes the options on the client, another new message named `keith/setOptions` is sent over the LSP. That message also contains an argument with two fields:

uri — The URI pointing towards the resource the new options should be applied to. This is the same URI as in the `keith/getOptions` message.

synthesisOptions — A list of all options that should be changed for the next diagram generation of the referenced resource, along with their new values.

When receiving this message, the options should be changed on the server and considered during the generation of the next updated diagram. Additionally, the server responds to the client with an "OK" if applying the options was successful or an "ERR" otherwise. Also, the server should restart the diagram generation with the Steps 9a or 9b from Figure 3.2, depending on the kinds of options changed in the message. That can be differentiated because the diagram options are split up into two different kinds of options with different effects, as explained in Section 3.1.

Because all messages are sent via JSON-RPC in the LSP, the objects sent in every message need to be serialized to JSON before transmission and deserialized on the client. When sending for example a `SetModelAction`, the `SKGraph` instance in that action needs to be translated to a JSON string containing a description of the entire object structure of that instance. When that JSON string arrives at the client it then needs to be translated back into an object structure again. The server using the JVM uses the Gson framework translating Java objects to and from JSON and the client using JavaScript uses the deserialization already built into its core. However, the `SKGraph` model cannot be serialized by Gson correctly by default, because the contained `KRenderings` have circular object references inherited from the EMF `EObject` model

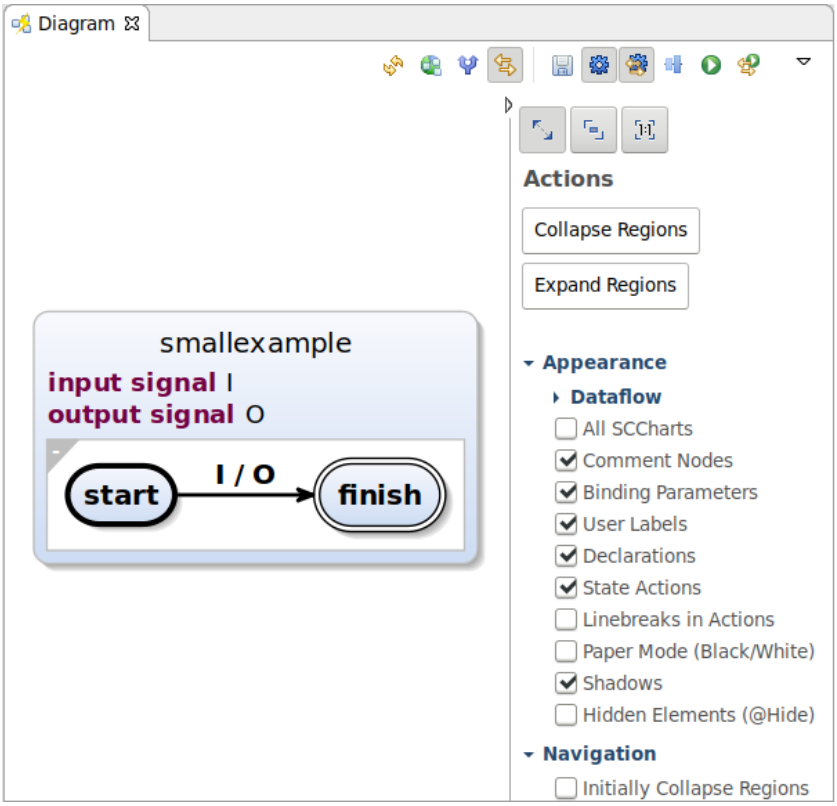


Figure 3.7. The diagram view and the diagram options view in KIELER.

it is structured in. Those circular references and some more problems arise, which are further discussed in Section 4.1.3.

3.5 Diagram Options View

The options to alter the look of drawn diagrams in KEITH have to be displayed and made interactive to be usable and to round off the concept of drawing diagrams in Theia. I have thought about some concepts on where to put a view containing the options. It should fit the overall design of Theia and have the same basic functionality as the diagram options view in KIELER. Also, a link between the diagram options view and the diagram view should exist, so the user can easily identify both parts as belonging together.

The KIELER implementation of the diagram view with the attached options can be seen in Figure 3.7. It only partially follows the basic UI design in the editors-views-perspectives paradigm for Eclipse [Rub06]. It provides specific functionality via a view that integrates into Eclipse via its API. However, actions such as synchronizing the view with the editor, saving the drawn model, and simulating the modeled state machine are in a separate toolbar instead

of the main action bar in Eclipse and the diagram options are an expandable sidebar within the diagram view, which is different from Eclipse's main UI design.

Reimplementing this style in Theia is a first option to design the views. Redesigning the diagram view from scratch in Theia is a lot of additional work though, because sprotty already defines that view with functionality to display SVG diagrams and communicate with a server for diagram generation. Having a view with a sidebar to contain a view on the diagram options would need a change in sprotty's internal implementation of its widget, if a split part should always stay on the right and be rendered independent of the view model sent to the sprotty client, or a completely new implementation of a view with all the client side features of sprotty. Furthermore, compound views do not fit the design of Theia and break the uniform look. Also, adding the buttons as a separate toolbar is not one of the design choices of Theia. In fact, toolbars and buttons to interact with the views are not found anywhere in the UI documentation of VSCode [Mic18d]. Instead, VSCode and Theia use a more textual approach in using a command palette that shows keyboard shortcuts for commands and provides an intelligent search feature. It chooses to have a tidy and clean look without buttons everywhere that most users may not even know what to do with. To make this concept work in Theia the extra buttons have to be removed and the two parts of the view have to be separated into two views.

Another concept inspired by a view in the GNU Image Manipulation Program (GIMP)[21] is to take a multi-window approach, where the main editing view, the toolbar, tool options, and more are opened and displayed in their own windows, which can be resized and positioned freely on the screen and even overlap each other. This level of freedom can be annoying to set up, though, and it clutters the screen with more window borders, close buttons, and so on. Also, this approach is not really usable in Theia, because Theia itself runs in a single window in a web browser and supports views in so-called *widgets*.

To keep Theia's UI design in mind when conceptualizing the diagram options view these widgets should be used. Theia uses PhosphorJS[22] widgets for its views. They can also be dragged around and snap to other groups of widgets. PhosphorJS widgets offer some default placement options for the diagram options view.

The *area* is one of the more important placement options to decide and its options are shown in Figure 3.8. It can be the *main* area, where usually text editors reside and sprotty's diagram view has its default set to. A drawback here would be that the options are open all the time in the biggest part of the available space until manually closed, from where it would need to be reopened via a command or the task bar again. It would not be able to be collapsed and expanded quickly.

The other available areas *bottom*, *left*, and *right* are much smaller by default and cannot be split up by other widgets. They can be collapsed and expanded with a single click, though, and do not need to be closed and reopened just to be hidden for a short moment.

Within the main area the *insert mode* option is also important and defines where a newly

[21] https://www.gimp.org/

[22] http://phosphorjs.github.io/

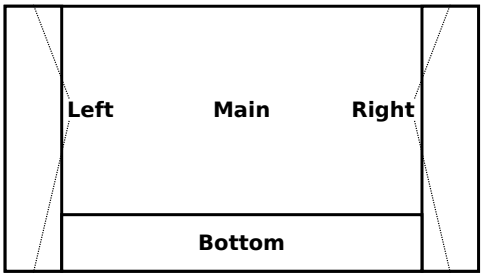


Figure 3.8. The placement options for the area available in PhosphorJS widgets.

opened widget is placed. sprotty's diagram widget for example is placed splitting the widget with the text document it belongs to in half and placing the diagram in the right half. As the diagram options belong to the diagram, this could be done as well here. The drawback though is that the diagram view would be split in half again, leaving only a quarter of the main area horizontally for the diagram and the diagram options. For the options this could work very well, as in KIELER they are just different options listed on top of each other, so the large vertical space works well for that view. But diagrams are only rarely in an aspect ratio such that a long vertical strip at a quarter of the main area's size fits well.

Another insert mode is that new widgets are opened in a new tab before or behind some tab in that space, but because the options actively change the diagram and both should be visible at the same time, this is not an option for the diagram options view.

Because a long vertical view is a viable option for the diagram options view, the left or right panels fit very well. To maintain a reading direction, such that the text model is on the left, the diagram belonging to that model to the right of that, and the diagram options again to the right of the diagram, the right panel is the best choice for it to be in. That also matches the order of these views, as they are displayed in KIELER.

For a consistent order of widgets docked to the right panel, a *rank* option can also be defined, so that regardless of when the widget is opened, it is always on the same side relative to the tabs of other widgets on the right panel.

For the content of the widget PhosphorJS, does not provide a graphics library as Eclipse does with SWT and *JFace*, but relies on typical web content with HTML, CSS, and so on. That enables the view to display the options in almost any way possible on websites. However, the content that should be displayed in the diagram options view is customizable and contains arbitrary data generated by the diagram synthesis on the server. So the different possible structures need to have a visualization fitting to Theia's UI. More details on that implementation can be seen in Section 4.2.3.

3.6 Rendering on the Client

The rendering of any view model on the client is conceptually mostly done by sprotty already, as messages containing the `SGraph` view models are already handled by sprotty. Only the

rendering from the view model to SVG has to be reimplemented and configured, so that sprotty can actually use them. sprotty provides an API to define SVG renderings from the view model. By default sprotty wants an extension of the `SGraph` model, an SVG view written with JSX for that model, and a configuration that tells sprotty about these renderings for the given model.[23] As explained earlier, the view model used is not a specific extension of the `SGraph` model, where every class defines the view, but the `SKGraph` with attached `KRenderings`, where the renderings define what each element's drawing should look like.

So the view definition needs an additional step: for each element, the attached `KRendering` first needs to be translated to an SVG element, and that then needs to be used as the rendering for the model element. Also the styling of the SVG is different from what sprotty suggests. sprotty suggests using CSS to change the appearance of the different model elements. However, the `SKGraph` model uses the graph primitives of the `KGraph` model, which have no inherent stylings that can be used for every element they represent. The stylings are attached as `KStyles` to each `KRendering` and can be different for each node that the view implementation should show. That means that CSS cannot be attached statically to any SVG element, but rather has to be read dynamically from the `KStyle` of the `KRenderings` and put into the `style` attribute of each SVG element. Details on the translation to SVG can be seen in Section 4.2.1.

After the SVG is generated, sprotty will automatically take it and place it in the DOM inside its widget and the browser will take care of rendering the SVG to the screen. Most commonly used browsers support SVG, so this step is unproblematic in that sense. However, the browsers support SVG differently and the same features defined by SVG may work differently or not at all on some browsers. So to reach full platform independence, every browser that KEITH should support still has to be tested.

[23] `https://github.com/theia-ide/sprotty/wiki/Getting-Started`

Chapter 4

Implementation

This chapter describes technical details regarding the implementation of the concept presented in the previous chapter. The implementation can be described well by following an example call from the client, explaining the way from a diagram request until the diagram is drawn on screen. Because the implementation is split up into a server part and a client part, this chapter explains both parts individually.

4.1 Server Implementation

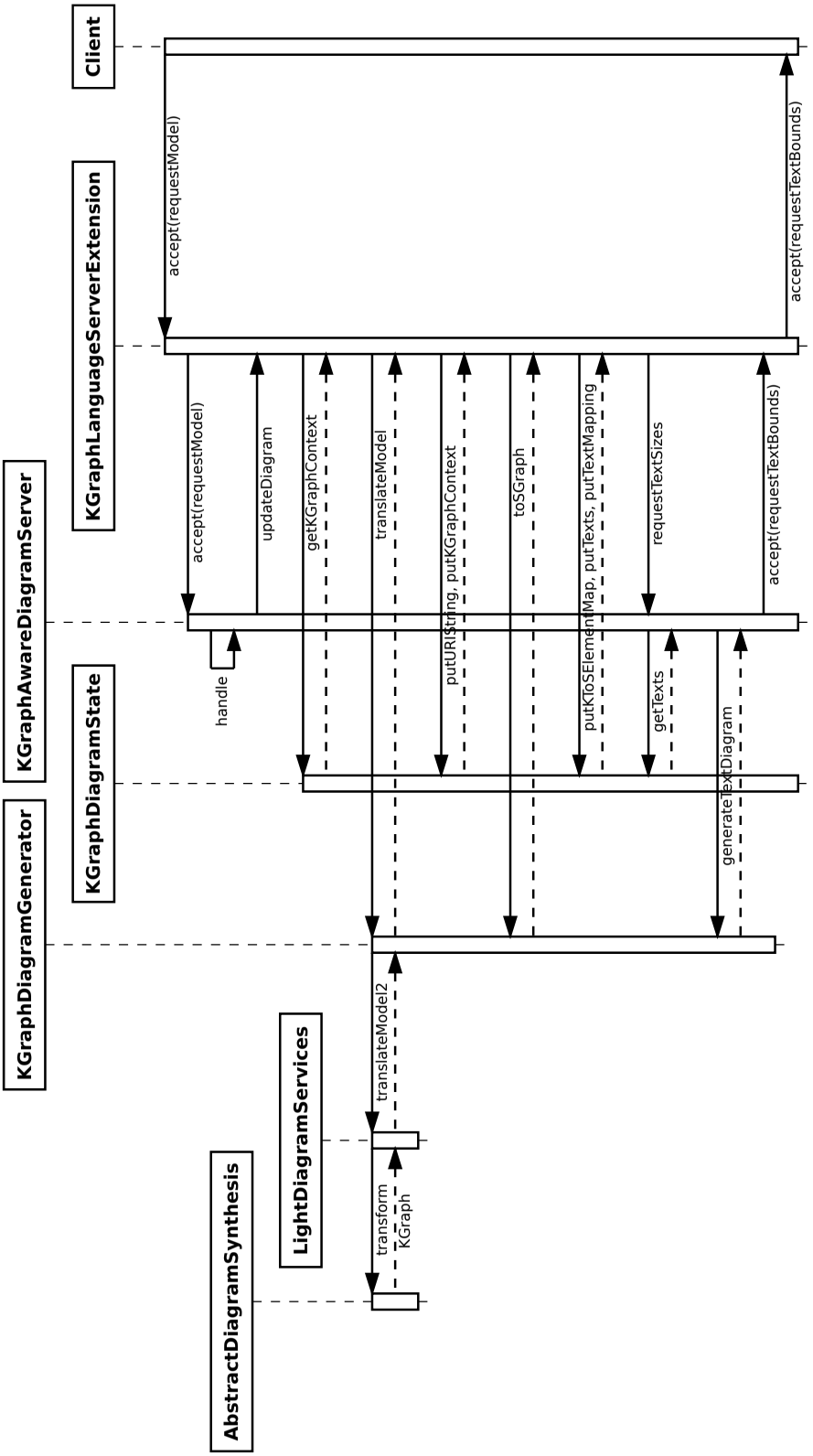
A look at a call requesting a diagram from the server is shown in Figure 4.1. The diagram contains a lot of information, so I will guide you through it in small steps in the following sections.

