A Web-Based Graphical Environment for Using Domain-Specific Languages in Lightning

Christophe Kamphaus
Laboratory for Advanced Software Systems
University of Luxembourg
6, avenue de la Fonte
L-4364 Esch-sur-Alzette
Luxembourg

TR-LASSY-17-01

Faculty of Science, Technology and Communication

A Web-Based Graphical Environment for Using Domain-Specific Languages in Lightning

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Christophe Kamphaus

Supervisor:
Prof. Dr. Pierre Kelsen

Reviewer:
Ass. Prof. Nicolas Navet

Advisors:
Christian Glodt
Loïc Gammaitoni

August 2016

Declaration of Authorship

I, Christophe KAMPHAUS, declare that this thesis titled, 'A Web-Based Graphical Environment for Using Domain-Specific Languages in Lightning' and the work presented in it are my own. I confirm that:

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

The Lightning tool provides an environment for developing modeling languages based on Alloy. The existing Lightning tool cannot edit language models with the concrete syntax applied. The object of this thesis is to equip the tool with the capability to generate web-based graphical editors for the domain specific languages modeled in Lightning.

The requirements for a web editor that is user friendly and allows the editing of visual languages were defined. For a subset of those requirements an editor allowing the edition of language models conforming to a Lightning language was designed and implemented.

The validity of those models can be checked with respect to some basic constraints present in the definition of the language of the meta-model.

We validated our work by analyzing how well the editor fulfills the requirements. The effectiveness of the editor was tested in a case study.

Acknowledgements

First I would like to thank my supervisor Professor Pierre Kelsen for giving me this interesting topic. His interest in my progress also drove me to achieve my best.

I would like to thank Christian Glodt for reviewing my code and guiding my refactoring efforts and Loïc Gammaitoni for helping me with any Lightning and Alloy related question. I would like to thank both for their advice and stimulating conversations and for taking the time to debug parts of the program and guide me in my efforts. Their feedback was also very valuable to improve this thesis.

Finally, I would also like to thank my family for their support.

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
ASM	**A**bstract **S**yntax **M**odel
ASM2VLM	**ASM to VLM** transformation
DB	**D**ata**B**ase
CSM	**C**oncrete **S**yntax **M**odel
CSS	**C**ascading **S**tyle **S**heets
CRUD	**C**reate, **R**ead, **U**pdate, **D**elete
F-Alloy	**F**unctional **Alloy**
GUI	**G**raphical **U**ser **I**nterface
HTML	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
ID	**Id**entifier
JAR	**J**ava **Ar**chive
JDBC	**J**ava **D**ata**B**ase **C**onnectivity
JPA	**J**ava **P**ersistence **A**rchitecture
MDD	**M**odel **D**riven **D**evelopment
MDE	**M**odel **D**riven **E**ngineering
MDSD	**M**odel **D**riven **S**oftware **D**evelopment
ORM	**O**bject-**R**elational **M**apping
SEM	**Se**mantic **M**odel
SEM2VLM	**SEM to VLM** transformation
UI	**U**ser **I**nterface
UUID	**U**niversally **U**nique **Id**entifier
VLM	**V**isual **L**anguage **M**odel
WAR	**W**eb application **Ar**chive
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

1.1 Context

Models are abstractions meant to emphasize relevant details. For example, a map can be seen as a model of the real world. Depending on the information we are interested in there exist several kinds of map (e.g. road map, topographic, physical or political map).

The essence of Model-Driven Development is to represent big, complex systems by the means of such abstractions. This is done to cope with the complexity and paired with formal verification – mathematical proof of correctness – to reduce the cost of projects by finding design errors at an earlier stage. During such development, domain-specific languages can be created to capture domain knowledge and make it available for reuse. These languages use a vocabulary from the problem domain and as such facilitate communication among domain experts. By assigning a well defined behavior to concepts of the language programmer productivity can be increased.

Domain-specific languages (DSL) can be either textual or graphical. In this work, we focus on graphical DSLs. Graphical DSLs have a grammar consisting of shapes and colors arranged according to the constraints of the language. To use these languages, we need editors. There are two kinds of editors [3, p. 100]: free-format editors (or symbol-directed editors), which allow all symbols of the alphabet to be entered at any position, but require a validation process, and structure editors (or syntax-directed editors), which require that only valid grammatical structures be entered. The development of these editors is often complex, very costly and time-consuming.

The consequence of that is that often features are missing, limiting the usability. Most editors need to be installed and some require an environment like Eclipse to run. Often these tools come with added complexity that harms the user experience, e.g. Eclipse is bloated with plugins and relies on a complex graphical framework.

To help the development of DSLs, meta-tools (tools used to create tools), called language workbenches, have been developed. These tools allow the specification of DSLs and generators for code or editors.

Lightning is a language workbench in the sense that it allows the definition of a DSL. Lightning's most distinguishing feature is the lightweight formal verification of the language, by generating language models satisfying or violating certain properties from the language definition. Lightning does not allow the definition of generators.

1.2 Problem Definition

In this thesis we propose to extend Lightning with the ability to generate from a language definition a graphical editor for that language. To do so, we define the requirements needed for an easy to use visual language editor. We then design and implement a generator that takes as input the definition of a visual language in Lightning and outputs an editor for that language. To comply with Lightning's current approach – one button push validation – we want to offer this generation capability with as few intermediary steps as possible (ideally with one button push as well).

1.3 Structure

The following part of the thesis is structured as follows:

In chapter 2 we provide background on Alloy, the definition of DSLs and Lightning needed for the proper understanding of this work. Chapter 3 presents the research questions of this thesis. In chapter 4 we define the requirements needed for our tool. In chapter 5 we present design choices for the implemented editor. Chapter 6 elaborates the details of the implementation. In chapter 7 we evaluate the resulting tool with respect to the requirements. Chapter 8 discusses the answers to the research questions and presents existing applications comparable to our tool. Finally, in chapter 9 we summarize our work and discuss future work.

Chapter 2

Background

2.1 Alloy

Alloy is a declarative language for describing structural properties. It is based on first order logic and relational calculus. The language comes with a tool called the Alloy Analyzer that allows the generation of instances either satisfying or violating certain properties, called facts, from a meta-model. This approach can be called lightweight formal verification of models [4].

The Analyzer works by reduction to SAT [5]. While Alloy's relational logic is undecidable, the Analyzer works around that by imposing an explicit scope on the number of elements for each meta-model concept. The tool can then perform an exhaustive search within the scope for satisfying or violating instances [6]. Thus the absence of bugs can only be proven within the scope. For bugs outside the scope, Alloy has the *small scope hypothesis*, which states that "most bugs have small counterexamples".

In Alloy elements of a meta-model concept are called atoms. Relations between atoms are called tuples.

2.2 Lightning

Lightning[1] [7] is a first step towards a language workbench based on Alloy [1]. It follows the approach proposed by Kleppe (cf. section 2.2.1) to define a language (shown in fig. 2.1).

[1] https://lightning.gforge.uni.lu/

The strength of Lightning is the possibility to formally verify (using Alloy) every part of a language definition. Weaknesses are that arithmetic and string operations are not well supported by Alloy and that there is no code generation or interface for using Lightning languages outside the Lightning environment.

Model transformations in Lightning are written in a sub-set of Alloy, called F-Alloy [8], which has the particularity of being interpretable, thus allowing the computation of model transformations in polynomial time (analysis is worst case exponential time).

See the appendix A for the Lightning license.

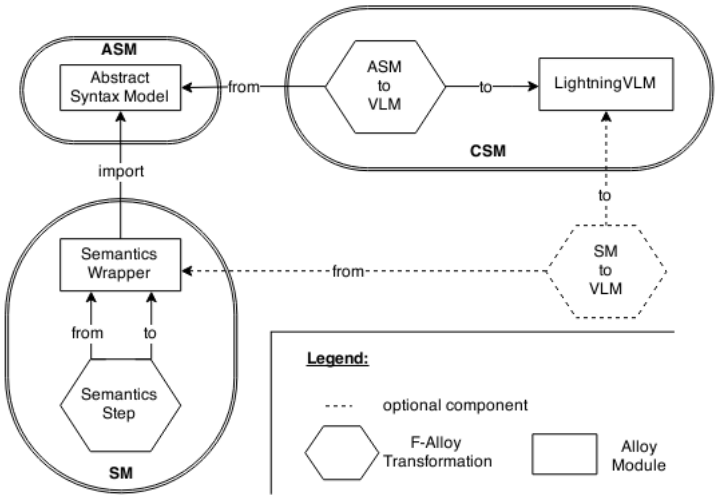

FIGURE 2.1: Representation of languages in Lightning [1]

2.2.1 Language definition

Abstract Syntax Models (ASM) specify the set of valid language models. Concrete Syntax Models (CSM) define the relations between language models and their graphical representations. A Semantics Model (SM) provides a meaning to those models.

Kleppe [3] defines a language as an abstract syntax model (ASM), concrete syntax model (CSM) containing a visual language model (VLM) and an ASM to VLM model transformation (ASM2VLM). Optionally a language can also contain a semantic definition (SEM) and model transformations to define a semantic step and the semantic to visual language mapping (SEM2VLM).

In Lightning the ASM, VLM and initialization of the semantic model are formally defined as Alloy modules. The mappings between ASM and VLM as well as SEM and VLM are

defined as model transformations, written in F-Alloy. The semantic step is also defined as an F-Alloy transformation.

We call the concrete syntax of an abstract syntax concept the representation of that concept. Appendix B shows the meta-model of the default Lightning VLM.

Kleppe also describes how a meta-tool supports the creation of DSLs (fig. 2.2) by having editors for the specification of the DSL, analyzers to verify the specification, executors to simulate it and transformers to create code, reports or other models. One such transformer is used to generate an editor or IDE for the DSL. This IDE might be a part of the meta-tool or a separate tool.