4.1.1 Classes Used on the Server

A language server following the LSP can be generated for all the DSLs for which diagrams can be created with this concept, because the languages have to be based on an Xtext grammar and Xtext is able to generate a language server for every DSL written in its grammar. Such language servers can be extended with new messages to add language specific messages and behavior to the protocol. Therefore the interface `ILanguageServerExtension` for language server extensions was created.

The sprotty framework then adds a lot of architecture around creating diagrams for Xtext-based language servers with the `DiagramLanguageServerExtension`, which implements the `ILanguageServerExtension` interface. That extension adds the `diagram/accept` method to the protocol to be able to send and receive sprotty actions. The `DiagramLanguageServerExtension` adds the functionality to generate diagram models and update them when their source document changes on the client and it adds synchronization of selected elements between the document and the diagram, handling of a diagram server and generator, and much more.

Because generating diagrams in the style of KLighD with the `KGraph` view model needs some additional features, such as handling the actions and options from KLighD and maintaining a mapping between `KGraph` and `SKGraph` model elements, the `DiagramLanguageServerExtension` gets extended even further with that functionality in the `KGraphLanguageServerExtension`. Along with many other classes used for diagram generation, this class is registered to Guice in a module during the start of the application running the language server. That way, it is used as the language server extension at every part in the code where a language server


Client
KGraphLanguageServerExtension
KGraphAwareDiagramServer
KGraphDiagramState
KGraphDiagramGenerator
LightDiagramServices
AbstractDiagramSynthesis
accept(requestModel)
accept(requestTextBounds)
accept(requestModel)
updateDiagram
getKGraphContext
translateModel
putURIString, putKGraphContext
toSGraph
putKToSElementMap, putTexts, putTextMapping
requestTextSizes
accept(requestTextBounds)
handle
getTexts
generateTextDiagram
translateModel2
transform KGraph

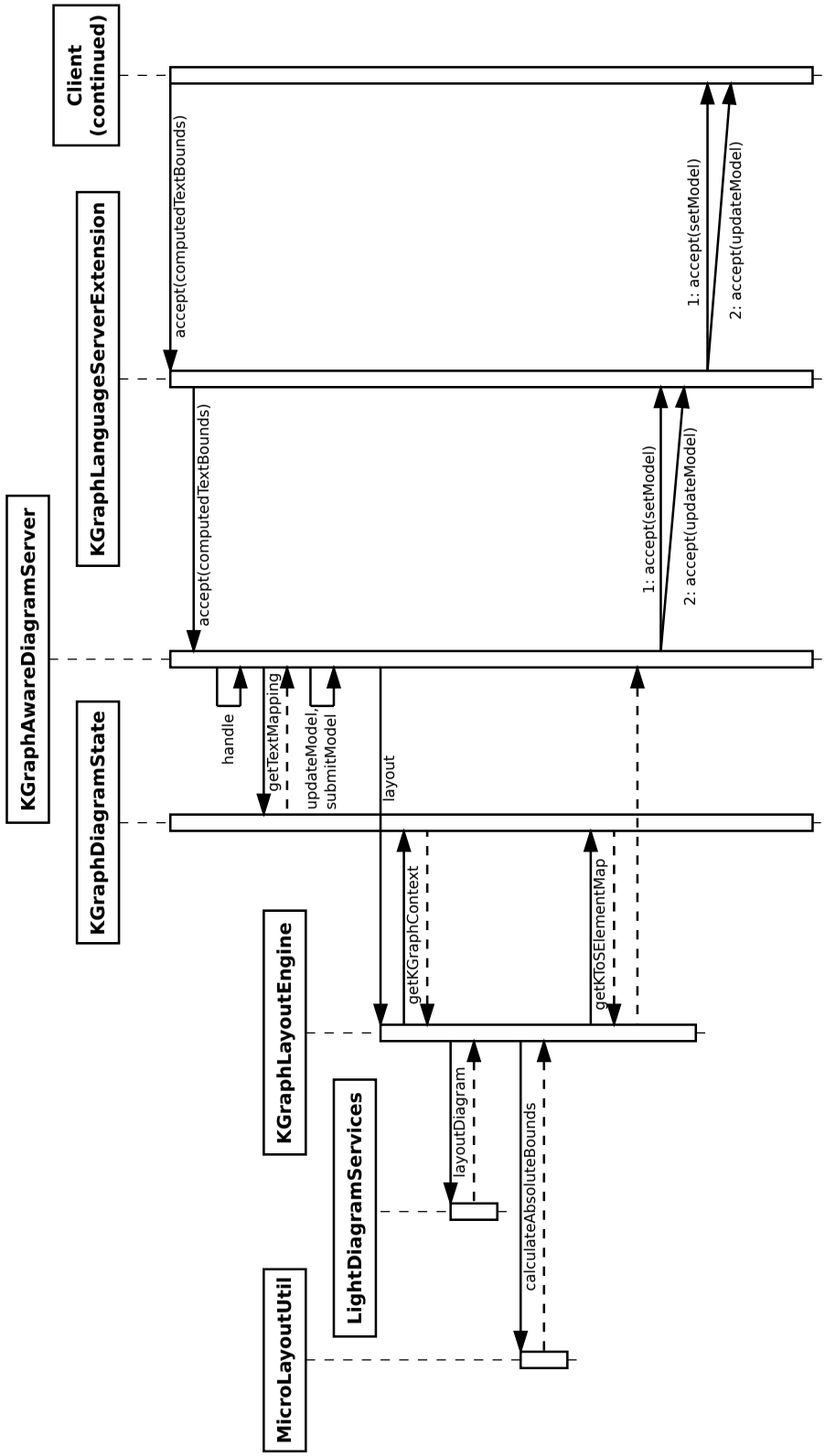


Figure 4.1. Sequence diagram describing every relevant part of the server implementation when the client sends a `requestModel` or `updateModel` action. The two separate diagrams have their lifelines connected between identically named classes to form the complete diagram.

extension is required. This class and the other classes explained next are the main actors in Figure 4.1.

The diagram server follows a similar approach as the language server extension. A diagram server handles the sprotty actions forwarded to the server for a specific diagram view in the client, identified by its client ID. The diagram server also remembers the current model, as it is sent to and displayed by the client. The default implementation of that diagram server, the `DefaultDiagramServer`, is extended by sprotty's `LanguageAwareDiagramServer`, so it knows about Xtext generated language servers and their domain models. That server is further extended by the `KGraphAwareDiagramServer` to also correctly handle the added sprotty actions `RequestTextBoundsAction` and `ComputedTextBoundsAction`. It is also registered to Guice, so that this diagram server implementation can be injected anywhere in this language server.

Apart from the current `SKGraph` model, the server also needs to remember the original `KGraph` model from the synthesis that belongs to the domain model to invoke the KLighD actions on and apply the options to. Furthermore, a convenient way to store the graph containing only all texts for the `RequestTextBoundsAction` and a mapping between that graph and the texts in the original `KGraph` model needs to be found. All these additional models and maps are stored within the singleton instance of the `KGraphDiagramState`, which contains all additional data in maps under the key of the client ID by sprotty. That way, the instance of that class always has the data for all opened diagrams on the client side still stored on the server for a quicker turnaround time on interactions on any of the diagrams.

The next class that is also registered with Guice as the specific class to transform the domain model into the view model is the `KGraphDiagramGenerator`. In this case, it handles the transformation in two steps as it first translates the domain model to the `KGraph` view model which is then translated to the `SKGraph` view model. sprotty usually expects a translation from a specific domain model directly to some `SGraph` model in a class implementing the `IDiagramGenerator` interface, which gets called in the language server extension. However, the first step here enables the solution to be used with all diagram syntheses. The `DiagramLanguageServerExtension` then again knows about this specific view model generator through dependency injection. For this concept, the `KGraphDiagramGenerator` also handles the generation of the separate text graph and provides the `KGraphDiagramState` with the models and mappings it stores.

The last class that extends sprotty behavior is the `KGraphLayoutEngine`. It extends the `ElkLayoutEngine` implemented by sprotty, which provides the diagram layout connector to translate from any `SGraph` model to the graph model of ELK and back and to apply the layout computed by ELK to the `SGraph` model. The behavior of sprotty's `ElkLayoutEngine` is not used as the `SKGraph` model should not be layouted. Instead, the `KGraph` model is layouted, because it has the layout options of KLighD stored in its model and because the `KGraph` model also has a diagram layout connector. Also, the `KGraphLayoutEngine` handles the remaining micro layout for all `KRenderings` and the mapping of the layout data from the layouted `KGraph` model back to the `SKGraph` model for the transfer to the client.

The rest of the classes are utility classes (whose purpose is explained in the next section) or classes from the KLighD framework that are also used for the implementation in KIELER.

4.1.2 Example Request of a Diagram

Following an example call of the client to generate a diagram is a good way to show how all classes work with each other on the server and how they accomplish their tasks. The reader is advised to follow Figure 4.1 along while reading this section.

View Model Generation When a diagram is requested on the client or something on the client causes the diagram to update, a `RequestModelAction` is issued on the client by sprotty. Because sprotty is configured to forward this action to the server, the diagram server on the client side packs the sprotty action into a `diagram/accept` message and sends it to the language server.

The language server extension on the server side then looks into the message to find the diagram server responsible for it with the client ID contained in the message. If no diagram server for that ID has been started yet (which will be the case if the diagram is requested for the first time), a new diagram server is started. The action message is then forwarded to that diagram server to handle it.

For the case of a `RequestModelAction`, the implementation of the `DefaultDiagramServer` then calls the language server extension again to update the diagram model stored on the server. The language server then issues another process to read the source file to get access to the domain model contained in it and returns.

What is done in the other process with access to the domain model is then shown as the next step in the diagram. That is the first point where the behavior deviates from the default implementation of sprotty and where my implementation begins. sprotty would now call the registered diagram generator to generate the `SGraph` view model to be layouted in the next step. Before that, some more steps have to be executed first, though. The domain model has to be translated to KLighD's `KGraph` view model first to get a view model with attached `KRenderings` as in KIELER. If the requested model already had a diagram generated before with changed options, those options have to be applied to the new view model as well. The method `getKGraphContext` is called on the `KGraphDiagramState` to get the `KGraph` view model and, more importantly, the options applied to the model in a previous run to generate a new diagram. With that options, the `translateModel` function of the `KGraphDiagramGenerator` gets called to do the first step of the model translation. The diagram generator calls KLighD's `LightDiagramServices` to translate the generic domain model to the `KGraph` view model. It calls some diagram synthesis extending the `AbstractDiagramSynthesis` registered to KLighD via an extension point in Eclipse to translate any domain model to `KGraph`. This step only works if a synthesis for the requested domain model is registered, otherwise an empty `KGraph` will be returned and ultimately the diagram view on the client will stay empty. This `KGraph` model is then returned to the language server extension for further processing. The new `KGraph` model is again stored in the `KGraphDiagramState` for the next call for a new diagram.

Next, the KGraphDiagramGenerator is called again to return the SKGraph that should later be sent to the client and drawn there. This step has some more important details to be explained next.

Next to the straightforward translation of each KGraph element to an SKGraph element with the representing KRendering of each KGraph element attached to the SKGraph, some more bookkeeping for the client and the server is executed. One part of it is the tracing attached to every rendering. Tracing attaches the URI of an element in the domain model to its corresponding view model elements. This is a concept of sprotty for a better integration of Xtext languages. When the user clicks on any rendered view model object on the client, the client sends a SelectAction with the traced URI to the server. The server then sends the lines of code in the text source of the domain model back to the client, so it can highlight which lines belong to the selected element in the rendering. Using this tracing is an easy way to have some interaction between the text source and the diagram on the client, because the basic behavior is already implemented by sprotty. However, the default concept of sprotty will not work in this case. Because the model being translated by the toSGraph function is not the domain model representing the source code in the client, but the KGraph view model already translated by KLighD, the tracing also needs to take an additional step. So the SKGraph element should not trace back to the KGraph element used in this translation, but to the domain model element the KGraph element originates from. The KGraph synthesis luckily stores that domain model element in a property of the KGraph element, so the tracing uses that object instead.

The KGraphDiagramGenerator also generates a map to link each KGraph element to the SKGraph element that was generated for it. That map will be needed later to map the layout results back from the KGraph model to the SKGraph model.

What is also generated during the translation to the SKGraph is the list of all KTexts and KLabels contained in the KGraph, conveniently stored already as SKLabels to be sent to the client for the text size estimation later. Each KLabel contains a KText rendering, but other elements can have one or multiple KText elements in their rendering as well that all need to be included in the list. So while the KGraphDiagramGenerator already generates the SKGraph elements, it also prepares the SKGraph containing only all texts of the model within SKLabels for the text size estimation on the client. This list of all SKLabels can then be stored in the KGraphDiagramState later.

Another map linking the ID of each SKLabel element to the KText rendering the bounds should be applied to is built to be stored in the KGraphDiagramState. This is to allow the results of the text size estimation on the client to be applied to the original renderings in the KGraph model for the layout step.

The last special requirement is that sprotty needs an ID for every SKGraph element. It is used in the model to allow cross references between elements other than the inherent parent-child relationships induced by the tree structure of the SGraph. That is especially true for SEdges that refer to their source and target SNode or SPort by their IDs. IDs are also used by sprotty on the client to match up elements between separate diagram generations to allow a smooth animation from all elements to their representation in the new graph.

ID Generation The KLighD framework already contains an ID generation for parts of the `KGraph` model, but not for all elements. It also has the problem that different elements could get the same ID assigned which causes errors for sprotty on the client side. Therefore I reimplemented a new ID generator for `KGraph` elements to give every element a unique ID that is also readable by humans and tries to preserve as much from the graph structure as possible.

The IDs for the elements are built hierarchically and always contain the ID of their parent element as the prefix. Also, each level of hierarchy in the ID is separated by a `$` character that never appears elsewhere in the ID. An example: the root `KNode` of each graph always has the ID `$root`. Each child element of that node then starts its ID with the prefix `$root$`. Depending on the class of the child element, each element then adds a single letter to the ID string:

- an `N` if the element is a `KNode`,
- an `E` if the element is a `KEdge`,
- a `P` if the element is a `KPort`,
- and an `L` if the element is a `KLabel`.

After that, the name of the element which has to be unique among its siblings is added to the ID. If it does not have a name, a randomly generated number is added instead after another `$` separator to avoid possible clashes between the random number and the name of any other object.

An possible ID for an element generated by these rules would be `$root$Nn1$$L42` and indicates a `KLabel` with the randomly generated number `42` as the child of a `KNode` with the name `n1` that is itself a child of the root node.

IDs following that scheme are added to each `SKGraph` element during the translation. That concludes all additional data being generated during the `toSGraph` call, the execution then returns to the `KGraphLanguageServerExtension`. As mentioned, the map from `KGraph` elements to `SKGraph` elements, all texts, and the mapping of all text IDs to their `KText` rendering are then stored in the `KGraphDiagramState`.

Now, that the `SKGraph` view model is generated on the server, the first big step from the concept of generating diagrams is already done. The next step now is for the diagram server to store the view model and perform micro and macro layout on it. So the micro layout for the texts now needs to be calculated on the client. To do that, the diagram server takes the stored `SKLabels` from the `KGraphDiagramState` with the `getTexts` call and lets the `KGraphDiagramGenerator` generate the special diagram containing only the texts. It generates a special `SKGraph` with only all these `SKLabels` as its children. When sending that to the client, the implementation to render the elements on the client can be exactly the same as it is implemented there for arbitrary `SKGraphs`. For more details on that implementation, see Section 4.2.

The `KGraphAwareDiagramServer` now takes that `SKGraph` containing only all the texts from the original `KGraph` view model and issues the newly defined `RequestTextBoundsAction`.

Because that action is configured to be executed on the client, it is forwarded to the KGraphLanguageServerExtension, from where it is then sent to the client.

Layout After the client is done and sends back the ComputedTextBoundsAction, that action is handled in a very similar way to how the RequestModelAction is handled. The language server extension receives the sprotty action and immediately forwards it to the diagram server.

The diagram server takes the mapping for all IDs of the SKGraph with only texts to their origin KTexts from the KGraphDiagramState, where the mapping was previously stored. The bounds for each text stored in the sprotty action are then stored in a new property of each KText, so they can be taken into account later during macro layout.

The model is then forwarded through some default sprotty behavior before it is handed over to the KGraphLayoutEngine to receive its layout. As mentioned earlier, the layout engine here does not layout the SKGraph model directly, but does so with the original KGraph model and applies the bounds later to the SKGraph model. So the KGraphLayoutEngine takes the KGraph model from the KGraphDiagramState to perform the macro layout on it. ELK is then configured to use the layout options stored with the KGraph view model and calculate the layout with those options with KLighD functionality. The layout regards the bounds for all texts as stored with the KText elements and internally calculates the remaining micro layout and the macro layout and stores that macro layout on the KGraph model.

However, the micro layout with positions and bounds for all renderings is not stored anywhere to be applied to the SKGraph model. The micro layout needs to be calculated again to be persisted with each KRendering, so that the client does not need to reimplement the micro layout of KLighD but can just rely on bounds and positions for each rendering sent with the model.

So the new MicroLayoutUtil class is called to handle the micro layout and calculate the size of each KRendering. When looking at the KRendering structure, a new problem shows up. The KRendering model portrayed in Figure 4.2 shows this problem. The KRendering describing how an element should be rendered is not necessarily just an object, directly describing its representation, but can also be a KRenderingRef. That KRenderingRef refers to some rendering contained in the KRenderingLibrary that can be found in the data of the root KNode. Multiple elements in the KGraph model can have a KRenderingRef pointing to the same KRendering. They are, for example, used for edge renderings without labels, as the renderings should look the same. That causes the model to be smaller, because multiple elements point towards the same element instead of defining equivalent and redundant renderings. So the calculated size of each rendering cannot simply be attached to every rendering, because the renderings in the KRenderingLibrary are used multiple times and can therefore have different positions and bounds for every reference. This is solved by attaching a map to every KRenderingRef to link the ID of a rendering in the library to its bounds used for this specific reference. That also means that every rendering in the library needs a unique ID to be referenced to after sending it to the client. Since all other renderings also need a unique ID later on the client for rendering, they are all calculated first.

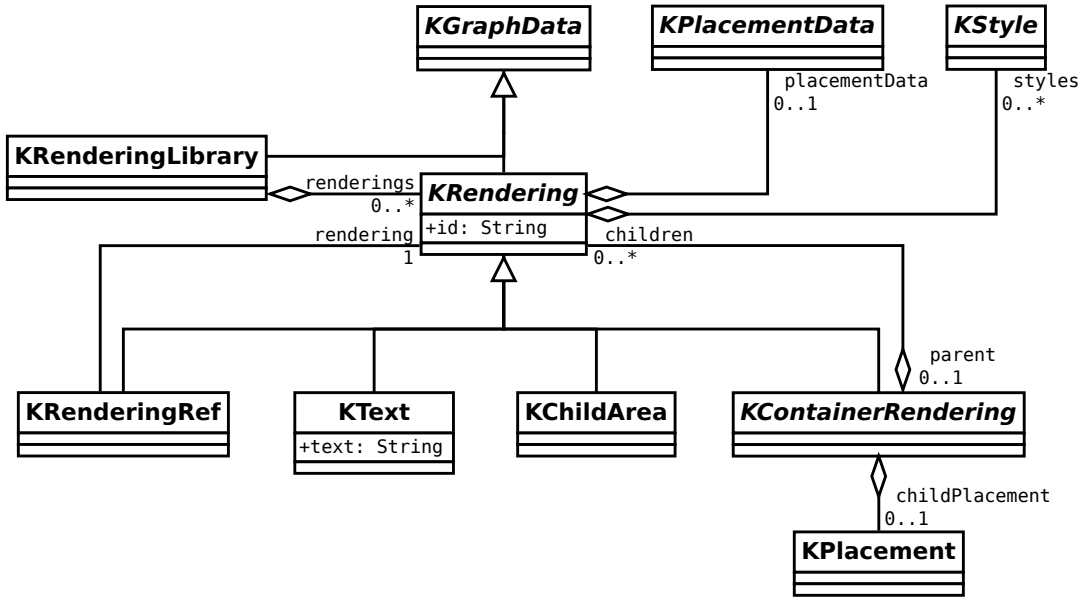


Figure 4.2. The immediate aggregations and generalizations of the `KRendering` class.

All `KRendering`s get their IDs in a similar way to `KGraph` elements. The ID of each rendering either is just the ID it already has (in case of the `KRenderingLibrary`, each top-level rendering already has an ID) or it starts with the prefix `rendering`. Then, each child rendering adds the hierarchy separator `$`, an `R` to indicate a new rendering, and a consecutive number starting at `0`.

There is one special case where one kind of rendering for edges, the `KPolyline`, has an additional rendering for all of its junction points. Those renderings are not indicated with the letter `R`, but with the letter `J`. An example ID would be something such as `rendering$R0$R1$J` for the junction point rendering (`$J`) of the second rendering (`$R1`) of the first rendering (`$R0`) of the element (adding `rendering` in front).

Let us get back to the calculation of the micro layout for each rendering now. Depending on whether the rendering is contained in a `KContainerRendering`, the calculated bounds will be stored either in a map in the `KRenderingRef` or the `KRendering` itself. The bounds of each child rendering of each `KContainerRendering` are then calculated with KLighD's micro layout classes depending on its `KPlacement` and `KPlacementData`. The `KPlacement` can either be a `KGridPlacement` or nothing, which delegates the placement to the `KPlacementData`.

A `KGridPlacement` indicates that all child renderings should be placed on a grid with a given number of columns and a number of rows depending on the number of child renderings to fill those rows. KLighD already has the micro layout for this case implemented, so that is called to calculate the bounds of each rendering which then gets attached to the rendering in a new property.

If the renderings should not be placed on a grid, the `KPlacementData` has information to place every rendering individually. Figure 4.3 shows the possibilities for all placements of

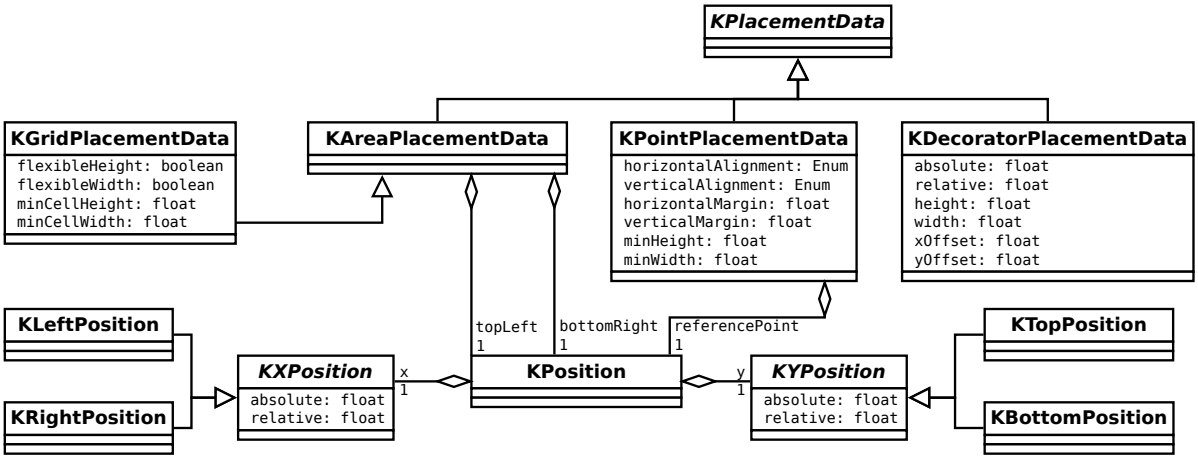


Figure 4.3. The possible placement data to calculate a unique position for each `KRendering`.

the renderings. The `KGridPlacementData` is already handled for each rendering in the case explained above and for the `KAreaPlacementData` and the `KPointPlacementData` the bounds of the rendering can be calculated with a single call to a micro layout function of KLighD. Only the `KDecoratorPlacementData` is a special case that needs to be treated separately. `KDecoratorPlacementData` is used to describe how decorator renderings should be placed. A decorator rendering is always an additional rendering decorating some `KPolyline`, so some rendering describing how an edge should be rendered. A decorator for example is the arrowhead of an arrow. As the look of this type of rendering depends on how the edge is routed, it cannot be placed before the macro layout. Furthermore, the decorator rendering can be set to rotate with the line it is rendered on. Otherwise, each arrowhead would always be rendered pointing towards the right, even if the edge is pointing another direction. That means that the decorator does not store its bounds, but a special data structure called `Decoration` instead. Decorations are stored in the rendering's properties with the same concept as the typical bounds of any other rendering, though, so this is no further issue here. However, the path of the edge as it was routed by ELK needs to be retraced to calculate the `Decoration` with the help of KLighD again.

All these steps for calculating the bounds and decoration are called recursively for each rendering, so that each rendering will have its bounds or decoration stored as a property or in a map in the `KRenderingRef`. This also completes what is done in the `MicroLayoutUtil` class.

As the last steps done on the server shown in Figure 4.1, the layout is then mapped back from each `KGraph` element to the `SKGraph` element generated by it. This is only needed for the elements and not for the rendering, as the `KRendering`s are also directly attached to the `SKGraph` elements, so the bounds and decorations just calculated are already in the `SKGraph` model.

Next, the `layout` call returns to the diagram server which creates a `SetModelAction` or an `UpdateModelAction` to be sent to the client via the language server extension again.

If the diagram was never generated before, the `SetModelAction` will be issued and the `UpdateModelAction` otherwise.

```
{
   "x": {
      "absolute": 0.0,
      "relative": 0.4
   },
   "y": {
      "absolute": 0.0,
      "relative": 0.3
   }
}
```

Listing 4.1. A `KPosition` translated to ambiguous JSON.

```
{
   "x": {
      "type": "KLeftPosition",
      "absolute": 0.0,
      "relative": 0.4
   },
   "y": {
      "type": "KTopPosition",
      "absolute": 0.0,
      "relative": 0.3
   }
}
```

Listing 4.2. A `KPosition` translated to unambiguous JSON.

That is everything that needs to happen on the language server and diagram server on the server side.

4.1.3 Translating the `SKGraph` to JSON

Because the communication between the server and the client happens via JSON-RPC calls, every parameter sent with the messages needs to be translated to JSON first. When using Gson and its default implementation, some problems emerge in the form of incorrect or ambiguous translations.