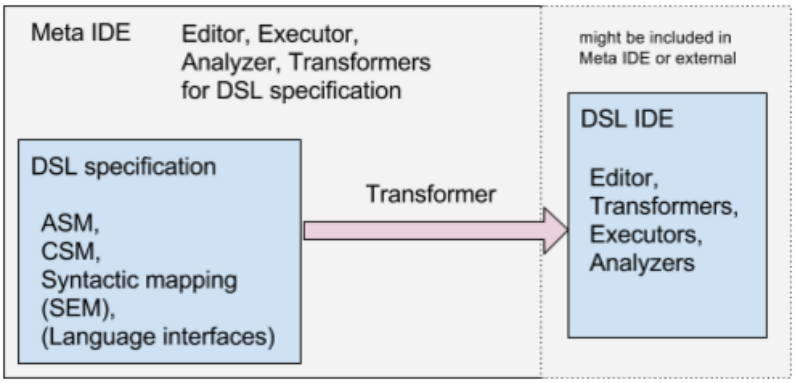

FIGURE 2.2: Relation between a DSL an its IDE according to Kleppe

Generally, the concrete syntax model (CSM) is understood to contain all the information necessary to visualize and edit the visual representation of language instances.

The Lightning CSM however only has the information needed to visualize language instances, by mapping abstract instance elements to visual elements. This transformation is not a priori bi-directional. We can thus not parse any arbitrary instance of the visual language model and obtain the ASM instance.

2.2.2 Feature overview

Features of Lightning include:

- An editor for Alloy and F-Alloy files with syntax highlighting, error markers and error correction
- An instance generator

- An instance editor using a tree-view of the instance
- An instance viewer which can depict instances using their concrete syntax
- A semantics simulator
- Import and export wizards from, respectively to Ecore
- A graphical meta-model viewer

Chapter 3

Research Questions

During the following part of the thesis we aim to answer two questions:

1. Is it possible to derive an editor from the existing Lightning language definition?
2. If not, what is missing in the language definition and how do we define these missing information in the Lightning plugin?

We already know that the existing Lightning language definition's CSM does not have a bi-directional transformation from ASM to VLM. It is thus not specified how to go from a VLM instance to an ASM instance.

The question is whether we can extract enough information from the ASM2VLM transformation to fill an editor palette and to synchronize ASM and CSM instances.

If the language definition does not have enough information to generate an editor, we need to extend the Lightning language definition to provide the missing information.

We need to define what is missing and how we specify this information in the language definition in the Lightning plugin.

Chapter 4

Analysis & Requirements

This thesis focuses on implementing an editor generator. First, we have to understand the requirements of a language workbench, of which our tool would be a part of. This is done in section 4.1. Second, we also need to understand the requirements of the resulting editor. In section 4.2 we list the requirements of a visual language editor. Requirements for a web editor are listed in section 4.3. Finally, we prioritize the requirements and select the requirements to implement in this thesis in section 4.4.

4.1 Requirements of a language workbench

In [9] Grundy et al. come up with key conceptual requirements for a meta-tool, like a language workbench. For that they use Sutcliffe's design meta-domain model [2] (shown in fig. 4.1) and their own experience.
Sutcliffe's design meta-domain model is part of domain theory and provides a conceptual framework for the design process. This model has functional components that act on resources and communicate through information flows to achieve a common goal. One part of this model is the use of tools to model the resulting design. With other tools, the quality of the result can be assessed (simulation or verification). The design process is supported by allowing communication through annotations of the models and brainstorming as well as collaboration. Designing is a creative process. Enhancing the way information is visualized shifts the cognitive burden away from comprehending the model and towards finding the best design solution. Since design rarely innovates from nothing, it is important to consider lessons learned, domain knowledge and reusable designs.

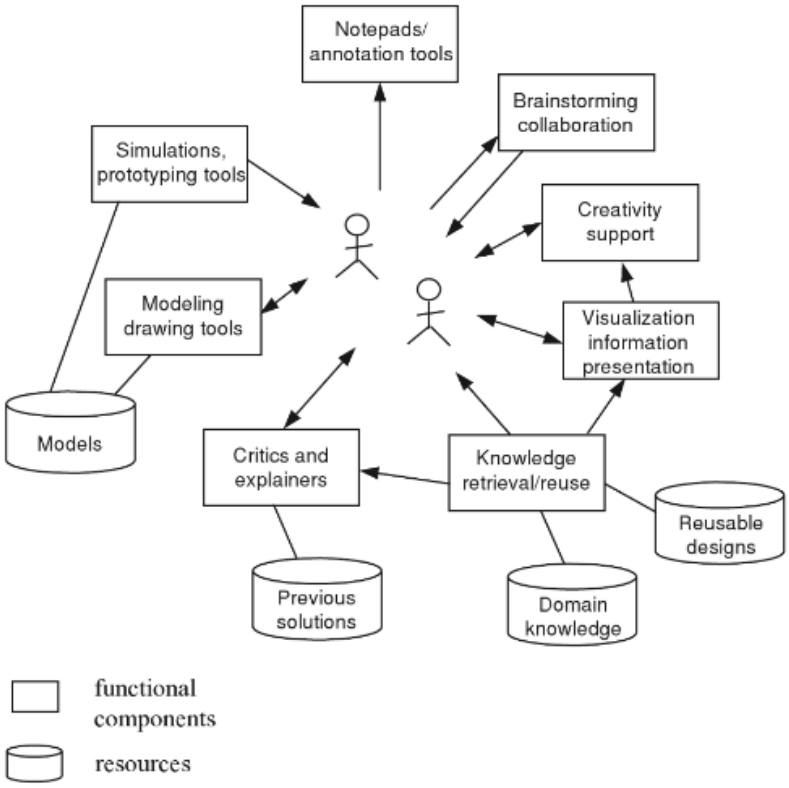

FIGURE 4.1: Sutcliffe's design meta-domain model [2]

The key conceptual requirements are:

(G1) **Specifying domain-specific visual language** (DSVL) **structural aspects**, which comprise the abstract syntax model (ASM), the concrete syntax model (CSM), consisting of the visual language model (VLM) and the syntax mapping between ASM and VLM (ASM2VLM) of the DSL.

(G2) **Specifying DSVL tool behavioral aspects**, which include dynamic and interactive tool effects such as event and constraint handling for both model and view manipulations and automated operations or processes.

(G3) **Critics and Transformations**, which support proactive feedback on model quality and the exchange of view and model information with other tools, and backend code generation.

(G4) **Human-Centric Tool Interaction**, which includes scalable, sharable, usable and intelligent support for collaborative and sketch-based editing and review.

(G5) **Architecture and Implementation**, leverage existing IDE facilities and related tools and making models available to domain users in appropriate ways.

In [10] Dietrich et al. analyzed how visual languages are defined. They compared the existing meta-tools for defining DSVLs – i.e. ADONIS (2000), ARIS(2008), AToM3 (2004), DIAGEN (2002), DIAMETA (2006), DOME (2000), EuGENia (2009), GEMS (2007), GenEd (1996), GenGed (2002), GME (2001), GMF (2012), GraMMi (2000), JComposer (1998), Marama (2006), MetaEdit+ v4 (2003), Moses (2001), PDE (2011), Pounamu (2007), TIGER (2006), VisPro (2001), VMTS (2008) and VS DSLT (2007) – according to the following capabilities:

(C1) **Specification at run-time** means that the user of the meta-modeling tools can specify their modeling language at tool runtime as opposed to having to recompile the editor.

(C2) Support for **multiple languages** means not having to redistribute a separate model editor for each language.

(C3) **Editor for representations** is important, since visual languages typically use symbols for their representations. Therefore the symbol editor has to be easy to use and flexible.

(C4) **Customizability of symbols** means that it must be possible to assign a different representation for different types.

(C5) **Customizability of edges** means that edge representations should be customizable.

(C6) **Customizability of edge symbols** means that it should be possible to use customizable symbols at the edge start and end points.

(C7) **Implicit relationships** use constraints like the positioning of elements to express a relationship.

(C8) **Shared repositories**, i.e. a centralized, consolidated repository of models instead of, for example, distributed flat-file storage, facilitates distributed modeling and makes it easier to maintain a consistent state.

The result of the analysis of all those tools shows that no tool has all the capabilities and many lack multiple capabilities. While this analysis is focused on (G1), we can assume that many of those tools are limited in how well they fulfill the requirements (G2-5) or that they ignore the requirements (G2-5) completely.

4.2 Requirements of a visual language editor

From the tools in [10] Dietrich et al. synthesized the following requirements for a graphical model editor:

(R1) A canvas with a 2D coordinate system is necessary.

(R2) Any graphical representation needs a position on this canvas.

(R3) Each element of a modeling language has its own graphical representation.

(R4) To account for modeling tool runtime specification capabilities, the graphical representation has to be configurable even after the meta-modeling tool's source code has been compiled. Therefore, it has to be importable and storable (serializable).

(R5) The easiest way to represent visible relations is to use edges between elements and provide them with different looks for easy distinction.

(R6) An edge needs a start and end location specifying where it is attached to the elements it connects.

(R7) Visible edges also have a path on the model canvas.

(R8) The path describes the way that is taken from its start to end point. To support proper routing of edges, an interface to graph layout algorithms is vital.

(R9) Implicit relationships between elements must be specifiable based on the relative position of elements to each other.

(R10) Explicitly supporting the documentation of implicit relations within the data model instead of leaving the relation truly implicit (i.e., only visible through relative positions and sizes is an important requirement. Establishing such implicit relationships the element that should contain another one should automatically adapt to the size of its content or support manual size modifications.

(R11) Some of the graphical representations need labels to further describe the modeling language elements.