By default, Gson serializes a Java object by creating a JSON object with all fields of the Java class and their content as key-value pairs. The `KPosition` seen in Figure 4.3 is also used in some rendering elements, such as polygons, to define its corners within the bounds it should be rendered in and should therefore be serialized.

An example `KPosition` translated to JSON then would look as shown in Listing 4.1 if the default Gson serialization was used. The fields `x` and `y` were translated with their fields `absolute` and `relative` put in the child object. But the serialization does not take into account, that the runtime classes can be different and have a different meaning. The `KXPosition` for example can either be a `KLeftPosition` or a `KRightPosition`, defining whether the `x` position should be calculated relative to the left or the right border of its bounding box. The left and right positions also do not define any further fields to make clear, which of the two classes is meant. The same problem also applies to `KYPosition`.

To avoid this problem and to enable the client to distinguish both objects, Gson gets configured to add the class name of these ambiguous cases into the JSON object as if it was another field of the object. The resulting, improved JSON code is presented in Listing 4.2.

The `KRenderings` and `KStyles` are polymorphic as well, so each of their subclasses also get an added `type` attribute during serialization to remove all ambiguities in the `SKGraph` model.

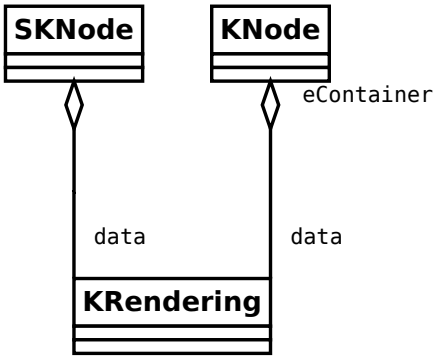


Figure 4.4. Extract of the fields in an `SKNode`.

Another problem arises when translating `KRendering` objects to JSON. Because the `KGraph` model and all its containing parts extend the EMF `EObject` class, every instance inherits fields that are not needed for the visualization on the client. Furthermore, they cause a problem during serialization with Gson.

Figure 4.4 shows a small part of the fields serialized for an `SKNode`. When an `SKNode` gets serialized, the `KRendering` gets then serialized as a child object of that under the key `data`. Because the `KRendering` knows where it was originally created from the EMF model, that `KNode` is stored in the `eContainer` field. That means Gson will also serialize the original `KNode` of that object that is not at all needed on the client anymore when the `SKNode` already contains all required information. To make matters worse, the `KNode` now serialized also contains the `KRendering` in its `data` field. That will cause Gson to serialize the `KNode` before completely serializing the `KRendering` and the `KRendering` before completely serializing the `KNode`. This circular structure will cause Gson to eventually fail the serialization because of a `StackOverflowError`.

Therefore, an exclusion strategy is registered with Gson that excludes all unnecessary fields generated by the EMF model, such as `eContainer`, to avoid this error and to optimize the resulting JSON string.

The last problem with Gson and the EMF model is the handling of their attached properties. Every `KGraph` element and every `KRendering` has some properties attached that are used by KLighD during the micro layout, by ELK during the macro layout, and by this implementation to store the micro layout results for each rendering. Again, not every one of these properties is needed on the client and should be included in the JSON string. Furthermore, these properties can be of arbitrary type and are not just available as a list of all properties to be serialized. Instead, the properties behave like a map where any property can only be extruded with the correct key. So the Gson serialization for these objects is also changed to not serialize the properties field directly, but write additional fields to the JSON string. The additional fields are a `calculatedBounds` and a `calculatedDecoration` field to store the bounds or decoration for any rendering attached to a graph element and a `calculatedBoundsMap` and

a `calculatedDecorationMap` field to store the bounds or decoration for all renderings in a `KRenderingRef`, as explained in Section 4.1.2.

With these changes to the Guice serialization, all messaging needed between the server and the client can be sent correctly, so that the client receives all information it needs for rendering the diagram.

4.1.4 Option Handling

The one thing missing from the implementation details is how the messages to get or set the options of KLighD to configure the `KGraph` diagram synthesis or the layout of ELK are handled.

The first message to get the options for displaying them in the diagram options view on the client is straightforward. The `getOptions` message is handled by the language server extension which then gets the `KGraph` context from the last synthesis stored in the `KGraphDiagramState` and returns the options stored in them. Because this is a message in the request-response style of the LSP, the language server does the rest of the work in serializing the returned options and sending them as a response back to the client.

Because the diagram options can be requested at the same time as the first diagram is requested, the `KGraphDiagramState` may not have any options stored yet to send back to the client. The `waitForDiagram` field in the option parameter described in Section 3.4 can be set to `true` in a request to tell the server to wait for the diagram state to update in that case, so the diagram and its options can be displayed at the same time on the client without the need for an additional request to the server.

The `setOptions` message is also not too complicated in the implementation. The language server extension handles this as well and reads the last `KGraph` from the `KGraphDiagramState` to apply the received new options to. When receiving the list of all new options that should be applied, each of those options in the `KGraph` context is changed to the new value and the diagram generation as shown in Figure 4.1 is restarted after the call to `updateDiagram` in the `KGraphLanguageServerExtension`. Since the `UpdateModelAction` does not have to be a direct response to a `RequestModelAction`, the newly generated and sent diagram will be accepted by the client and rendered on screen. This updates the diagram the same way as it is updated when a change in the text source changes the model that should be drawn. In that case the language server also updates the view model by itself and sends a new diagram which is then rendered on the client.

The new options applied to the view model will be considered during the new diagram generation as explained in Section 4.1.2 and cause the new diagram generated to have the new options be applied as well.

4.2 Client Implementation

The client implementation is mainly based on the implementation of Yangster, which builds on the sprotty framework to get rendered diagrams on the screen in Theia quickly. This section will only go into detail on the parts that differ a little more from that implementation.

At first, all behavior needs to be reconfigured to now accept a language server extension working for the `KGraph` textual language in a way that every language that should be visualizable is registered on the client. Then, the entire `SKGraph` with all possible `KRenderings` and `KStyles` needs to have interfaces, so the server implementation can at least refer to each element of the received model as it is modeled in the `SKGraph`. Next, the views for each `SKGraph` element and all renderings and styles needs to be implemented. That will be explained in Section 4.2.1 with more detail. Also, all the views need to be registered to sprotty, so it knows about them and uses them to render the diagram. And as the last part, the handling of the new messages for the diagram generation and the option handling and the diagram options view implementation is explained in Sections 4.2.2 and 4.2.3. For the remaining implementation I refer to the blog post made for the Yangster implementation by Köhnlein [Köh17a].

4.2.1 View Generation

Usually, the `type` field in `SGraphs` is determining, how the generated SVG should look like. However, as the rendering is handled differently for the `SKGraph`, another approach is used here. Each `KGraph` element calls a generic function `getRendering` that then returns a translated SVG representation of the attached `KRendering` with all `KStyles` evaluated and applied as well. Furthermore, the view generation for the elements handles a few special cases. For example, a default rendering for `SKPorts` is generated, as they do not have default `KRenderings` attached, or hierarchical children of each element are rendered, even though the `KRendering` defines no explicit area to place these children. As the root `KNode` of each `KGraph` is not supposed to be rendered, any rendering of it will be ignored as well and only its children will get rendered. Other than that, most of the SVG generation logic is contained within the `getRendering` function. Figures 4.2 and 4.5 show all the possible `KRenderings` that need to be implemented to reach full support of the possible renderings of KLighD. The following list shows the current support of each of these cases.

`KRenderingLibrary`: This is a special case, as it is not a `KRendering` itself. It can only be contained in the parent `SKNode` of an `SKGraph` and contains all renderings that can be referenced by `KRenderingRefs` for any child elements. Therefore, the `KRenderingLibrary` needs to be stored in an accessible way, such that every `getRendering` call for any child element can look up its rendering if it is a `KRenderingRef`. sprotty provides a rendering context as a parameter in the call for the view generation of every element. That context is usually used to get access to the children of the elements to also render them and other functionality that may be needed for other views generated with sprotty. Here it is extended to the `KGraphRenderingContext` and is used to store the `KRenderingLibrary`, so the library is accessible when needed.

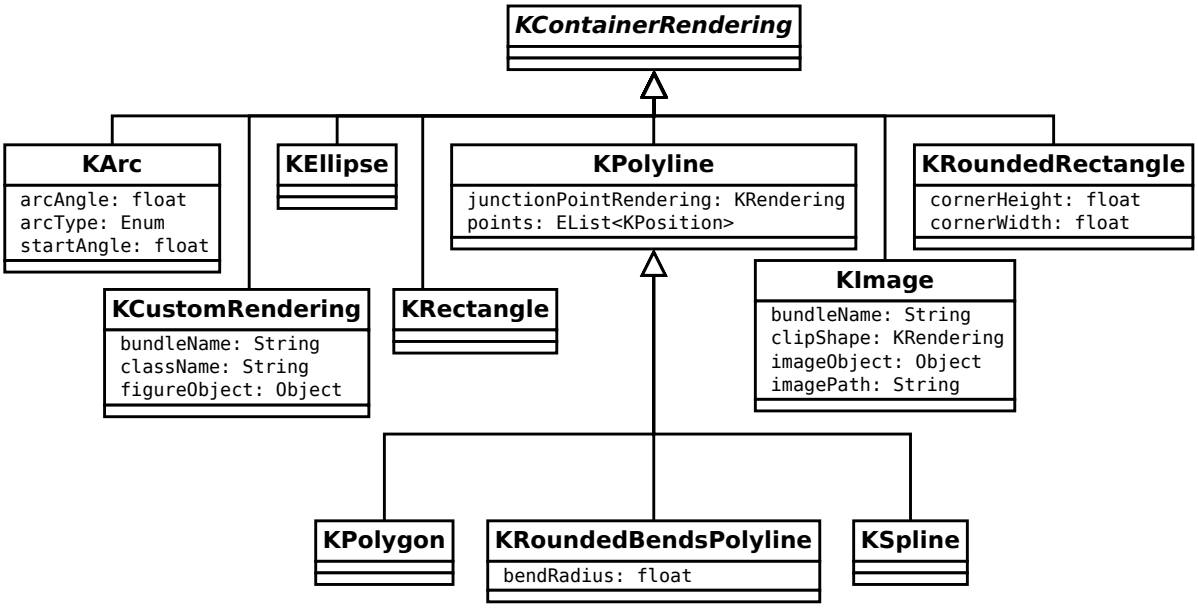


Figure 4.5. Class diagram of all possible `KContainerRendering`s.

```
1       [...]
2       return <text
3          x = {x}
4          y = {y}
5       >{rendering.text}</text>
```

Listing 4.3. Implementation of a `KText` rendering with JSX.

`KRenderingRef`: A `KRenderingRef` contains a reference to its rendering in the `KRenderingLibrary` on the server. However, as the object gets serialized to JSON and deserialized to a JavaScript element on the client, the `KRenderingRef` only contains an ID referring indirectly to its rendering in the `KRenderingLibrary`. So here the implementation looks into the `KGraphRenderingContext` and searches for the rendering in the library, whose ID matches the one in the reference. The rendering found in the library is then handled like any other rendering following in this list.

`KChildArea`: This is the last special case in the renderings. The child area rendering is used to define where the children of the current `SKGraph` element should be placed. Because sprotty already has a function to call the view generation for each child element, this implementation just calls that function of sprotty.

`KText`: A `KText` is an element that has a simple translation to SVG. Ignoring the possible `KStyles` for now, the basic translation is straightforward, as shown in Listing 4.3. The text from the rendering is just put inside as SVG `text` element and the position is added. That

is everything needed to display a basic text element in the sprotty diagram view with the JSX writing of SVG code inside TypeScript.

However, finding the bounds for each element is just a little more tricky. Depending on whether the rendering is a normal rendering, coming from a `KRenderingRef`, or whether the view generation should use the bounds of its parent object makes multiple options where the bounds of any rendering could be stored. Therefore, the bounds for any rendering are first searched in the `rendering.calculatedBounds` field that is added to the JavaScript objects during the serialization from the micro layout properties on the server. If that bounds are not stored, it means that this rendering comes from the rendering library and has its bounds stored in the bounds map added to the parent `KRenderingRef` during serialization. If the bounds cannot be found in there either, a fallback to the bounds of the parent `SKGraph` element itself is implemented.

That is where the `x` and `y` variables come from in this example and in any other rendering described below.

`KContainerRendering`: This rendering is never created directly, only extensions of this class can be rendered. All container renderings can have child renderings inside them, so they need to be rendered as well for each one. So for all child renderings of a `KContainerRendering`, they are generated via the `getRendering` function and put inside their parent SVG element. As all child renderings are positioned relative to their parent rendering, they need to be grouped and transformed via the SVG `g` element and the `transform` attribute of any SVG element.

`KArc`: This rendering should draw some arc with the arguments shown in Figure 4.5. This is not implemented yet.

`KCustomRendering`: This is used in KLighD to call another renderer defined with the `bundleName` and the `className` to draw the `figureObject`. As this requires rendering on the server for arbitrary objects, this needs further discussion how it could be implemented. So this is not implemented yet.

`KEllipse`: This translates directly to the SVG `ellipse` element. The radii in `x` and `y` direction are calculated from the bounds' width and height.

`KRectangle`: The `KRectangle` is using the implementation of the `KRoundedRectangle` with the corner width and height set to `0`.

`KPolyline`: All polylines are rendered using the SVG `path` element. The `path` element has an attribute `d` that needs to be a string defining control commands of a path. These control commands define how each line segment should be drawn. The following commands are used for all types of `KPolylines` and their extensions:

- *`M p`:* The cursor drawing the lines moves to the point `p` without drawing anything.
- *`L p`:* The cursor draws a straight line from where it was to the point `p`.

Z: The cursor draws a straight line from where it was to the first point of the path.

Q c1 p: The cursor draws a quadratic Bézier spline from where it was to the point `p` using `c1` as the control point.