(R12) To assist the user in modeling, the engine should provide certain standard functionality, which could be a mechanism to align model elements, to scroll through the model

canvas, or to zoom in and out. Other such functionality could be model export and printing.

(R13) It is necessary to integrate the representational aspects into the data model of the tool and thereby to enable the distribution of the representations by a shared repository.

(R14) The representation of model elements should be reusable in other modeling languages.

(R15) To design the representations of modeling language elements, a user-friendly tool (e.g. a wizard) is necessary.

4.3 Requirements for a web editor

In addition to the requirements identified for a DSVL editor, it is important to consider the following requirements:

(W1) **Ease of use** includes UI usability as well as response time and reliability. By making the tool a web-application, this helps with the ease of use, since the users will not need to install any programs. Usability is important for the acceptance of the tool by users.

(W2) **Collaboration** in real-time on an instance is useful in a team environment. It helps fulfill (G4).

(W3) **Security** is a critical requirement, for web editors. This means that the users need to authenticate to access any private models and languages. All API calls need to check the user's authorization concerning the language and instances they operate on, e.g. a user can only access languages that are public, that he owns or that he has been given access to.

(W4) **Logging** is necessary for monitoring use of the service and diagnosing any errors that occur.

(W5) **Availability**, while less important while the service is in the development phase, will become a requirement when the service is to be put into production and the service hopefully will have hundreds of users.

(W6) **Versioning** will allow users to have a history of a language or an instance and enable them to recover earlier versions.

(W7) **Validation** of instances gives us a strong assurance that an instance the user created is correct according to its meta-model.

4.4 Choice of requirements for our tool

By making the editor for DSLs web-based, users will not need to install any software to use the model editors and the editor will be platform independent. If this editor is hosted in the cloud, the languages and any language instances will be accessible from any machine from anywhere in the world. It will thus be very easy for the language engineer to share the languages with the users and to provide language updates. Users can also easily share language instances and thus collaborate. Any person familiar with the problem domain can directly start creating and editing instances of that DSL.

During the course of this thesis we will thus strive to implement a web editor for Lightning languages and a generator component for the Lightning plugin such that the Lightning plugin combined with the web editor fulfill the requirements (G1), (G2), (C2), (C4-8), (R1-14) as well as (W1), (W4) and (W7).

Despite what the design meta-domain model and research show are requirements for editors, We will not implement the following requirements during this master thesis:

- (G3-5) are requirements for a language workbench and not necessary for just an editor.
- (C1) is postponed due to time constraints, but a deployment scenario mentions this (cf. 5.3.3). In the future, the Lightning Web Editor is meant to be independent from the Lightning Eclipse plugin and then it will fulfill (C1).
- (C3) and (R15) An editor for representations is essential to improve the usability of the tool, but outside the scope of this thesis due to time constraints.
- (W2), (W3), (W5) and (W6) are also to be postponed.

The reason for not implementing them is that time is limited and we need to prioritize the core features of an editor.

In the following chapters, we will focus on fulfilling the retained requirements, but we may mention considerations taken in the perspective of extending the editor to also fulfill the postponed requirements, especially in regards to security.

4.4.1 Security considerations

Since the ultimate aim of the Lightning Web Editor is to be a language workbench hosted in the cloud, it will need strong security. Before the realization that implementing security would take too much time, we already made some design and implementation considerations for security. The server needs to integrate a framework for authentication and authorization, as well as communicate over HTTPS with any client.

While there are other minor Java security frameworks, the two main frameworks that are widely used and mature are Apache Shiro and Spring Security.

Spring Security has the advantage of being part of the Spring framework and thus more likely to work well with Spring JPA (persistence). It has the disadvantage of being more complex than Shiro. Apache Shiro still has all the important features (e.g. Resource Based Authorization Control, authentication with Basic header or with OAuth token). There also exist example code and tutorials on how to integrate Shiro with the Jersey framework.

If the implementation of security had not taken to much time, we would have chosen the Shiro framework. Now we leave its integration as future work.

Chapter 5

Design

First, we will present the workflow of the Lightning plugin with the editor and how the editor is structured. Second, the architecture of the server components is explained. Third, deployment scenarios for the Lightning Web Editor are explained. Then, the design of the different server components is elaborated. Finally, the validation of language instances created with the editor is described.

5.1 Structure of Lightning and Editor

Fig. 5.1 illustrates the workflow of Lightning and the new editor. Points (1), (2) and (3) are covered by the existing Lightning plugin.

(1) An engineer first defines a language.

(2) The engineer checks if the language is correct by generating and visualizing instances of that language.

(3) If it is not correct the engineer will refactor the language definition.

(4) The engineer will also be able to execute the generation of an editor.

(5) Users can use this editor to create instances of the language that are validated against the language definition.

(6) If the language definition is incorrect or incomplete the user can ask the engineer for modifications.

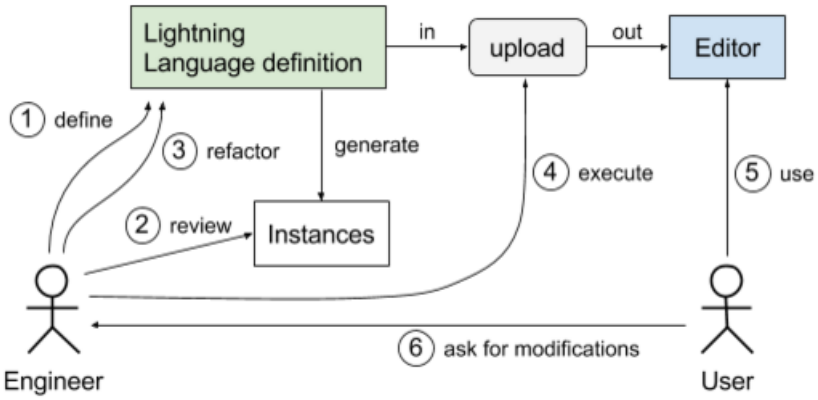

FIGURE 5.1: Use case of Lightning and Editor

The Lightning Web Editor is composed of two parts (shown in fig. 5.2): The first part is static. It is the same for all languages. This part contains the GUI and the server backend. The second, dynamic part is everything to do with the definition of the language. When the language developer uploads the language definition from the Lightning plugin, it generates this dynamic part and makes it available for use in the Lightning Web Editor.

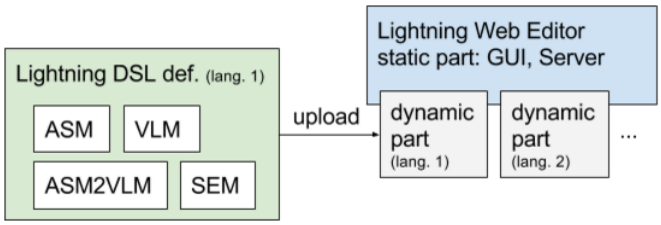

FIGURE 5.2: Static and Dynamic parts of the Lightning Web Editor

5.2 Tiered web application architecture

A typical web application consists of three tiers: a user interface tier also called frontend, an API tier also known as backend and a persistence tier. Fig. 5.3 shows how this typical architecture of a web application [11] is applied for the Lightning Web Editor. In the first tier, we have a graphical user interface to display and edit language instances as well as initiate operations like loading languages and analyzing them or loading and saving language instances. In the second tier, we have the API, which executes the analysis of models and performs model transformations. Finally, in the third tier, we store the data (e.g. the languages, instances and user data).

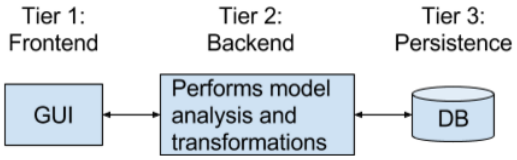

FIGURE 5.3: Tiers of the Lightning Web Editor

5.3 Deployment of the Lightning Web Editor

The Lightning Web Editor is designed to be deployed in two ways. One would be integrated into the Lightning Eclipse plugin for development. The other would be to deploy Lightning Web Editor on a standalone server.

In the following, we present for those two ways how and where each tier is deployed.

5.3.1 Integration into the Lightning Eclipse plugin

To help in the development of languages, the Lightning Web Editor is included in the Lightning plugin. An embedded server hosts the whole web application (all tree tiers). This is shown in fig. 5.4. There is no access to the Lightning Web Editor external to the development machine, and the persistence layer is cleared after every Eclipse session in this deployment scenario.

This means that the language engineer can test how the language is displayed in the Lightning Web Editor, without installing any other software than the Lightning plugin and even if he has no internet access. He also does not need to worry about old versions of the language in his online workspace since the persistence is cleared on every restart of the plugin.

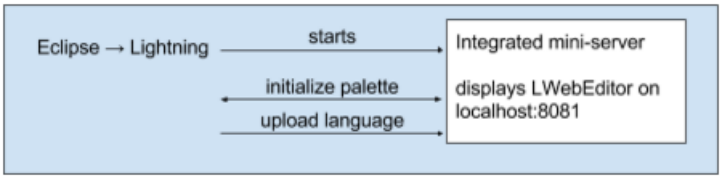

FIGURE 5.4: Embedded server for development

5.3.2 Deployment on a server

In this use case, the Lightning Web Editor is hosted on a server that is external to Eclipse. This server can be internet accessible or in the local network. For better performance and scalability this server should be a dedicated machine.

The Lightning plugin is still used to develop the language. The language engineer can either test the language locally (previous scenario) or upload the language to the Lightning Web Editor to test it. When he is satisfied with the developed language, the engineer can publish the language and make it accessible to all users, who can then use the web editor to create instances of that language.

Fig. 5.5 shows how the different tiers can be hosted on different machines. Here the frontend and backend are hosted on the same server, but the database is hosted on a dedicated machine. Other configurations are also possible, for example where every tier is hosted on its own machine.

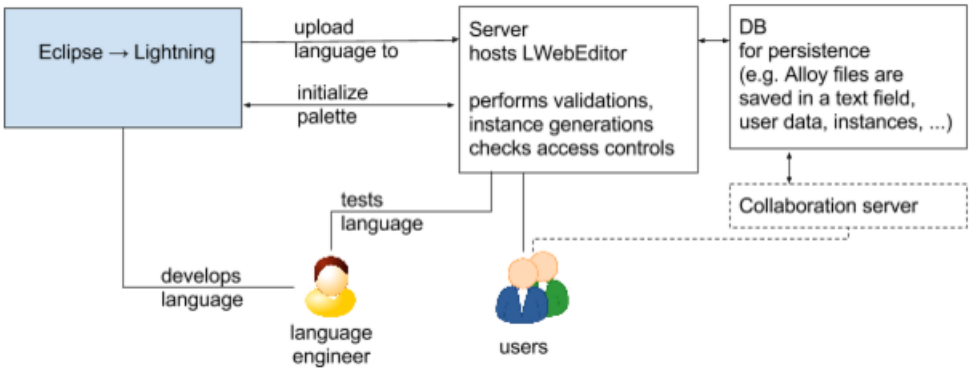

FIGURE 5.5: Deployment on a standalone server

5.3.3 Standalone Lightning web application

Since the aim of the Lightning web application is to make development and use of a DSL as easy as possible, the needs of the developer should not be neglected. An idea would be to offer all the features of Lightning in the web application. This would require the development of a web text editor supporting the editing of Alloy and F-Alloy files, through syntax highlighting and marking of errors in the code. An alternative could be a visual language for F-Alloy, i.e. have a wizard to define the visual representations for

abstract syntax concepts. Other supporting features of Eclipse like version control and file management would also need to be supported by the web editor.

Since these features could mean a signification development effort, they are outside the scope of this thesis.

5.4 Tier 3: Persistence

The Lightning Web Editor needs to store information about Lightning projects, Lightning languages, language instances and user data. This data has to be persistent. The exception to this is deployment scenario 1 (embedded in Lightning plugin) where there is no persistence between Eclipse sessions.

Lightning projects can contain one or more languages. Languages can have any number of instances. Projects are used to organize related languages and to remain constant, while languages can have updates if the language engineer deploys a newer version.

5.5 Tier 2: Backend

The user calls the backend from the Lightning plugin to upload Lightning languages or from the UI to perform the following operations.

- Generation of language instances through analysis of the Alloy ASM.
- Execution of F-Alloy model transformations on a language instance.

When a Lightning language is uploaded to the backend, the backend also transforms the meta-models of the language into a format that is easily digestible by the client.

The backend needs to communicate with both the frontend and the persistence tier. For the communication with the UI, written as a web application, it is best to provide an API using standard web technologies. REST APIs use standard HTTP [12]. It is thus simpler to create clients and develop the API [13].

The idea behind REST is to use unique URLs as resource identifiers, also called endpoints, on which to perform HTTP requests. For example the URL `/api/v1/project` will operate

on all projects, while `/api/v1/project/42d14bd3-d530-4748-9ec5-6b9df5b27203` will operate on the project with ID 42d...7203. The HTTP method of a request indicates what kind of operation is to be executed.

It is specified [14] that the GET method is a safe method (or nullipotent), i.e. it does not change the state of the server. PUT and DELETE methods are idempotent, meaning that the operation will produce the same result no matter how many times it is repeated. PUT, POST and DELETE are unsafe methods. To achieve these properties, it makes sense to implement them accordingly and assign them the semantic of create, read, update, delete (CRUD) resource operations (cf. table 5.1).

HTTP Verb	**CRUD**
POST	Create
GET	Read
PUT	Update/Replace *
PATCH	Update/Modify *
DELETE	Delete

TABLE 5.1: Relationship between HTTP method & resource operation
* Not implemented by the Lightning Web Editor API

To manage access to the persistent data storage the backend has endpoints that perform CRUD-Operations (create, read, update and delete). When executing these methods, the backend in addition to performing the operations on the persistent storage also performs access control and validity checks.

The access control, while necessary for a full online web editor, is left as future work.

5.6 Tier 1: Frontend

The GUI is designed as a single page application, meaning the user should be able to perform all functions without reloading the page.

Fig. 5.6 shows the planned components of the GUI: a menu, palette, overview and canvas.

- The menu bar enables access to common editor functions, as well as the language analysis feature.
- To allow editing, a palette displays all nodes and links that can be dropped on the canvas.

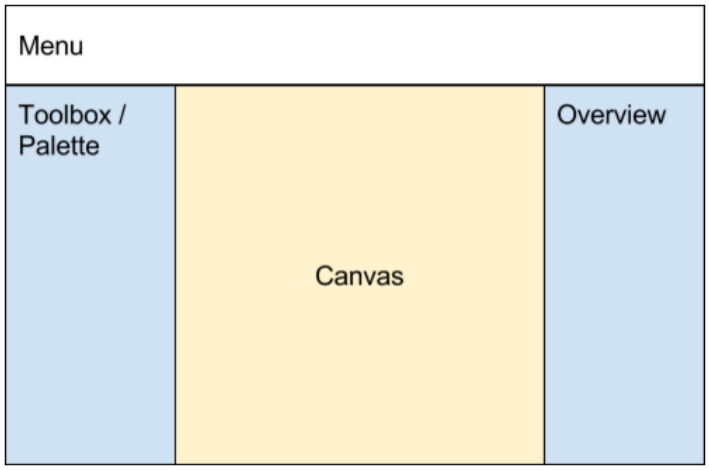

FIGURE 5.6: GUI mockup

- The canvas allows scrolling and zooming, as well as editing of the current instance.
- An overview shows the whole displayed instance with a reference to the current view.

The GUI can work in two modes:

1. Used to debug a DSL, in **ASM mode** the GUI displays the ASM instance of the language instance.
2. In the default **CSM mode**, the GUI displays the CSM instance, i.e. the result of the ASM2VLM model transformation performed on the backend on the ASM instance for users to see.

The architecture of the client underwent refactoring during the implementation. The architecture before the refactoring is described in section 5.6.1 and the architecture after refactoring is described in section 5.6.2.

5.6.1 GUI aspect architecture

Fig. 5.7 shows the architecture of the client before refactoring. All the GUI code responsible for synchronizing the ASM and CSM instances was split into aspects according to whether it was concerned with the ASM instance, the CSM instance, the multiplicity check or with the interaction with GoJS.

The aspects could transmit the data in two directions, from storage to the GoJS library or in the other way. Each aspect could change the data.

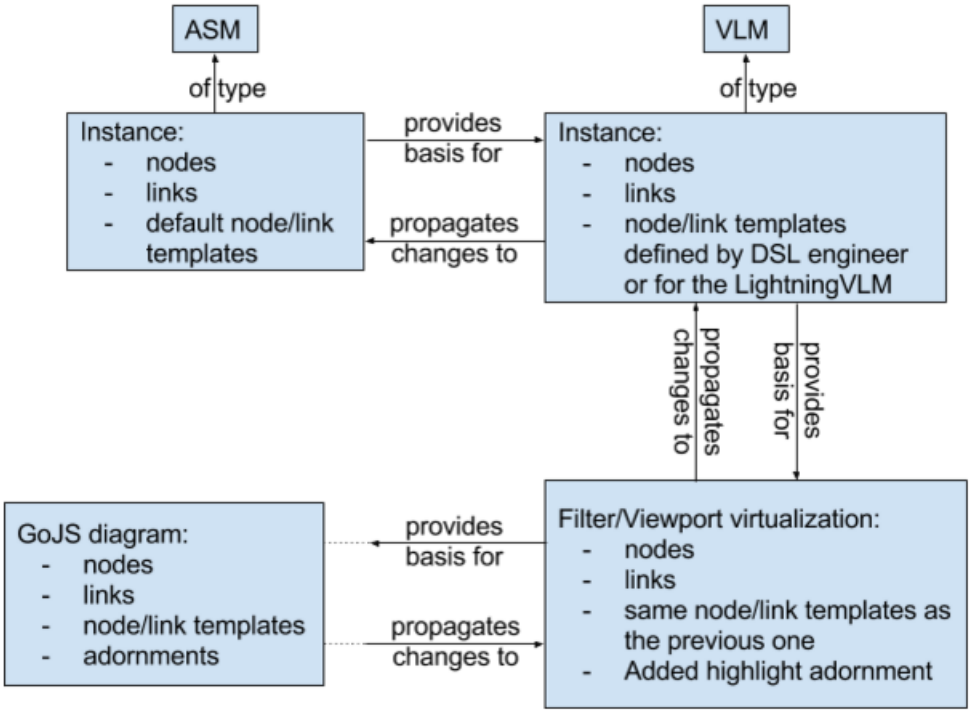

FIGURE 5.7: GUI aspect architecture

5.6.2 GUI MVC architecture

Fig. 5.8 shows the architecture after refactoring. A Model-View-Controller pattern is applied. Special is that there are two controllers. One controller handles the *ASM mode*, while the other handles the *CSM mode*.

The controllers can respond to events triggered by the GoJS library and perform changes to the Lightning instance. The controllers can also set settings of the GoJS library and which instance the GoJS library displays on the canvas. Changes carried out on the canvas are directly performed on the instance displayed, but also call the event handlers of the controller.

Particular care needs to be taken in order to keep the ASM instance and the CSM instance synchronized.