C c1 c2 p: The cursor draws a cubic Bézier spline from where it was to the point `p` using `c1` and `c2` as the control points.

`KPolylines` contain a list of `points` that can be used to define such a control sequence. However, when the `KPolyline` is the parent rendering of an `SKEdge`, the points of the `KPolyline` are not used, but rather the `routingPoints` of the edge itself.

A basic `KPolyline` uses the `M` command for its first routing point and the `L` command for every other routing point.

KPolygon: A polygon is basically a closed polyline. So The `KPolygon` implementation is equal to the implementation of a `KPolyline`, except that a `Z` command is added to the path in the end.

KRoundedBendsPolyline: This is just a `KPolyline` where each corner is replaced with a round corner with a specified radius. However, this is not implemented yet.

KSpline: Splines are a lot more interesting mathematical objects than simple lines and describe smooth curves through an arbitrary number of prescribed points. The cubic Bézier splines also used in the rendering in KIELER use every third routing point as a point the spline passes through and all other routing points as control points for the individual cubic Bézier splines. The same behavior can be achieved with SVG `paths` when using the `M` command for the first routing point and the `C` command for all following point triplets. A default line segment with the command `L` or a quadratic Bézier spline with the command `Q` is also used if there are not three, but one or two remaining routing points at the end, respectively.

KImage: Like the `KCustomRendering`, this also requires additional communication to the server or a different concept to be drawable on the client. The image path could refer to some image on the server that is not yet synchronized with the client. As this needs a new concept, the `KImage` is not implemented yet.

KRoundedRectangle: This translates directly to the SVG `rect` element. The extra parameters `cornerHeight` and `cornerWidth` are also directly translated in the `rect` attributes `rx` and `ry`.

Every rendering can have attached `KStyle` elements that alter the overall look of each rendering. Depending on the rendering and the style, some parts in the SVG generation are changed compared to the default implementation of all renderings explained above, while others need additional SVG elements that are added to the rendering. The following list explains what each of the `KStyles` shown in Figure 4.6 does and which ones are already implemented.

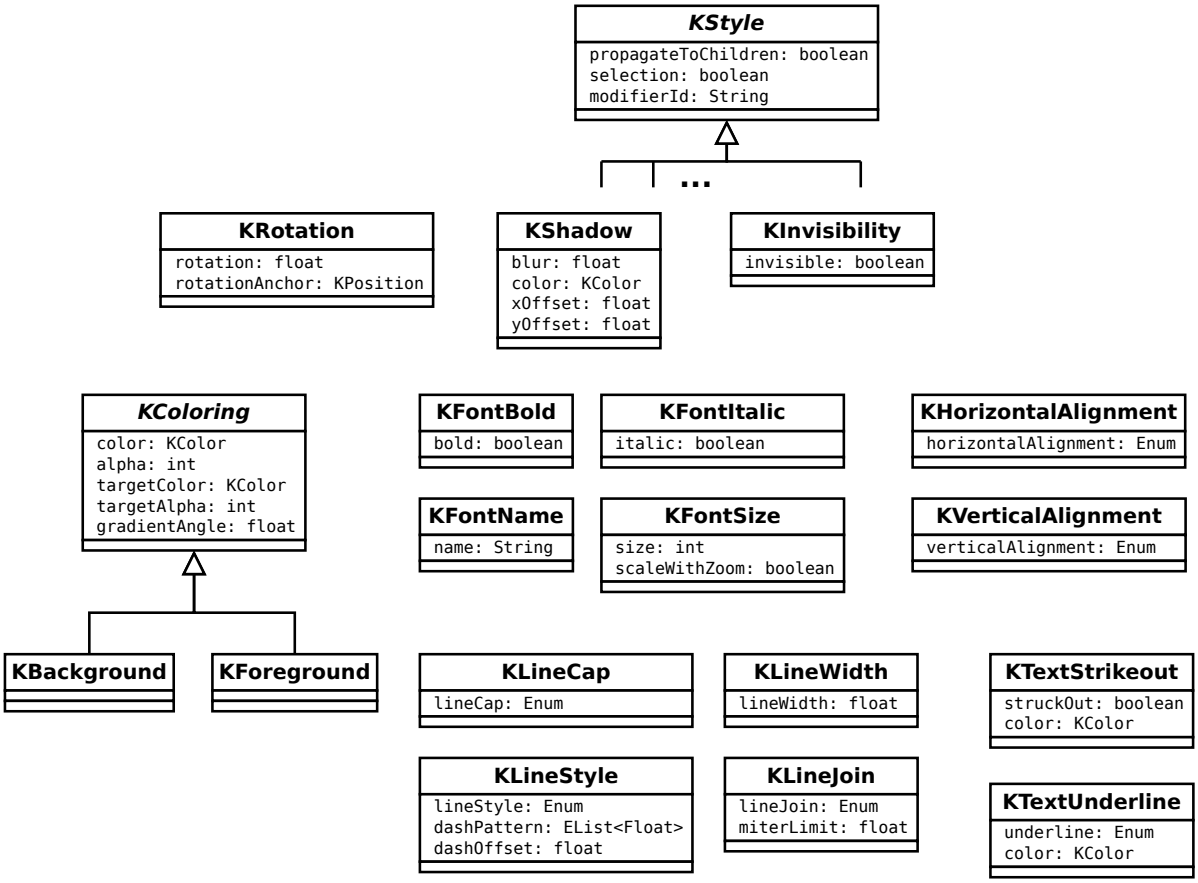


Figure 4.6. Class diagram of all possible `KStyles`. Each class extends the `KStyle` class if not shown otherwise.

`KStyle`: First of all, the generic `KStyle` has some attributes that change the way each style behaves. The property `propagateToChildren` indicates if the `KStyle` should also be applied to the child renderings of the `KRendering` that it is attached to. This is not implemented yet. The `selection` property indicates a style that should only be applied when the rendering was clicked on and is therefore selected. This is also not implemented yet and styles with this modifier are currently ignored.

`KRotation`: The first style is the `KRotation`. It indicates that an entire rendering should be rotated around the anchor point given as a property with the angle also given as a property. This style should not be too complicated in SVG, as SVG supports rotation of any group of renderings, it is not implemented yet, though.

`KShadow`: This is used to add a shadow to any rendered object to make it stand out more. In KIELER the shadow is implemented in a way to drop the entire element that the shadow is for four times to the bottom right of that element. The color defined in the shadow is used for each copy with a decreasing opacity. Because I do not like that kind of shadow

```
function shadowDefinition(style: KShadow, id: string) : VNode {
   return <defs>
      <filter id = {shadowId(id)}>
         <feDropShadow
            dx = {style.xOffset / 4}
            dy = {style.yOffset / 4}
            stdDeviation = {1}
         />
      </filter>
   </defs>
}
function shadowFilter(id: string): string {
   return "url(#" + shadowId(id) + ")"
}
function shadowId(id: string): string {
   return id + "$" + "shadow"
}
```

Listing 4.4. Definition of the filter for `KShadows`.

and I wanted to experiment with the possibilities of SVG, I implemented it in another way. SVG supports filters that can be applied to any SVG element and one of these filters is the `feDropShadow` filter. It renders the same element offset by some configurable amount to the bottom right with a blur that can also be configured. It is always black, as shadows usually only make the background darker opposed to the shadows possible in KLighD, where the shadow color can be defined as well. To make the shadow look similar to the shadows in KIELER with this new concept, I found out that an offset of $\frac{1}{4}$ of the offset given by `xOffset` and `yOffset` with a standard deviation of `1`, ignoring the remaining two properties, gives the nicest shadows still similar to the KIELER implementation.

This filter is added to the SVG rendering in two parts. The first part is defining the filter and including its definitions before the code for the SVG of the `KRendering` and the second part is using the `filter` attribute of the SVG element to refer to that shadow. Both parts need some definitions shown in Listing 4.4. It shows how the definition of the shadow is implemented with the `feDropShadow` filter and how it should be referenced in the `shadowFilter` function. Furthermore, this shows how every rendering needs its own ID that is unique among all renderings when combined with the ID of the `SKGraph` element the rendering belongs to. Multiple renderings can define a shadow whose IDs would otherwise clash in the SVG and create ambiguity on which filter definition should be used.

To recreate the KIELER style shadow, this has to be reimplemented to how it is implemented there.

`KInvisibility`: This style just states, whether the rendering should be rendered invisible. This can be mapped to the SVG attribute `opacity`, where the `opacity` for an element

with the invisibility style should be set to `0`. This is currently only implemented for `KRoundedRectangles`.

`KBackground:` The `KBackground` needs an implementation in two parts, similar to the implementation of `KShadows`. A definition of the coloring is implemented as an own SVG `linearGradient` element. Colors in that element are defined with the `stop` element with the `stop-color` and `stop-opacity` attributes to define the color and alpha value of the style properties. The `gradientAngle` is then defined with the `gradientTransform` attribute of the `linearGradientElement`.

This definition is then included in the SVG and referred to in the `fill` field of `KEllipses`, `KRoundedRectangles`, and `KPolygons`.

`KForeground:` The definition of the coloring is handled the same way as it is for `KBackground`. The only difference is the point where the definition is referred to in the SVG. It is mostly used in the `stroke` attribute (in every `KEllipse`, `KRoundedRectangle`, `KSpline`, `KPolyline`, and `KPolygon`) and for `KText` in the `fill` attribute, because the text color is defined in a `KForeground` style.

`KFontBold:` The following six styles can only be applied to `KText` renderings. This style changes, whether the text should be drawn **bold**. It is defined in SVG via the `font-weight` style attribute of texts.

`KFontItalic:` This style affects, whether the text should be drawn *italic*. It is defined in SVG via the `font-style` style attribute of texts.

`KFontName:` This style changes the font in which the text should be drawn. It is defined in SVG via the `font-family` style attribute of texts. As the font names coming from the server are written in camel case (for example `sansSerif`), whereas SVG expects the name in kebab case (for example `sans-serif`), the font names are translated to the latter first.

`KFontSize:` This style changes the text size measured in points. It is defined in SVG via the `font-size` style attribute of texts. The additional property `scaleWithZoom` is not yet implemented.

`KHorizontalAlignment:` This style changes the horizontal alignment of the text relative to its origin point. It is defined in SVG via the `text-anchor` style attribute of texts. The three possible values for the horizontal alignment are center, left, and right and are translated to the anchors `middle`, `start`, and `end`, respectively. Because the bounding box is still calculated with its origin point at the top left of the text, the `x` value is also adjusted with respect to the width of the text.

`KVerticalAlignment:` This style changes the vertical alignment of the text relative to its origin point. It is defined in SVG via the `alignment-baseline` style attribute of texts. The three possible values for the vertical alignment are center, bottom, and top and are translated to

the baselines `middle`, `baseline`, and `hanging`, respectively. For the same reason as for the `KHorizontalAlignment`, the `y` value is adjusted with respect to the height of the text.

`KLineCap`: The next four styles are very similar again and can all be implemented for the `KRenderings` containing some lines in their rendering. Currently they are not implemented for every of those renderings yet, but implementing them is just applying the same concept to the other renderings. This style defines how the end of a drawn line should look like. It is defined in SVG via the `stroke-linecap` style attribute. The three possible values for the line cap are flat, round, and square and are translated to the SVG line caps `butt`, `round`, and `square`, respectively.

`KLineWidth`: This style defines the width of the drawn lines. It is defined in SVG via the `stroke-width` style attribute. This is just a number, so no further calculation is done there.

`KLineStyle`: This style defines the line pattern of all drawn lines. It is defined in SVG via the `stroke-dasharray` style attribute. There are some predefined styles in the `lineStyle` property that are all translated to some dash pattern. A dash pattern is a series of numbers that defines how long an analogous pen should be changing between drawing on the path and not drawing, but following the path. An example dash array `[3, 2, 1, 2]` would mean that the pen draws for 3 units, shifts over for 2 units, draws again for 1 unit, and finally shifts over for 2 units before starting the process from the beginning until the line is completely drawn. Because each unit is as long as the lines are wide, this resembles a dash-dot-pattern, alternating between a line and a dot for the length of the path, such as this one: -·-·-·-·-·-·- These dash arrays from this style are also supported in SVG in the same form.

`KLineJoin`: This style defines how two lines connected at an angle should draw the corner. It is defined in SVG via the `stroke-linejoin` and `stroke-miterlimit` style attributes. The three possible values for the line join are bevel, miter, and round and are implemented in SVG under the same names.

`KTextStrikeout`: These last two styles only apply to `KTexts` as well. This one changes, whether a text should be struck out and have a line through it (~~struck-out text~~). It is not yet implemented.

`KTextUnderline`: This style changes, whether a text should be underlined. It is also not yet implemented.

With all these `KRenderings` and `KStyles`, a lot of graphs can already be drawn in Theia with sprotty. Implementing the remaining ones is just some work that has to be done at some point later or needs a new concept for the special renderings on the server.

4.2.2 New Messages

There are now four new messages that need to be handled and cause the correct behavior on the client. The two messages for the text size estimation should be sent as sprotty actions

over the LSP protocol. sprotty uses a diagram server on the client that decides, whether any action should be handled locally on the client or sent to the server. That means that the diagram server on the client side needs to be extended as well. sprotty already defined the `TheiaDiagramServer` with all handling of `RequestModelActions`, `SetModelActions`, and so on that are needed for the protocol.

The `KeithDiagramServer` extends the `TheiaDiagramServer` and adds the two new sprotty actions, the `RequestTextBoundsAction` and the `ComputedTextBoundsAction`, to its known actions. The `ComputedTextBoundsAction` is then set to be handled on the server so the client will send this action via the LSP to the server, while the `RequestTextBoundsAction` is set to be handled locally. The `RequestTextBoundsAction` is implemented similar to the implementation of the `RequestBoundsAction` by sprotty to render the model given in the action in sprotty's hidden renderer via the `HiddenTextBoundsUpdater`. That alters the behavior of the `HiddenBoundsUpdater` from sprotty used for handling the `RequestBoundsAction`, except that it only reads the bounds from the hidden DOM and issues a `ComputedTextBoundsAction` instead of a `ComputedBoundsAction`. The behavior is configured to be used by sprotty also via dependency injection, similar to the injected extensions in the server implementation.