5.7 Validation of language instances

The validation of language instances can be split into a basic validation of types and multiplicities and an extended validation of the constraints in the ASM. Since the basic

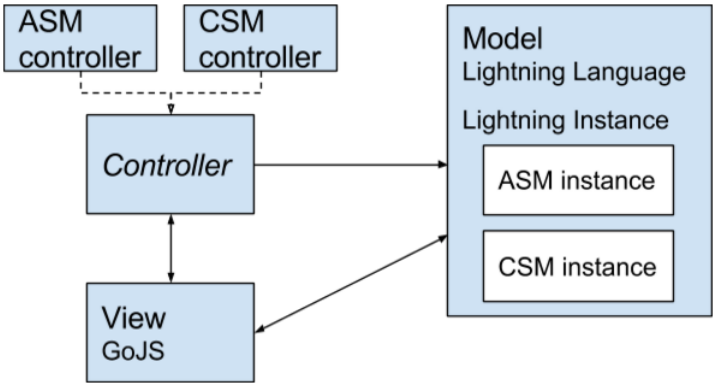

FIGURE 5.8: GUI MVC architecture with multiple controllers

validation is lightweight, it can be performed client-side. The extended validation, however, requires access to the Alloy API implemented in Java and thus needs to be performed server-side.

For extended validation there can be two approaches shown in fig. 5.9 and fig. 5.10 respectively. In the first approach, the CSM instance that is displayed on the client is sent to the server and can be validated against the concrete syntax model. Then an F-Alloy transformation from CSM to ASM is executed on the CSM instance. The resulting ASM instance is validated against the abstract syntax model. Errors found during validation are sent back to the client and displayed.

The second approach sends only the ASM instance back to the server for validation. This requires that there be a synchronization mechanism between the ASM and CSM instances that is executed on the client.

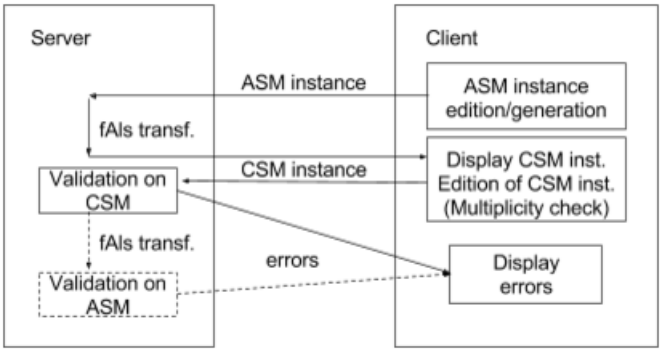

FIGURE 5.9: Validation of ASM instance, by using a CSM2ASM transformation

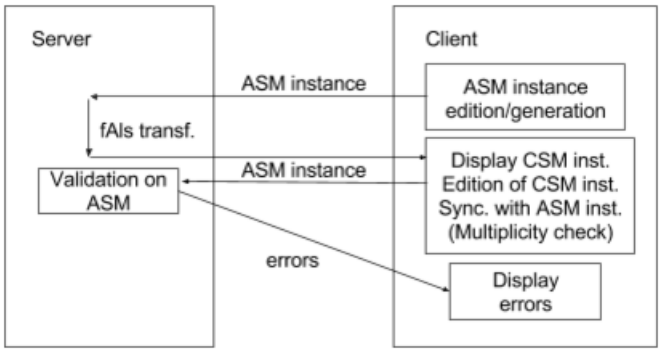

FIGURE 5.10: Validation of ASM instance, by synchronizing ASM and CSM instances on the client

Drawbacks of the first approach are that it will take longer since the model transformation needs to be executed and that the language engineer must provide two model transformations (ASM2CSM and CSM2ASM). Specifying the CSM2ASM transformation could be more complicated than specifying the ASM2CSM transformation and thus places a burden on the language engineer.

Disadvantages of the second approach are that the synchronization mechanism needs to extract enough information from the ASM2CSM transformation to keep the ASM instance up-to-date. This synchronization is a heuristic and thus may not be correct. This can especially be the case if the ASM2CSM contains conditional assignments.

To mitigate this drawback we could imagine that in a second step the ASM2CSM is performed on the suspect ASM instance and the resulting CSM instance is compared with the actual CSM instance. If there is a difference, the user is notified.

We choose the second approach since it will place less burden on the language engineer and the risk mitigation is acceptable.

In the meta-models there can be two types of constraints: global facts, that must hold for all atoms, and signature facts, that must hold for all atoms of a signature. If a constraint is violated, we must alert the user. For a signature fact we can adorn the representation of the violating atom with an error message. Multiplicities can be seen as a type of signature fact. For a global fact, we can have a list of all errors.

The error message must not be cryptic. It should offer an explanation in plain English. For multiplicity constraints this error message is clear (i.e to many/few atoms of type 'X'). However, for other facts this is not so trivial. Signature facts do not have names and global

facts may only have names from a very restricted character set. We must give the language engineer the ability to specify meaningful error messages. For this we can use annotations of the facts, as illustrated in listing 5.1.

```
1 fact acyclic {
2   /// There must not be any cycles in this structure.
3   no d: Dir | d in d.^contents
4 }
```

Listing 5.1: Annotated fact in meta-model

Chapter 6

Implementation

We will start by specifying which technologies were used in our tool. Then we will detail the database schema that is persisted. After elaborating on the implementation of the server-side as well as the client-side, we will conclude this chapter with the challenges that we encountered during the implementation of the tool, as well as some documentation for the tool.

6.1 Chosen technologies

6.1.1 Tier 3: Persistence

Lightning Web Editor uses an SQL database for persistent storage. Section 6.2 describes the database schema.

In the first deployment scenario *Embedded in the Lightning plugin* (cf. 5.3.1) this database also needs to be embedded in the Lightning plugin, which is why the backend has an embedded H2 database that it uses in that case. The H2 database resides entirely in memory and does not persist further than the Eclipse session. This is done since more persistence is not needed to allow testing of the DSLs, the language engineer just needs to upload the language again after a restart of Eclipse. It also ensures a clean database since it removes old languages and only shows the recently uploaded and thus relevant languages.

For the other use cases, the backend connects to an external relational database. Any SQL database would do. We choose a MySQL database since we already had the MySQL server installed. Since the backend and the database are external to the Lightning plugin, the persistence of the data is independent of any Eclipse session.

We did not consider the use of a NoSQL database. The integration of the technology and development of the schema would probably have been more complicated. There was no need for one since the advantages of NoSQL for very high scalability is not a requirement for now.

6.1.2 Tier 2: Backend

Since the backend must use the Alloy API written in Java, it follows that the backend itself is to be written in Java. This choice of language for the backend also allows reusing code written for the Lightning plugin.

For deployment scenario 1, the Lightning Web Editor is to be embedded in the Lightning Eclipse plugin, thus we need to embed a Java application server in the Lightning plugin to run the Lightning Web Editor servlet. Furthermore, the Lightning plugin needs a REST client to be able to deploy languages to the backend.

There exist many mature technologies upon which we can build. Our work in the backend is thus more one of integration than of development. Annex C lists the configuration files of the backend.

6.1.2.1 Choice of embedded web server

Since the backend is written as a servlet, without using any server-specific features, it can run on any of the following servers. However, for the deployment scenario 1, we have to choose a Java application server to be embedded into the Lightning plugin. The choice is between Tomcat, Jetty, JBoss and Glassfish.

- Tomcat is an open-source web server developed by the Apache Software Foundation. While Tomcat can be embedded, it is complicated to do.
- Jetty, an open source project of the Eclipse Foundation, is a lightweight alternative and easily embeddable.

- Oracle's Glassfish is a full Java EE application server and the reference implementation of the Java EE specification.
- JBoss is also a full Java EE application server and thus heavy-weight.

For the Lightning Web Editor we do not need the server to support the full Java EE specification. Thus a lightweight server is better. Since the server needs to be embedded in the Lightning plugin, the choice is Jetty.

6.1.2.2 Choice of REST framework

The selection for a Java REST framework is among the following frameworks: Dropwizard, Jersey, Ninja Web Framework, Play Framework, RestExpress, Restlet, Restx, Spark Framework and Spring boot framework.

Gaić gives a comparison of the most popular Java REST frameworks [15] with advantages, disadvantages and an example each.

The choice was made to use Jersey because it has the clearest annotations, which define path and HTTP method (GET, PUT, POST, DELETE). It is easy to configure. Jersey does not constrain the developer on how to structure the web application. Since it is a very popular framework, there is much documentation on how to integrate it with other libraries. This documentation is necessary since persistence and the Jackson object mapper needed to be integrated into the Lightning Web Editor.

Furthermore, Jersey provides a client to connect to REST APIs, which we can use in the Lightning plugin to upload languages.

6.1.2.3 Persistence

Persistence is supported in Java by JDBC, a standard for database access, and JPA, a specification for object-relational mapping (ORM). JPA is a specification, meaning there is no implementation. The reference implementation is Hibernate, but there are others. Under the hood, Hibernate and all the other JPA implementations write SQL and use JDBC to read and write to the database.

For the connection with the persistence tier, we have the choice of Hibernate, Spring Data JPA, OpenJPA, JDBM and many more.

We will use Spring Data JPA since it uses a very high level of abstraction. We have configured Spring Data JPA to use the reference JPA implementation Hibernate as its engine because it is a very mature technology, widely used in industry. Although we need to integrate the Spring framework for Spring Data JPA, it provides an easier to use interface than Hibernate. It also makes it possible to swap out the JPA implementation, instead of being locked into Hibernate.

With this configuration, we significantly reduce the amount of boilerplate code for persistence, since we only have to provide the entity classes as well as a repository interface for each class (cf. Annex D for an example with the project class). Moreover, the Spring framework will automatically create the database schema, provide the implementation of the repository class and use dependency injection to provide instances of the repository classes.

Another advantage of our solution is that it eliminates the risk of SQL injection attacks since we do not directly use SQL queries. Those queries are managed by Hibernate, which performs all needed bindings of class attributes to table fields and conversion functions on those fields.

A disadvantage is that we sacrifice some flexibility in writing optimized queries, but this is outweighed by the advantages and not needed in our application.