The messages for the diagram options are not sent as sprotty actions and therefore need to be handled differently. Because they work closely together with the diagram options view, they are explained in the next section.

4.2.3 Diagram Options View and Messages

After starting the Theia application, the implementation of Yangster adds specific behavior other than features needed for diagram generation or already implemented by the language client of Theia in a new class that is called after Theia is started. The equivalent of that class in this implementation is used to also add the functionality to create and maintain the diagram options view and the new messages `getOptions` and `setOptions` sent to the server.

It creates a new PhosphorJS widget to contain the diagram options. The widget is also set up to update its content when either a new diagram for any document is requested or when the active editor changes. The content of the diagram options widget is then updated in both of these cases.

To update the view and show the available options, a `getOptions` message gets sent to the server with the URI of the currently opened text editor and the property `waitForDiagram` set to true if this message is sent parallel to the request for a new diagram. When the server then returns with the options to display, those options are given to the diagram options view widget, which then updates and renders the options to display in the widget.

The rendering of the diagram options view widget is implemented with JSX to allow easy to implement and interactive HTML code. It is supposed to render a list of diagram options that can have different types. Some HTML code is generated for every option and put inside a `<div>` element. The available option types are:

- `check`,
- `choice`,
- `range`,
- `separator`,
- and `category`.

As a proof of concept of the diagram options view, only the `check` and `choice` option types are implemented to be rendered.

A `check` option in KIELER is a checkbox with a text label that can either be checked or unchecked. That is implemented via an HTML `input` tag with the `type` attribute set to `checkbox` and the name of the option set as the label drawn next to the checkbox. When rendered, this will create a simple checkbox that can be clicked to be checked and unchecked.

A `choice` option is a little more complex as it represents multiple different values of which exactly one can be selected at a time. The HTML tag `input` can also set the `type` attribute to `radio`, which creates a button that can only be checked and gets unchecked automatically if another radio button in the same group gets checked. The implementation is shown in the code sample for explaining JSX in Listing 2.1. This also exactly replicates the behavior of this option in KIELER with a similar look. The look of these options is not finalized yet and can be changed in the future via CSS or just reimplemented using different HTML structures.

Furthermore, each checkbox and radio button has a click event attached to it to send the updated options to the server. The `setOptions` message gets sent to the server for each changed option value by the language client. The answer of the server is currently ignored, but that can be changed to display a message if the server fails or anything else.

Chapter 5

Evaluation

This chapter evaluates how the implementation of the automatic view generation in a web environment compares to the monolithic approach in KIELER. Also, this chapter touches upon the lessons I learned while working on this project and compares my view on this project before starting and now while finalizing it. A user report for developing with Theia and sprotty is furthermore described.

5.1 Comparison Between the Web-Based Approach and KIELER

One obvious change between the KIELER system and the new web-based IDE KEITH is that the environment has moved from an installable program that runs locally on the user's system to the possibility of running in a browser tab or still as a local program that uses a very different concept for UI and user interaction. What I want to compare here, though, is how the view generation performs in both systems, where the new concept introduces new bottlenecks, and how the renderings compare between both systems.

5.1.1 Performance Comparison

An advantage of using a server to do most of the calculation is that it is not able to block the client process during that calculation, so the browser stays responsive. Especially when generating diagrams from big models in KIELER, where document interaction and the automatic diagram generation cause resource intensive processes to be executed, the performance is affected negatively. Some interactions could even freeze the IDE completely if the action is not executed properly in a different thread. When using the language server, this problem cannot happen because the server is always running in a separate process or even on a separate machine altogether. Only when the client receives a diagram from the server and needs to convert that to an SVG for rendering, the responsiveness of the browser could suffer.

However, when using the LSP and the concept of generating the diagram in two different programs, the goal of having a fast and responsive program fails at the point of being fast compared to KIELER. Because the diagram generation uses most steps also used by the diagram generation in KIELER and adds even more steps to the chain, such as the transformation to the `SKGraph` view model and serialization to JSON, this process cannot really get any faster than in KIELER. Surely the amount of work done on the client is much less than having to calculate everything in the same process, because the server takes parts of the process away from the client and leaves only the rendering as its main task to the client. But when it comes to raw

Table 5.1. Time to diagram in KIELER and the new environment.

Model Size (LoC)	KIELER	New Concept	Factor
130	1 s	2 s	2
1 569	9 s	32 s	3.56
35 571	25 s	186 s	7.4

performance, this concept cannot keep up. A monolithic approach is much better at absolute speed on powerful machines, but the client-server approach has a much wider use and may run even on lower-end hardware, such as smartphones.

To back up these hypotheses, I measured the time needed to generate diagrams from the same source model in KIELER and the new implementation for three differently sized SCCharts. Furthermore, the time needed for each step is recorded as well to find possible bottlenecks.

Table 5.1 compares the speed of the implementations. The different models are all SCCharts models, because it is the main DSL considered during implementation and all `KRenderings` it uses are supported. That means that the comparison is fair in the sense that both frameworks draw the same graph with the same renderings. A visual comparison between both renderings is presented later in this section. The models used for the test are chosen with very different model sizes to also test the scalability of both concepts.

The first model with a size of 130 Lines of Code (LoC) is a small example of a motor control system that already uses up a typical desktop monitor worth of space to still be readable when drawn as a diagram. However, since the renderings can be zoomed and browsed in both implementations, bigger models are also feasible to be requested as a diagram. The waiting time is reasonable for both implementations. But that example already hints at a performance slowdown with the new concept.

The next two models were taken from an earlier student project working with SCCharts that I was a part of. Because the second model with 1569 LoC is one of 22 copies in the big model with 35 571 LoC, they both compare well with each other in their modeling and coding style. Both of these bigger models already had diagrams that were too big to be of any particular use during the student project, because every time while programming one would need to zoom in very far into those graphs to see the changes made in the code. Furthermore, the 9 or even 25 seconds of time just to render those diagrams every time a small change was made to the code was already too much and the diagram generation was just turned off while modeling for most of the time. But despite the 22.6 times increase in size, KIELER only took 2.78 times as long to generate that diagram, compared to the new concept taking 5.81 times as long. So both concepts scale better than linear to the LoC with bigger models, but the new concept struggles even more with them.

I chose to use the LoC as a measurement because every line of code adds some visual element in the SCCharts language and the message sizes of the models sent via the LSP scale similar to the LoC of the source model. This is shown in Table 5.2. It shows for all three

Table 5.2. Model sizes sent via network.

	JSON Size		
Model Size (LoC)	**Graph**	**Texts**	**Combined per LoC**
130	0.28 MB	0.07 MB	2.68 kB
1 569	3.61 MB	1.04 MB	2.96 kB
35 571	86.23 MB	25.7 MB	3.15 kB

models that the resulting JSON models of the complete graph with all attached `KRenderings` and `KStyles` and the graph containing only all texts with their styles are around 3 kB in size per LoC for SCCharts. The bigger models have slightly bigger model sizes per LoC after serialization, which is mainly because deeper hierarchy in the models causes the generated IDs included in the JSON representation to be longer. The main problem visible here is the size of the graph models. The biggest of the examples with 35 571 LoC has a 2.03 MB file size, whereas its generated graph model is over 42 times that size with 86.23 MB. Furthermore, that complete model has to be serialized from the `SKGraph` model on the server to a string, sent over the connection to the client, and deserialized there again. Not sending the complete graph twice, such as sprotty's concept is for server and client layout of the graph, already is a big improvement in data transfer though. Instead of sending the entire graph with its 86.23 MB for the micro layout and the final drawing to the client, resulting in 172.46 MB of data sent over the connection to draw the diagram just a single time, the new concept just uses 25.7 MB + 86.23 MB = 111.93 MB or 64.9 % of that data. If the server and client run on the same machine, this is possible to be sent and received quickly, but if the server and client run on separate machines, the connection has to be fast to not add even more delay.

5.1.2 Bottlenecks in the New Concept

Taking a closer look at the big example, showing how much time was spent in which step, better conclusions on bottlenecks and necessary improvements will show up. Table 5.3 provides this insight. It is divided into many steps that are executed on the server, on the backend part of Theia on the client, and on the frontend part of the client in the browser. The server was running Java 10, the backend part of the client with Node.js version 8.10, and the frontend part of the client with Chromium version 69. All parts were running locally on a single machine, so that connection speed between the systems was not an issue. The table lists all translations and costly calculations done on the model and added to the project by me, as well as some other important steps by KLighD, sprotty, and so on.

The first step to generate the `KGraph` view model is needed on the KIELER implementation and the new systems and is very important, as the graph structure with a description of all renderings is already generated in that step. Translating that `KGraph` model to the new `SKGraph` structure is needed so that sprotty can work with it on the client. The first new and

Table 5.3. Time needed per step during diagram generation.

	Step	Time needed
Server	SCChart to KGraph	3.28 s
	KGraph to SKGraph	1.23 s
	Text graph to JSON	3.50 s
	Sending text graph and other overhead	6.20 s
Client Backend	Receiving text graph, parsing etc.	4.60 s
Client Frontend	Text graph to SVG	0.63 s
	Other overhead	20.00 s
Server	Micro layout	0.54 s
	Macro layout (ELK)	4.37 s
	Mapping of results	0.02 s
	Complete SKGraph to JSON	28.66 s
	Sending complete graph and other overhead	4.60 s
Client Backend	Receiving complete graph, parsing etc.	38.26 s
Client Frontend	Complete graph to SVG	3.31 s
	Other overhead	66.68 s
	Total time needed	**185.88 s**

more expensive step now is the estimation of all text sizes. Generating the 25.7 MB of JSON, sending it to the client, and receiving and rendering it there with more overhead produced by the browser and sprotty is additional time needed by this concept. Just these steps alone take as long as KIELER needs to generate and draw the entire diagram. Back on the server, the steps for complete layout are executed, just like in KIELER, and now the complete graph with all renderings, styles, and positioning for every element and rendering is translated to JSON and transmitted to the server. Note that the model sent in this step is much more complex with deeper hierarchy than the model with only all texts, and is therefore more difficult to serialize and parse. Also, the 86.23 MB of data that need to be generated, sent, and processed take a long time. The 28.66 s needed to serialize the graph, the 38.26 s needed by the client to receive and deserialize it again, and the 66.68 s needed by the browser to handle other overhead of Theia and sprotty are the main bottlenecks. They need more attention to reach a more competitive comparison to the diagram generation of KIELER.

When new features were implemented in KIELER and the system of compiling, simulating, and generating diagrams improved over time, one question asked as a benchmark was always "…but can it run the railway controller?" The railway controller is an SCCharts program from

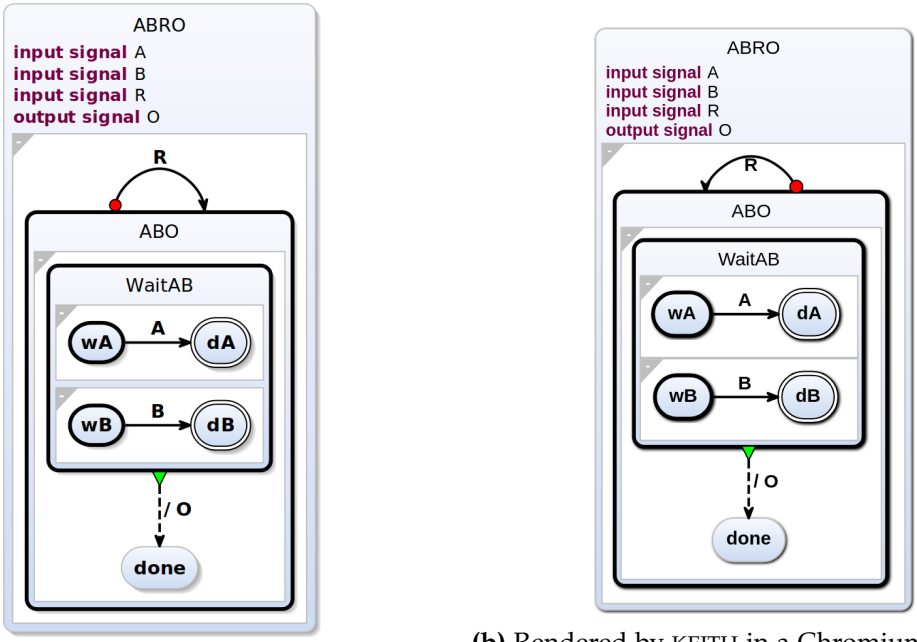
the same project that the largest example in this evaluation comes from. The goal of that project was to implement a controller which can automatically schedule 11 trains to drive around a model railway. This includes that they all automatically calculate the path they need to drive without crashes and that no part of the system can run into a deadlock where some train cannot reach its destination anymore. Because this program is one of the biggest productive SCCharts ever created, it is a good benchmark to test newly implemented features.

So my new implementation was also tested with the railway controller and it causes a problem in the last step while drawing with sprotty. I already tracked down the issue to be fixed soon after the end of this thesis. It has to do with some named elements in the completely expanded version of the controller that are not unique within the same level of hierarchy. This is marked as a warning in SCCharts and was not yet foreseen as a possibility during the implementation of the view generation. However, the railway controller and the railway environment (the big example file used earlier with the 35 571 LoC) are both very similar in size and complexity. The railway environment program does not have the one problem that causes the diagram generation to fail with the controller and the implementation is able to draw a complete diagram. So yes, the new system will probably be able to run the railway controller, although it is slower than KIELER.

Another bottleneck is the memory usage. The new concept needs to store the `KGraph` model and the `SKGraph` model on the server and the `SKGraph` model a second time together with the SVG rendering on the client. That is double the amount of completely stored models compared to the old concept, where only the `KGraph` model and the space used by KIELER's viewing framework for the rendering are stored.