The Entity classes are generated from a UML class diagram of our database schema (cf. 6.2). This generation is done with the GeneSEZ Eclipse plugin [16]. Using the GeneSEZ JPA UML profile [17] we specified additional information for some fields like association between classes, special field types and primary key fields. The concrete Java types for our custom types were given to the GeneSEZ plugin as a configuration.

6.1.3 Tier 1: Frontend

Since the client needs to run in a web-browser without the need to install additional software, the frontend needs to use a few standard technologies, namely HTML5, CSS and JavaScript. To implement the client faster and easier, it is advantageous to use a JavaScript

framework (cf. 6.1.3.1) and JavaScript libraries, since they solve compatibility issues between browsers, provide higher levels of abstraction and more features than JavaScript alone.

For the editor part, it is necessary to use either a JavaScript graphing or diagramming library (cf. 6.1.3.2).

In a supporting role the following JavaScript libraries were used:

- RequireJS 2.2.0 [18] helped split up JavaScript files into modules. It solves dependency management between modules. It eliminates potential global naming conflicts. This helps tackle complexity in bigger software projects.
- Lodash 4.13.1 [19] provides utility functions for a more functional style of programming. These functions include iteration over arrays and objects, filter and mapping functions.
- jQuery 2.2.4 [20] provides utility functions for JSON parsing, and shallow/deep copying of objects.
- Page.js 1.7.1 [21] is a client-side router integrated in anticipation of including UI state (e,g, language/instance loaded or the UI mode) in the URL hash. This URL can then be shared or bookmarked and when revisited the editor can process this information.

Bower 1.7.9 [22] was used to manage the JavaScript dependencies.

To reduce the amount of code written and improve readability the template engine Jade 1.11.0 [23] was used.

Finally grunt 1.0.1 [24] was used to compile the templates, to perform code analysis, host the UI files on a development server and in the last step deploy the files to a Java Web Application project. The Java Web Application project can then be used to manually export a WAR file containing the UI. This WAR file is needed to include the UI in the embedded server in the Lightning Eclipse plugin.

Since the compiled UI files are static HTML, JavaScript and CSS, they can be hosted on any web server.

To get started, yeoman [25] was used with the generator `base-polymer` [26] to generate a skeleton web application integrating RequireJS, jQuery, Page.js, Polymer and Jade.

6.1.3.1 JavaScript framework

We considered to use one of the JavaScript frameworks listed in table 6.1.

Library	**Version**	**Company**
Bootstrap [27]	3.3.6	Twitter
jQuery [20]	2.2.0	
jQuery mobile [28]	1.4.5	
Angular [29]	1.5.7 (v2 in beta)	Google
Polymer [30]	1.0	Google
React [31]	0.14.7	Facebook
react-bootstrap [32]	0.28.3	

TABLE 6.1: JavaScript frameworks

Since the Lightning Web Editor does not involve sophisticated manipulation of the HTML DOM depending on some data, but more manipulating the graphing library, the choice criteria for a JavaScript framework are ease of entry, documentation and modularity.

Bootstrap is mainly a library of UI components and does not provide modularity.

jQuery is not well suited for a modern JavaScript framework since its primary function is to manipulate the DOM. There is no modularity. For UI components jQuery relies on other frameworks, like Bootstrap or jQuery mobile.

According to Kaufman [33] Angular is harder to learn than React. React works better with libraries and is less opinionated.

Polymer uses Google's material design for the UI components, a favorite modern design language. Since Polymer is based on the use of web components, the resulting code is well-structured and modular.

While React does not have UI components, it can be employed together with Bootstrap.

The choice comes down to react-bootstrap and Polymer. We choose to use Polymer since it is more mature than react-bootstrap, and the material design gives the editor more a look of an application.

6.1.3.2 Diagramming library

There exists many different JavaScript diagramming or graphing libraries. The most prominent are shown in table 6.2. The diagramming libraries provide higher levels of abstraction and some libraries even have editor features. We also considered graphing libraries for the case that the available diagramming libraries would have been unsuitable, for one reason or another.

Library	License	Price	Academic license	Technology	Level of Abstraction	Editor
JointJS	MPL	free		SVG	high	No
Rappid	Commercial	1 500,00 €		SVG	high	Yes
MxGraph	Commercial	5500.00 €		SVG	high	Yes
GoJS	Commercial	$ 1,350.00	available	Canvas	high	Yes
Raphael	MIT	free		SVG	low	No
Draw2D	GPL2/Commercial	4,99 / 399 €		SVG	medium	possible
D3	BSD	free		SVG	low	No
FabricJS	MIT	free		Canvas	low	No
PaperJS	MIT	free		Canvas	low	No
JsPlumb	MIT/GPL2	free			medium	No

TABLE 6.2: JavaScript diagramming or graphing libraries

We decided to use GoJS [34] for the following reasons:

- It has an academic license [35]. This license is free but limits the purpose for use as part of a student project, a teaching aid, or Ph.D./dissertation work.
- The license [36] is reasonably priced for other purposes (including research).
- GoJS has a high abstraction level, meaning it is a diagramming library. It works best by using a JavaScript object as a model. This model is then presented as nodes and links on the canvas.
- It has inbuilt capabilities of an editor. This gives us functions such as a palette, overview, copy/paste, layouting and undo/redo.

The version of GoJS used is 1.6.10 with the academic license.

6.2 DB schema

Fig. E.1 in appendix E shows the database schema for the backend of the Lightning web editor. Some parts of the schema are not used since there is no need for them in the retained requirements. These parts are the User and Authentication provider classes as well as the relationships with them. These parts are included in the schema nonetheless in anticipation of future development.

The specification of the external types as concrete Java types is realized as a configuration of the GeneSEZ plugin during the generation of the entity classes.

The Project class represents a Lightning project. It has a name, a description and a flag whether it is publicly accessible.

A project can contain multiple languages. The Language class also has a name and various fields containing the language configuration data. The configuration data is the paths to the different models and transformations, i.e. ASM, CSM, semantic model, semantic step and semantic CSM. During upload, the backend transforms the various meta-models into a format that is easily digested by the UI, i.e. JSON. The result of the transformations to JSON is stored in the data field.

The language engineer should be able to modify a DSL by uploading the Lightning project containing that DSL again. We introduce an intermediary class RawProject between Project and Language to contain the data of each upload. RawProject has a date field containing the timestamp when that version of the project was uploaded to the Lightning web editor and a large field containing the zipped project files from when the project was uploaded. The project class has the relation `current` to RawProject to indicate the newest version. RawProject has a relation with the Language class, which represents all the languages contained in that project in that upload. The API endpoint `/api/v1/language/current` shows all the languages connected to a project through RawProject and the `current` relation.

The Instance class is used to store saved instances. It has the similar meta-data fields than in a file management system (data created/modified, creator, last user to modify it). Specific to the instance class are the meta-model relationship to the Language as well as a data field storing the JSON serialized language instance.

The database tables are created automatically by JPA by setting the property `javax.persistence.schema-generation.database.action` to `create` in the `additionalPropertiesExternal` method of the `PersistenceJPAConfig` class (cf. listing C.5).

6.2.1 Advantages and disadvantages of this schema

Advantages of this schema are that it has little more complexity than is needed to fulfill the requirements and thus easy to implement and use, while still permitting the future implementation of user access controls. It supports all the basic editor features.

A problem of this database schema is the upgrade of instances when their meta-model is updated. The new meta-model would be contained within a newer RawProject with its associated languages. There is no direct way to know for a given instance which the newer language containing the newer meta-model is. Another problem with the upgrade of meta-models is that an instance valid in the old meta-model could become invalid in the new meta-model.

Also not supported is versioning of instances, meaning the ability for one instance to store multiple versions with the possibility to display and restore any version.

Since versioning is not part of the retained requirements these disadvantages are acceptable and we leave their solution as future work.

6.3 Implementation Server-side

After integrating the different technologies, the main work in implementing the backend consisted in handling the API requests. Most requests are simple create, read, update or delete requests for objects in the database. However, some require validation of the parameters and further processing.

The most important API request is the one to upload a Lightning language (shown in fig. 6.1). It requires the storing of the whole DSL definition in the table `RawProject` for the case that instance generation or validation of instances of this language is required. Furthermore, some model transformations need to be executed such that the output format is more easily digested by the client than Java objects.

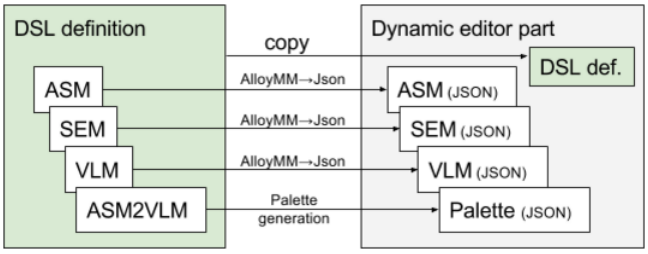

FIGURE 6.1: Upload of a language definition to the editor

The backend is written in Java, while the client runs JavaScript code. The communication between both is done using the JSON format. We use the Jackson library's `ObjectMapper` to convert plain old Java objects (POJO) to and from JSON. What we lack is a transformation to and from Alloy and Lightning specific classes and POJOs. We implemented a transformation from Alloy instances of type `A4Solution`, which return a POJO containing a list of atoms and a list of connection. Another transformation takes that POJO and returns an A4Solution. These transformations are needed since the client works with the JSON format, while the Alloy API uses A4Solution to represent an instance.

The client needs the meta-model to check the types and multiplicities and to display the meta-model in order to give the user a better idea, which objects and which connections are possible. Similar as for the Lightning instances a transformation was added from the Alloy meta-model to a JSON representation. It has as input a `CompModule` and outputs a map of all signatures and connections in the meta-model with their type as index. Each signature has a name, a minimal and maximal multiplicity, a flag whether the signature is abstract, a flag whether it is visible from the main Alloy module (every module can import other modules) and the parent signature if it inherits from another signature. Each connection has a name and minimal and maximal multiplicities for start and end point. The multiplicities depend on whether a signature or connection start/end point was declared `one`, `lone`, `some` or `set`.

Sometimes the client needs to execute a model transformation contained in a language, for example, to transform an ASM instance into a CSM instance. These transformations can be one of the following: ASM2VLM, semantic step or SEM2VLM. They are defined in F-Alloy and can reuse the existing code from the Lightning plugin and the Alloy API as the model transformation executor. These F-Alloy model transformations are exposed as an API endpoint. For this the endpoint takes as input the ID of a language and which of the transformations (ASM2VLM, semantic step or SEM2VLM) are to be performed as

well as a language instance in JSON format. The files of the language are extracted into a temporary folder and the file containing the asked for model transformation is passed onto the model transformation executor. The instance is transformed into an A4Solution which is then passed into the model transformation executor. The result is converted into JSON format and returned.

Similar to the model transformations, we also added an endpoint for analysis of a language. This takes as input the ID of a language, extracts the files of that language into a temporary folder and uses the Alloy API to generate an instance of the ASM of that language. The result is returned in JSON format. It caches the result both server-side and client-side. On the server-side, it is cached to easily obtain the next instance. On the client-side, the cache is done to be able to easily display previous instances without loading them from the server.

The existing model transformation executor by default also returned the original atoms and tuples as well as the bridge and tuples linking origin to result. For scalability, to save bandwidth and processing time on the client we changed this default setting to return only atoms and relations from the output meta-model. Nonetheless, it is important in the UI mode CSM to know which ASM atom is responsible for a certain CSM atom. Thus a data field was added to the output atoms containing the origin information.

In ASM mode the client fills the palette with the types and connections from the meta-model. There is the question what the palette should display in CSM mode. It should be the same types and connections but with their concrete syntax applied. Furthermore, it might not make sense for every type to be included in the palette. To have the biggest flexibility and ease of use, we allow the language engineer to automatically generate a JSON description of the palette. He can then edit this description as he wishes, before deploying the language. When deployed this description of the palette is used in CSM mode to fill the palette.

What this description contains is for each item of the palette, an ASM instance and a CSM instance. When this item is dropped it not only adds the palette CSM instance to the canvas and the current CSM instance, but it also adds the palette ASM instance to the current ASM instance.

6.4 Implementation Client-side

The client was implemented using Polymer as web-components and some JavaScript classes and libraries. The JavaScript classes encapsulate the data of the Lightning language, Lightning instance, the ASM/CSM instance and the Lightning Meta-Model (for ASM, VLM or SEM) as well as the associated methods. The libraries are the following extensions of the GoJS library: a tool to rotate multiple selected elements, a tool to select elements in real-time by dragging, a dynamic layout, which uses different layouts for the content of a group depending on a data attribute of the group, and a command handler that pastes copied elements with an offset and responds to keypresses of the arrow keys by either moving the selection or scrolling the diagram if nothing is selected.

A basic client-side instance validation is the multiplicity check. It uses the JSON meta-model and instance to count the number of atoms of a signature, as well as the number of outgoing and incoming relations of a given type for every atom. It compares these counts with the minimal and maximal multiplicities of the meta-model. If the counts are not in the acceptable range, which is greater or equal to the minimal multiplicity and less or equal to the maximal multiplicity, then an adornment is added to the node representing an atom or the link representing a tuple. This adornment displays an error symbol and hovering over that symbol with the mouse will show an error message, describing which type and multiplicity constraint was violated (shown in fig. 6.2).

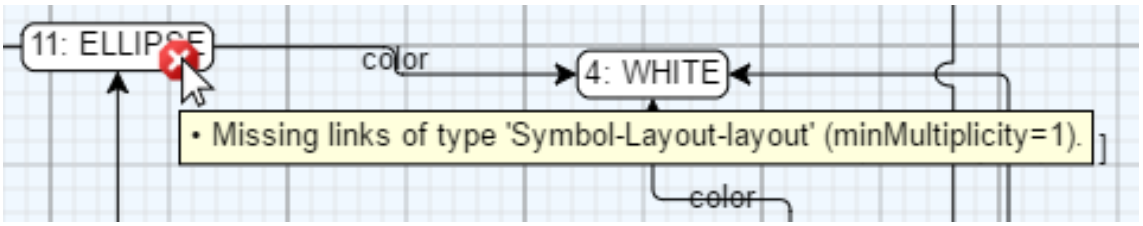

FIGURE 6.2: Screenshot of the error adornment with mouse hover

6.4.1 Synchronization of ASM & CSM instances in UI

A Lightning instance consists of an ASM instance and a CSM instance. The CSM instance can be found by executing the ASM2CSM transformation. Unfortunately, this takes some time. Therefore it is more efficient to do this transformation as few times as possible.

When working in the ASM mode, it invalidates the CSM instance and runs the ASM2CSM transformation to switch into CSM mode. A significant contribution is the synchronization of the ASM and CSM instances while working in the CSM mode. The synchronization needs to consider the following operations:

1. **Adding** an element to the canvas or **copying** an element of the canvas. This adds the ASM and CSM palette instances corresponding to that element to the ASM and CSM instances. While doing this, we need to ensure that keys and labels are unique and that connections and references from CSM to ASM elements are updated to the unique keys and labels.

2. **Deleting** an element from the canvas. This also deletes the corresponding elements from the CSM instances, after unlinking any connections to them, or among them. After that, it deletes any CSM instance, which is not referenced by an element of the CSM instance.

3. **Unlinking** a connection on the canvas. This removes the tuple the connection corresponds to in the ASM instance.

4. **Linking** two elements from the canvas by a connection. This adds a tuple corresponding to the two elements in the ASM instance to the field in the ASM instance which created the connection.

6.4.1.1 Limitations

The problem with the described synchronization algorithm is that it is not guaranteed that the CSM instance is always a valid image of the ASM2CSM transformation given the ASM instance. This can be the case if the transformation contains conditional clauses.

This presents a challenge for validation, because how are we sure that what is displayed as CSM instance is really what is stored as ASM instance and validated. We need to add a step after the validation where we perform the ASM2CSM again and verify that the output is the same as the displayed CSM instance. If it is not the same, we need to alert the user and change the displayed instance to the new one.

6.4.2 Interpretation of the VLM by GoJS

While implementing the editor, we noticed that by default, GoJS renders all nodes and links the same. We need to tell the GoJS library how the nodes should be displayed on the canvas, i.e. their shape, color, border thickness, any labels and so on. The same for links, which need to have defined start and end points, link routing, symbols for the start and end points as well as any labels.

GoJS solves this problem by having the option to provide node templates and link templates. Depending on the value of the `type` attribute of the data object associated with a node/link, the library uses the template of the same name to display that node/link.

Originally Lightning had a fixed VLM the LightningVLM. Now users can define their own VLM and describe how GoJS should interpret the elements of the VLM.

We choose to give this description in JSON format, so that it is easily stored in the DB and that it can easily be utilized by the client. A disadvantage of using JSON format is that it can only express basic data types. We cannot store functions or references to GoJS constants or objects. This is a significant obstacle since GoJS templates need to be instances of the `Node` or `Link` GoJS classes.

To overcome this obstacle we have written an algorithm to transform this description into proper templates. For each template, there is a map which has as key the name of the type and as value a description. This algorithm depends on structure and type of the JSON description to decide which value to return. The rules are listed in table 6.3.

N.B. There is the potential for ambiguity if the first parameter of Create needs to be a Namespace Lookup with no further parameters (e.g.: [['ck.DynamicLayout'], {}]). Without the second parameter {} this would be confused for an Array Escape. Fortunately, `go.GraphObject.make` will ignore any {} parameters. We can thus use such a {} parameter for disambiguation.

The complete description for the default LightningVLM is in appendix F.

Name	Example	Description
Create	['Node', { /* ... */ }]	This will be replaced by the return value of `go.GraphObject.make` called with 'Node' and f({ /* ... */ }) as parameters. f being the reflexive application of this algorithm.
Options	{ key: [/*...*/] }	Reflexively replaces the value of any key, by the result of this algorithm, i.e. [/*...*/] is replaced by f([/*...*/])
Binding	['Binding', { /*...*/ }]	Returns `new go.Binding(...)`. Required parameters are: source, target. Optional parameters: parse and stringify.
Namespace Lookup	['go.Spot.Center']	This will be replaced by the value of go.Spot.Center
Variable Include	['{{shapeConfig}}']	This will be replaced by the return value of f applied to the value of the variable {{shapeConfig}}.
Value	'Text' or 123	Keeps the value of simple types unchanged.
Array Escape	[[4, 2]]	Returns the inner array [4, 2]

Table 6.3: Description of how GoJS interprets the VLM: JSON types → GoJS objects

6.5 Challenges

6.5.1 Integration of technologies

We started out integrating Jersey into our servlet. This worked fine, but then we had to integrate the Spring framework to be able to use Spring data, a very easy to use interface to JPA and Hibernate. The problems were that dependency injection was not working, or the servlet was not reachable, and there was little documentation on how to integrate Spring into Jersey. The solution was to first configure the web application to load Spring and only then to load Jersey to handle requests. This allows for the dependency injection of Spring-managed classes into the endpoint classes.

For the connection to the persistence tier, we used JPA and Hibernate. There are many ways to use JPA and Hibernate, depending on if more or less control over the various aspects of the persistence process is required. This, combined with their maturity and plentiful documentation, made it difficult to find the right way to implement their use, especially since we had to to be able to use one of two data sources depending on if the backend was used embedded in the Lightning plugin or not. Furthermore, in more modern versions of JPA there was a migration away from using XML configuration files and towards using Java classes with annotations for configuration.