5.1.3 View Comparison

The generated graphs by KIELER and the new environment look very similar, which is a good indicator that the new implementation is correct for the renderings already implemented. Figure 5.1 shows a side by side view of both visualizations. The difference that the self loop with the label `R` and the label itself are placed differently is not because of the rendering itself, but because of a different configuration between the two programs for the handling of self loops in ELK. Apart from that, the differences are justified. Since the texts are rendered by different frameworks, the exact look of the same font in KIELER's SWT framework and the SVG rendering of Chromium is one of the differences. The bold font in the KIELER rendering is noticeably fatter and causes the text to stretch over a wider area. This also shows the necessity of calculating the text sizes on the client, because otherwise texts might be way smaller in the new environment than the space reserved for them or texts might overlap. Next, the shadows look different. This is because I implemented them in a different way than they are implemented in KIELER. When viewed closely, they are much smoother and more realistic now. Furthermore, some spacings differ between the models, especially visible between the two regions in the state called `WaitAB`. That is an issue that needs to be looked into at some point in the future.



(a) Rendered by KIELER.

(b) Rendered by KEITH in a Chromium browser.

Figure 5.1. Comparison of a rendered SCCharts example.

5.2 Developing with Theia and Sprotty

The frameworks of Theia and sprotty developed by TypeFox formed the basis of this thesis. Both frameworks are relatively new, the following is a brief experience report. First of all, the integration of the LSP and the concept to extend it for any communication between the client and the server is a great way to start developing in a multi-system environment and with web technologies. The LSP handles so much of the background functionality to make the communication work in the first place that I did not need to worry about that aspect.

Getting a simple Theia extension up and running was easy, because the tutorial on how to extend basic behavior is understandable and leads the user the right way. However, getting more complicated concepts to work in Theia, such as the viewing framework sprotty itself was not as easy. The Yangster example helped a lot to have a working version of Theia with sprotty, but changing its behavior was not as easy, because just understanding what is happening in that code is not easy without detailed documentation of the classes or a tutorial on how to extend more of the functionality of sprotty. Not surprisingly, given the novelty of this infrastructure, the blog posts only give hints on how the separate parts work together. Thus I still had to look through the mostly undocumented code of Yangster to understand what was happening and where to change the behavior. It was therefore a big advantage for this project to have the helpful support from the tool developers in case of unclarities.

A step-by-step tutorial for sprotty on how extensions to the diagram server and to the language server work on a more technical level or at least some more documentation on the

examples such as Yangster would have helped my concept and implementation. Also, the concept of dependency injection and how default behavior of sprotty can be overwritten by placing the bindings in some modules on both the server and client side could be explained in more detail, as this is heavily used on both sides. With that missing, it is still visible that Theia and sprotty are still quite new with a steep learning curve, which could be flattened by solving those issues.

In general, using these frameworks with the web-based concept opens up new possibilities, as was hoped for when starting with this project. One remaining issue, at least with the current prototype, appears to be performance for bigger models, as explained further in this chapter and Section 6.1. Whether this is a problem of the current use of Theia in this project or whether improvements in Theia itself can address this issue remains to be investigated.

5.3 Completion of Proposed Goals

After getting the topic to work on for this thesis in the beginning of the semester I made a list of hard requirements that should be a core part of this thesis and therefore be met by the end as well as soft requirements that are assets to round off the work, but were no necessity.

5.3.1 Hard Requirements

As the title of the thesis suggests, one goal was to move the entire system away from Eclipse and into web technologies. It is now implemented with web technologies in Theia, sprotty, and the LSP, but it is still not yet separated from Eclipse completely. The language server has to be started as an Eclipse application because the extension points that define which diagram syntheses are known by KLighD are specific to the Eclipse environment and need a headless Eclipse application to be started with the server.

The first proposed hard requirement was to implement a Theia extension that contains the sprotty view. This forms the fundamental base implementation to build on. This goal was easy to realize after I found the Yangster example that was a proof of concept of exactly this task.

Another requirement needed a lot more thought than initially expected: getting the strict separation, which steps of the diagram generation to handle on the server or the client and which parts to reuse from the original implementation in KIELER. Most sections of Chapter 3 are designated to the separation. As described in that chapter, I found a solution which is different from sprotty's original approach and it was shown to work.

Furthermore, the extension of new messages for the LSP for KIELER style diagram generation in the web was a goal set in the beginning and was met. This includes the messages for the text size estimation, the messages for option handling, and the concept of how actions can be handled.

The translation of as many `KRendering`s as possible to `SGraph` was also seen as a goal after Theia and sprotty initially worked together on the client and server. This goal was rephrased

later though, because KRenderings do not represent graph elements, but still needed to be transferred to the rendering framework by sprotty. The solution was to take the KRenderings as they were and see them as a part of each SGraph element. What was also meant by this goal was that the KRenderings needed to be translated to the visual representation SVG. Not every possible rendering and style defined for the KGraph model is included in the implementation now, but enough of it to correctly display one of the DSLs that uses KIELER visualization, namely SCCharts.

The last hard requirement stated in the beginning was to have at least one of the possible interactions with graphs in KIELER to be also possible in the new implementation. Meant were the options to change the synthesis or the layout of the graph, which are now accessible and working as a proof of concept in a separate view within Theia.

So every hard requirement proposed in the beginning was met, which shows that the vague impression of the problems to be faced were correct with a few additional problems that were not foreseen initially.

5.3.2 Soft Requirements

An optional goal was to reimplement the *incremental update* from the KIELER implementation as well. The incremental update means that a small change in the code, such as adding a single object to the visualization, should also just update that single part of the diagram model. Because sprotty already supports a part of this feature, it is also already included in the new implementation. However, a complete new model is still sent to the client, where sprotty then notices the change in the model and only applies that change in the rendering. That way, an animation in the rendering lets it at least look as if only a small part of the diagram was updated, while in the background complete new models are still sent back and forth. Only sending a delta containing the changes between two iterations of the model would also relieve the communication.

Next, the interactivity to click on an object in the diagram and the corresponding lines of code for that getting highlighted was an optional feature. Because sprotty also has a feature for that, it only had to be adapted, so that every view model element knows about its originating object in the source model and this feature works as well in the new implementation now.

The last proposed soft goal was formulated very vaguely, as it stated that all diagram features from KIELER's diagram framework should also work in Theia. Because KIELER has been developed for multiple years now, this could not be reached in the short amount of time available here. But at least one aspect was conceptualized already on how it could work in Theia. That feature is the interaction with actions defined during the diagram synthesis. As described in Section 3.4, this needs to add more messages to the LSP and be executed on the server to update the model afterwards.

In conclusion, the goals proposed have all been looked into or implemented, but there still is a lot of future work to do in polishing the implementation and adding more functionality to it.

5.4 Lessons Learned

Working on the concept, implementation, and writing for this thesis is the biggest project that I have worked on by myself so far. So there are some lessons that I can take away for working on similar projects in the future.

The main hurdle in the beginning was the sheer amount of different technologies used for this implementation. Researching about the current setup in KIELER with KLighD, ELK, and similar used technologies and—after settling for Theia and sprotty—also learning about all the technologies they use was a big part in the beginning. Finding a point where to start implementing new functionality, such that executing the code shows a relevant change, was complicated. Starting in a slightly wrong direction while first experimenting with the technologies turned out to be not that bad. While trying to build an executable `.jar` file for a basic language server generated by Xtext within the main build system of the working group I already got a better look into some of the technologies. However, using this executable file turned out to hinder my development, because I would not be able to debug the new functionality I needed to add to the server.

When implementing a new software with many dependencies to other projects it is also necessary to start with a good and complete setup of the development system. I started programming already when I only had basic dependencies added to my project on my own and ran into problems with new dependencies regularly. At one point, the whole system stopped working for me for no apparent reason. Only after restarting to program with a new and complete development system, the real productivity started.

Furthermore, implementing with two different languages that were new to me was a nice step to expand my knowledge and expertise. I programmed with Java quite often already and the language extension Xtend only improved its usability. Developing with Xtend in Eclipse for the server side of the implementation was very easy with hot-swappable code and easy debugging. That made the turnaround time for server development very fast. Also developing with the JavaScript extension TypeScript made client programming a lot easier than anticipated for only using JavaScript. The object-oriented approach of TypeScript made the development much more Java-like and nicer to use. Because the Theia application and the Yangster example that I based my implementation on are built with Node and Yarn, they have a longer turnaround time. The client had to be rebuilt for every change in the code with no good working quick way to hot-swap code in a running program. Because of the project setup with submodules by sprotty, VSCode was not able to correctly show errors in the code while coding, which may as well be because of a faulty configuration on my side, and the build time of the application of two minutes caused this side of the implementation to progress a little more slowly. An especially typical error happening when programming in TypeScript was to forget the opening and closing brackets on a function call with no parameters, which caused the function to be assigned to some variable, not the result of that function. This is especially bad as it will not always cause an error while compiling, but when executing the behavior will not be as intended. Debugging the client application in Chromium was user-friendly, though.

As a last point, making estimations on how long any part of the implementation, researching, or writing text takes is especially difficult as almost everything takes longer than first expected. Some tasks that are thought to be completable within the same day often keep on needing more attention for details. Especially for the part of writing, doing so in a foreign language also takes more time writing, researching, and correcting than in my mother tongue. However, writing this thesis helped a lot to get a deeper look into the field of scientific research and working in an academic setting.

Chapter 6

Conclusion

This thesis shows a possible way to move transient views and automatic diagram generation from the monolithic approach in Eclipse to a new system using web technologies. The solutions for some problems that arose during the migration of the system are presented and discussed thoroughly. The concepts used for solving diagram generation on two separate systems, the server and the client, can be used for any other web-based application as well. Also, the use and extension of the LSP is a great way for having generic support for many IDEs using that technology, but also for adding language-specific details to the protocol. These can help to find new features to be added to the LSP by Microsoft and to define an own API on how to use the DSLs in different scenarios.

This thesis also shows, how changing the underlying platform of systems can be a hassle on one side, but open up new possibilities not thought of before yet.

As the concepts described here are implemented as a working prototype, this leaves room for improvement in the implementation or changes to the concept that can be researched and evaluated in future studies. Furthermore, the new platform presented here with the diagram generation and interaction features added to the LSP API opens up possibilities for KIELER diagrams to be used in other projects based on Theia or other web-based scenarios. The new IDE KEITH with Domrös' [Dom18] and my part is not completed yet and is currently in an experimental state, but will not be some closed project used only for our theses. New frameworks and techniques from the world of web technologies can be tested and used for new research projects, presented to students for a different insight into the topics of the working group, and much more.

6.1 Future Work

Because KIELER has much more features than KEITH regarding the diagram generation and because the new platform offers new problems and opportunities, what can be done is still a big list of new features, concepts, and improvements on the current implementation. This is a list of everything left over during the implementation, new ideas, and features left for later.

Bundle a Theia diagram extension: Currently the implementation of all diagram features is an extension of Theia bundled together with running a language server and everything needed for Theia to run as an application. But as Theia's concept is to bundle many extensions together to form a complete IDE, the one extension to handle everything is not the final way on how KEITH should be built. The basic KEITH IDE should be a Theia

extension with very little functionality. Running the language server is one of the parts, but not much more other than that. To have automatic diagram generation, another extension with the functionality described in this thesis should be added as a Node.js Package Manager (npm) dependency. Thus, the implementation needs to be restructured into an npm package, which can be included in KEITH or even in any other IDE based on Theia that should be able to display KIELER style diagrams.

Interaction with compiler results: The bundling of packages and building of a complete KEITH application also includes further interaction between Domrös' and my implementation. His compiler view generates a lot of intermediate results which can be drawn with KLighD in the KIELER equivalent. So not only the source model should be able to be drawn in a diagram view, but also all intermediate compilation results with registered view syntheses in KLighD.

Move micro and macro layout to the client: This is a feature that should be discussed further before implementing and should maybe be an option to toggle between server and client layout. The main reasons why the layout is executed on the server for this implementation are that a server will generally have more computing power to handle the layout and that micro and macro layout already work on the Java side because of the implementation of KLighD. However, since the micro layout for text size estimation has to be executed on the client anyway and big models are being sent to the client twice, this can reduce network usage and improve the generation time of diagrams. When the complete layout is calculated on the client, only one model needs to be sent to the client with even less data, because all renderings and graph elements do not need to have their positions sent as well. This is also a feasible approach because sprotty has the ability to render all elements in a hidden renderer for calculating the bounds of each element and because it can also call elkjs, the JavaScript version of ELK.

Remove text size calculation from the client: This is also an idea to be discussed, as it is the opposite of the previous point. The text sizes can be estimated on the server to remove the complete overhead of sending big models to the server twice. The server could send some representative example texts to the client and let it calculate the size of those texts. The server can use those sizes to estimate the size of all remaining texts by itself by scaling the calculated size of the texts on the server with the ratio between the server calculation of a similar example text with the client calculation of the same example text. This idea will remove some of the overhead caused by using the web-based approach at the cost of possibly inaccurate boxes around texts in the rendering.

Gitpod integration: TypeFox just recently released a public beta for Gitpod,[1] which is a Theia implementation that plugs directly into online git repositories, such as GitHub. That way, it allows developers to work on any project in GitHub with just one click, as Gitpod sets

[1] https://www.gitpod.io/

up a Theia IDE in any repository. Eysholdt [Eys18] published a blog post with further details.

Because KEITH is also based on Theia, this concept can be reused as the diagram generation features and the entire functionality can be added to a Gitpod implementation with Theia extensions. That way, coding examples, homework, and other projects can be set up with Gitpod to be available quickly as an application everybody can use. Also on conferences, where some new concept for synchronous languages such as SCCharts gets presented, the audience could easily play around and test what is presented by just clicking a link.

Interaction with KLighD actions: As already mentioned in Chapter 3, the interaction possible with KLighD is not completely implemented yet. The actions that can be attached to any `KRendering` during the diagram synthesis and are possible to be activated with mouse interaction on the rendered elements in KIELER are not implemented on the new SVG renderings on the client yet. Hence, the new messages for that described in Section 3.4 have to be added to the protocol and their handling needs to be implemented on the server and the client.

Structure-based editing: An idea to fill the work of another thesis is to extend the concept of generating a view in sprotty from a text source to also be able to modify the underlying text source from the sprotty view itself. Some interactions can be added to sprotty to allow the user to add new visual objects, change texts in the diagram, or change styles defined on the objects which modifies the text model accordingly. That way, the user can modify the diagram in a direct way without needing to know the syntax of any underlying language. This idea could be implemented for the `KGraph` model only or even for some DSL such as SCCharts that is visualizable with a `KGraph` view model.

Layout options: The diagram options view with options from KLighD syntheses is already implemented, but not all options are applied yet. The layout options that should change the way ELK calculates a layout on the `KGraph` view model are not used in the layout call yet. In fact, the layout currently only uses the default configuration. However, that can be changed easily in the future.

Diagram options view beautification: The diagram options view is only a working proof of concept right now and uses a very basic visualization within its widget. Prior to any kind of release of the software, the view needs to be beautified first. Furthermore, the diagram view is not even capable of displaying every type of option yet, as for example an option with a continuous slider is not yet implemented. Full support for all option types and a beautification of the view have to be implemented.

Diagram view and editor synchronization: Currently opening the diagram view is not changed from the default implementation of the Yangster example. That means, the user needs to right-click in the editor of the model that should be generated and click on “Open in Diagram” to open a diagram. The opened diagram will then be linked to that specific

source file. However, it should be linked to the currently active editor and always show a diagram for that, such as the diagram options view already links its content to the active editor and as KIELER also links the shown diagram to the active editor.

Improve communication to the user: If some error happens during a request to the server, if the server is slow in responding to requests, or if something else happens on the server or the client, the user should be notified in some way other than an error message in the console log of either the server or the client. That way, the user knows what went wrong and can file a bug report or notify the developer with a more detailed message than just "The application did not respond the way I expected." Also a message telling the user that the requested diagram is on its way could be a better way of communication than just showing a plain white background until the diagram generation is finished.

Implement the remaining `KRenderings` and `KStyles`: Because there are many different `KRendering` and `KStyle` objects that are not completely covered when drawing graphs for the SCCharts language, not every possible visualization has been implemented yet. The current collection already gives a good idea of how translating `KRendering` and `KStyle` objects to SVG works in TypeScript. The remaining objects also need an implementation for the full support of possible KLighD syntheses.

Optimize sent data and SVG generation: Some unnecessary values are sent during some of the messages for diagram generation and option handling. For example, the `KRenderings` sent to the client can contain insets that are only used for the micro layout, which is not executed on the client, so that data is not needed on the client anyway. Also some data attached to `KGraph` elements other than its current rendering is attached and serialized and can be removed from the messages.

Another optimization is to ignore default values during serialization. Default relative positioning could be 0 pixels in x and y direction, so renderings with that position will not send any value for x and y and instead omit it in the message.

Next, the message to estimate the size of every text element also contains too much information that is not needed. For example, the style describing the text color will not affect the size, so it does not need to be sent in that message. Another idea would be to remember the texts with their styles on the client sent in the first message. The second message would then not need to include all the texts with their styles again, but can just refer to the text stored on the client already via an ID.

Also, the SVG generation in its current implementation puts unnecessary empty group elements in some parts and explicitly sets values which are SVG defaults anyway. So the final generated SVG also contains superfluous data that can be removed.

These optimizations will all make the messages sent to the client and the SVG smaller and will therefore, assuming that excluding these parts from the messages does not add too much time overhead, increase the overall speed of the diagram generation process. Furthermore, there are probably even more optimizations possible than those listed here.

Compress diagrams: Another idea to reduce the network usage is to compress large text files, such as the JSON representations of the view models. Because it is just text with highly redundant syntax and keywords, those messages can be compressed very much. An example JSON graph from Section 5.1 with 3.61 MB can be compressed to 129 kB or 3.57 % of its original size with a simple `zip` command. Because it is more likely that network capacity is more of a limiting factor than processing power for a simple `zip` and `unzip` command, this tradeoff should definitely be considered.

Choosing the used diagram synthesis: For a single domain model there can be multiple syntheses registered in KLighD, which can all be used for diagram generation. Currently only the most specific one is always taken, but for example SCCharts can be synthesized to the `KGraph` view model directly with the synthesis for SCCharts or with the fallback synthesis for the Java `EObject`, which is a superclass of SCCharts. The user should be able to choose between every available synthesis.

There are probably even more features of KIELER that are not included in this list and also should be implemented in KEITH. Starting this big project now leaves a platform with many ways to extend and improve it for further use within the working group.

Bibliography

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9.

[BCE+08] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. *Xtext user guide*. 2008.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[BGM04] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. "Toolkit design for interactive structured graphics". In: *IEEE Transactions on Software Engineering* 30.8 (2004), pp. 535–546.

[Bjo10] M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020. RFC Editor, Oct. 2010. URL: http://www.rfc-editor.org/rfc/rfc6020.txt.

[Bün18] Hendrik Bünder. *What's new in Xtext and Xtend 2.14: The Journey of Tina Toolsmith*. 2018. URL: https://blogs.itemis.com/en/whats-new-in-xtext-and-xtend-2.14-the-journey-of-tina-toolsmith (visited on 09/18/2018).

[Dom18] Sören Domrös. "Moving Model Driven Engineering from Eclipse to Web Technologies". Unpublished master's thesis written alongside with this work. 2018.

[Eff17] Sven Efftinge. *Theia – VS Code in the Cloud*. 2017. URL: http://typefox.io/theia-vs-code-in-the-cloud (visited on 09/10/2018).

[EV06] Sven Efftinge and Markus Voelter. "oAW xText: a framework for textual DSLs". In: *Eclipse Summit Europe*. Esslingen, Germany, 2006.

[Eys17] Moritz Eysholdt. *Xtext 2.13.0 Released: Semantic Editing Made Easy*. 2017. URL: https://typefox.io/xtext-2-13-0-released-semantic-editing-made-easy (visited on 09/18/2018).

[Eys18] Moritz Eysholdt. *Gitpod: A One-Click Online IDE*. 2018. URL: https://typefox.io/gitpod-a-one-click-online-ide (visited on 10/30/2018).

[Fac14] Facebook, Inc. *Draft: JSX Specification*. 2014. URL: http://facebook.github.io/jsx/ (visited on 09/11/2018).

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2013.

[Hun13] Pete Hunt. *Why did we build React?* 2013. URL: https://reactjs.org/blog/2013/06/05/why-react.html (visited on 09/11/2018).

[Köh17a] Jan Köhnlein. *Extending a Language Server With Sprotty Diagrams*. 2017. URL: http://typefox.io/extending-a-language-server-with-sprotty-diagrams (visited on 09/21/2018).

[Köh17b] Jan Köhnlein. *sprotty – A Web-based Diagramming Framework*. 2017. URL: http://typefox.io/sprotty-a-web-based-diagramming-framework (visited on 09/21/2018).

[Köh17c] Jan Köhnlein. *YANG-Tools: One Language Server for Four IDEs*. 2017. URL: http://typefox.io/yang-tools-one-language-server-for-four-ides (visited on 09/21/2018).

[KS17] Jan Köhnlein and Miro Spönemann. *sprotty – Graphical Views for Web-Applications*. Presentation Slides from EclipseCon Europe 2017. 2017. URL: https://www.eclipsecon.org/europe2017/sites/default/files/slides/Sprotty%20-%20ECE17_0.pdf.

[Mic16] Microsoft Corporation. *TypeScript Language Specification*. Version 1.8. 2016. URL: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md (visited on 09/11/2018).

[Mic18a] Microsoft Corporation. *Language Server Protocol – Overview*. 2018. URL: https://microsoft.github.io/language-server-protocol/overview (visited on 10/19/2018).

[Mic18b] Microsoft Corporation. *Language Server Protocol Specification*. Version 3.13. 2018. URL: https://github.com/Microsoft/language-server-protocol/blob/gh-pages/specification.md (visited on 09/20/2018).

[Mic18c] Microsoft Corporation. *Visual Studio Code Documentation – Example – Language Server*. 2018. URL: https://code.visualstudio.com/docs/extensions/example-language-server (visited on 10/23/2018).

[Mic18d] Microsoft Corporation. *Visual Studio Code Documentation – User Interface*. 2018. URL: https://code.visualstudio.com/docs/getstarted/userinterface (visited on 10/22/2018).

[Ree79] Trygve Reenskaug. *Models – Views – Controllers*. Xerox PARC technical note. 1979.

[RLK+15] Jonathan P Rochelle, Micah G Lemonik, Farzad Khosrowshahi, and John Danaher. *Converting spreadsheet applications to web-based applications*. US Patent 9,009,582. Apr. 2015.

[Rub06] Dan Rubel. "The Heart of Eclipse". In: *Queue* 4.8 (2006), pp. 36–44.

[Spö15] Miro Spönemann. *Graph layout support for model-driven engineering*. Kiel Computer Science Series 2015/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2015. ISBN: 9783734772689.

[SSH12] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Transient view generation in Eclipse". In: *Proceedings of the First Workshop on Academics Modeling with Eclipse*. Kgs. Lyngby, Denmark, 2012.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

Detailed Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Related Work . . . 3
- 1.3 Outline . . . 4

2 Used Technologies **7**
- 2.1 The Language Server Protocol . . . 7
- 2.2 Browser Technologies . . . 9
  - 2.2.1 TypeScript . . . 9
  - 2.2.2 JSX . . . 10
  - 2.2.3 Theia . . . 11
  - 2.2.4 Sprotty . . . 13
- 2.3 Server Technologies . . . 16
  - 2.3.1 Xtend . . . 16
  - 2.3.2 Xtext . . . 16
  - 2.3.3 Gson . . . 17
  - 2.3.4 Eclipse Layout Kernel . . . 18
  - 2.3.5 KLighD . . . 18
  - 2.3.6 SCCharts . . . 19

3 Concepts **21**
- 3.1 Overall Architecture . . . 21
- 3.2 Choice of Frameworks . . . 24
  - 3.2.1 Why Theia? . . . 25
  - 3.2.2 Working with Sprotty . . . 27
- 3.3 The View Model . . . 29
  - 3.3.1 Diagrams 100 % Sprotty Style . . . 31
  - 3.3.2 Diagrams 100 % KLighD Style . . . 32
- 3.4 Communication . . . 34
- 3.5 Diagram Options View . . . 39
- 3.6 Rendering on the Client . . . 41

4 Implementation **43**
- 4.1 Server Implementation . . . 43
  - 4.1.1 Classes Used on the Server . . . 43
  - 4.1.2 Example Request of a Diagram . . . 47
  - 4.1.3 Translating the `SKGraph` to JSON . . . 53

4.1.4 Option Handling . . . 55
4.2 Client Implementation . . . 56
4.2.1 View Generation . . . 56
4.2.2 New Messages . . . 63
4.2.3 Diagram Options View and Messages . . . 64

5 Evaluation **67**
5.1 Comparison Between the Web-Based Approach and KIELER . . . 67
5.1.1 Performance Comparison . . . 67
5.1.2 Bottlenecks in the New Concept . . . 69
5.1.3 View Comparison . . . 71
5.2 Developing with Theia and Sprotty . . . 72
5.3 Completion of Proposed Goals . . . 73
5.3.1 Hard Requirements . . . 73
5.3.2 Soft Requirements . . . 74
5.4 Lessons Learned . . . 75

6 Conclusion **77**
6.1 Future Work . . . 77

Bibliography **83**

Lists **87**

List of Figures

2.1 Example communication using the LSP. 9
2.2 Theia's user interface. 13
2.3 sprotty's client architecture: flow of information and the virtual DOM. 14
2.4 Example application with two sprotty views of a parallel algorithm with server communication. 15
2.5 The diagram visualization of a small example SCChart. 19

3.1 KIELER Lightweight Diagrams: the road from a textual model to a drawn diagram. 21
3.2 The architecture for diagram visualization in a client-server environment. . . . 23
3.3 sprotty's server architecture. This shows a generic way to allow a diagram server to handle a view model and a layout engine with communication of actions to a server. 28
3.4 The `KGraph` view model. 29
3.5 A simplified `SGraph` view model on the Java side. 30
3.6 An extension on the `SGraph` model to represent the `KGraph` called `SKGraph`. . . . 33
3.7 The diagram view and the diagram options view in KIELER. 39
3.8 The placement options for the area available in PhosphorJS widgets. 41

4.1 Sequence diagram describing every relevant part of the server implementation when the client sends a `requestModel` or `updateModel` action. 45
4.2 The immediate aggregations and generalizations of the `KRendering` class. . . . 51
4.3 The possible placement data to calculate a unique position for each `KRendering`. 52
4.4 Extract of the fields in an `SKNode`. 54
4.5 Class diagram of all possible `KContainerRenderings`. 57
4.6 Class diagram of all possible `KStyles`. 60

5.1 Comparison of a rendered SCCharts example. 72

List of Tables

2.1 Language support without and with use of the LSP. 8

5.1 Time to diagram in KIELER and the new environment. 68
5.2 Model sizes sent via network. 69
5.3 Time needed per step during diagram generation. 70

Listings

2.1 Example code using JSX. 11
2.2 Translated JavaScript code from the code in Listing 2.1 without JSX. 12
4.1 A `KPosition` translated to ambiguous JSON. 53
4.2 A `KPosition` translated to unambiguous JSON. 53
4.3 Implementation of a `KText` rendering with JSX. 57
4.4 Definition of the filter for `KShadows`. 61