To ensure the best possible logging, the Jersey logging filter was changed to also log IPs. Unfortunately, there sometimes was a conflict between Jersey's dependency injection and Spring's dependency injection. Therefore we left the improvement of the logging as a future work and just used basic logging.

The Lightning Eclipse plugin needed some Java dependencies to run an embedded server as well as a REST client to communicate with the backend. The easy way would have been to use Maven, a dependency management and build management tool. However, it proved difficult to configure Maven to output an Eclipse plugin, even using Maven plugins specific to this task. Finally, we created a Maven JAR project, which output a JAR bundling all the dependencies. This JAR was then imported as a dependency in the Lightning Eclipse plugin.

6.5.2 Client architecture

The client's architecture was not sufficiently well designed in the beginning. This had as a consequence that during implementation, we had to refactor some parts of the client.

Initially, all the GUI code responsible for synchronizing the ASM and CSM instances was split into aspects according to whether it was concerned with the ASM instance, the CSM instance, the multiplicity check or with the interaction with GoJS.

This organization of the code was not viable since all data that needed synchronization concerned multiple aspects. So there was the issue to manage the data flow between aspects. Another problem was that every aspect had to handle the different UI modes. Thus every aspect held conditional code.

To simplify the code, it was refactored into two different controllers as well as model classes. The controller is responsible for handling onChange canvas events, the dropping of elements, the loading of a language or an instance and the answering of if two elements can be linked by a connection. One controller handles the ASM UI mode, while the other handles the CSM UI mode. To apply the concrete syntax or not is handled by an adapter in the CSM controller.

A problem of the synchronization was that it needs to keep the keys and labels unique in the ASM and then update any connections in the ASM and CSM instances and references from the CSM to the ASM elements to respect any changed keys and labels. Solving this

issue was the main reason for refactoring since without it development was very slow. After the refactoring, synchronization was solved in a timely fashion.

6.6 Documentation

All current API URI's are prefixed with `/api/v1` to enable smooth upgrades for future versions of the API. In transition phases, clients can still access old versions of the API. Progressively clients upgrade to the newer API version and old API versions are first deprecated and then disabled.

The API is documented in detail in appendix G. For every API endpoint, it gives the title, URL relative to `/api/v1`, the HTTP method, the URL parameters, any data parameters, the success response, the error response and any notes (if any).

The detailed documentation of the GUI of the Lightning Web Editor is included as appendix H. It describes all the actions in the menu bars 'File', 'Edit', 'View', 'DSL' and 'Layout'. The components of the GUI are described. All dialogs and overlays are covered. All keyboard commands and mouse actions are documented.

Chapter 7

Evaluation

In this chapter, we evaluate how well the implemented editor fulfills the requirements. We also validate its effectiveness and ability to scale with two case studies.

7.1 Validation of result

We validate our work by listing the requirements in table 7.1 with the status whether they have been fulfilled, the name of the requirement and relevant comments to each requirement.

Req.	Status	Name & Notes
G1	✓	Specifying domain-specific visual language fulfilled by the Lightning plugin
G2	O	Specifying DSVL tool behavioral aspects fulfilled by extending Lightning plugin, improvements possible
G3-5	/	Critics and Transformations, Human-Centric Tool Interaction & Architecture and Implementation requirements for an ideal meta-tool
C1	/	Specification at run-time future work, planned for a standalone Lightning Web Editor
C2	✓	Multiple languages capability to (re-)load languages from a central repository

Req.	Status	Name & Notes
C3	/	Editor for representations future work
C4	✓	Customizability of symbols allowed by the CSM, implemented in the editor
C5	✓	Customizability of edges allowed by the CSM, implemented in the editor
C6	O	Customizability of edge symbols possible, but difficult, cf. 2)
C7	✓	Implicit relationships allowed by the CSM, implemented in the editor
C8	✓	Shared repositories persistence tier has shared repository (DB)
R1	✓	Canvas with a 2D coordinate system implemented by GoJS, used by the Lightning Web Editor
R2	✓	Graphical representation with a position on the canvas implemented by GoJS, used by the Lightning Web Editor
R3	✓	Representation for each element of the DSL, cf. 1)
R4	✓	Configuration at runtime instances serializable/loadable to, resp. from DB
R5	✓	Customizability of edges, cf. 1)
R6	✓	Start and end location of edges, cf. 1)
R7	✓	Visible edges have a path on the canvas implemented by GoJS, used by the Lightning Web Editor
R8	O	Custom routing of paths possible, but difficult, cf. 3)
R9	✓	Implicit relationships customizable, cf. 1) 4)
R10	✓	Containment, cf. 1)
R11	✓	Labeling elements, cf. 1)
R12	✓	Standard editor functionality implemented by GoJS, used by the Lightning Web Editor
R13	✓	Serialization of representations position information is included in serialization of instances

Req.	Status	Name & Notes
R14	O	Reusability of representations ASM2VLM, VLM and GoJS interpretation of VLM are reusable, but no tool support
R15	/	Wizard for designing representations, future work
W1	✓	Ease of use verified using case studies (cf. 7.2)
W2	/	Collaboration, future work
W3	/	Security, future work
W4	✓	Logging implemented on the backend using slf4j
W5	/	Availability, future work
W6	/	Versioning, future work
W7	O	Validation basic validation (types & multiplicities) implemented

Legend for the status of a requirement:

✓ fulfilled, O partially fulfilled, X unfulfilled, / out-of-scope

TABLE 7.1: Fulfillment of requirements

Additional comments for some requirements are listed hereafter:

1) For the requirements (R3), (R5), (R6) and (R9-11) the Lightning Web Editor takes the description of how the VLM is to be interpreted by GoJS and the ASM2VLM transformation into account to map objects to a graphical representation, assign start and end nodes to links, assign labels and implicit relationships. Since the ASM2VLM is written by the language engineer, it is the engineer's responsibility to make sure that every element of the ASM has a visual representation and how these representations look like.

2) The default Lightning VLM does not allow the customization of edge symbols, but it could be extended so that connectors have a property for the shape of start and end symbol (e.g. either none, arrow or circle, ...). If the description of how the VLM is to be interpreted by GoJS is extended to take this property into account, then (C6) is also fulfilled.

3) (R8) is partially fulfilled since GoJS provides an interface to graph layouting algorithms for the routing of link edges. However, this interface is not exposed by the VLM or the implementation of the interpretation of the concrete syntax. A default routing algorithm is used. To customize the routing of link edges similar adjustment to 2) need to be made to the VLM and the definition of how the VLM is to be interpreted by GoJS. A custom link routing algorithm taking such data properties into account (similar to 4)) needs to be developed.

4) For (R9), implicit relationships based on relative positions, we implemented a custom layout algorithm (i.e. `ck.DynamicLayout`), such that nodes contained within groups are layouted according to the implicit relationship constraints.

In summary, we can say that 20/25 of the retained requirements have been fulfilled and the remaining 5/25 retained requirements have been partially fulfilled.

7.2 Case study

7.2.1 Structured Business Process (SBP)

The Structured Business Process language represents processes of an enterprise so that they can be analyzed and improved [37]. It describes information flows between tasks and control nodes. We have modeled this language in the Lightning Eclipse plugin and uploaded it to the Lightning Web Editor (fig. 7.1).

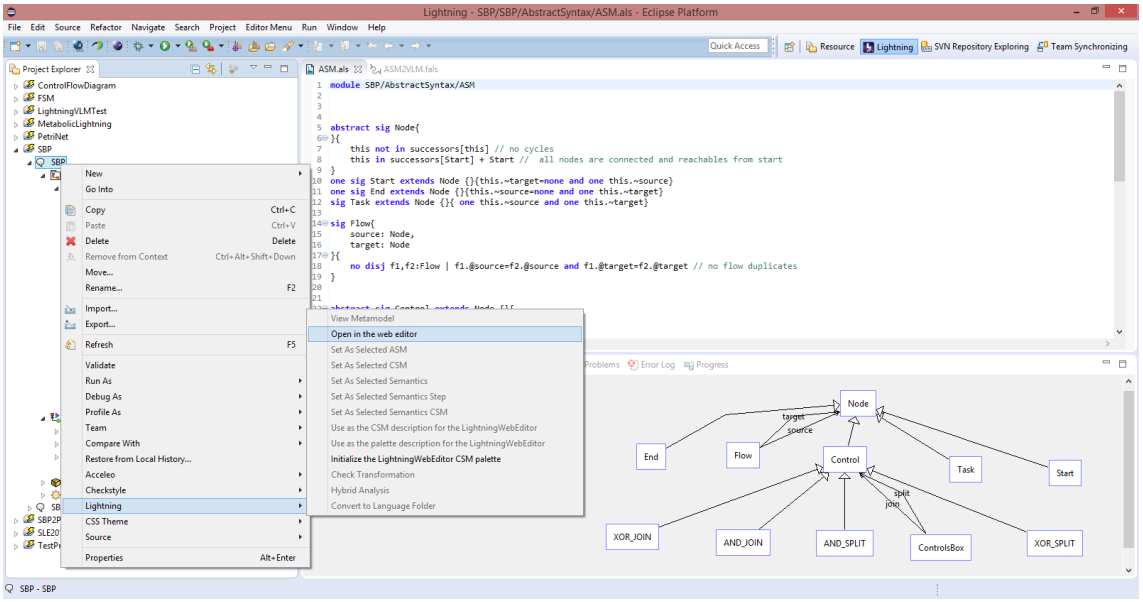

FIGURE 7.1: Deploying SBP to the Lightning Web Editor

After opening the language SBP in the web editor (fig. 7.2) we can quickly generate an instance through analysis by using the 'Generate instance' button in the 'DSL' menu (fig. 7.3).

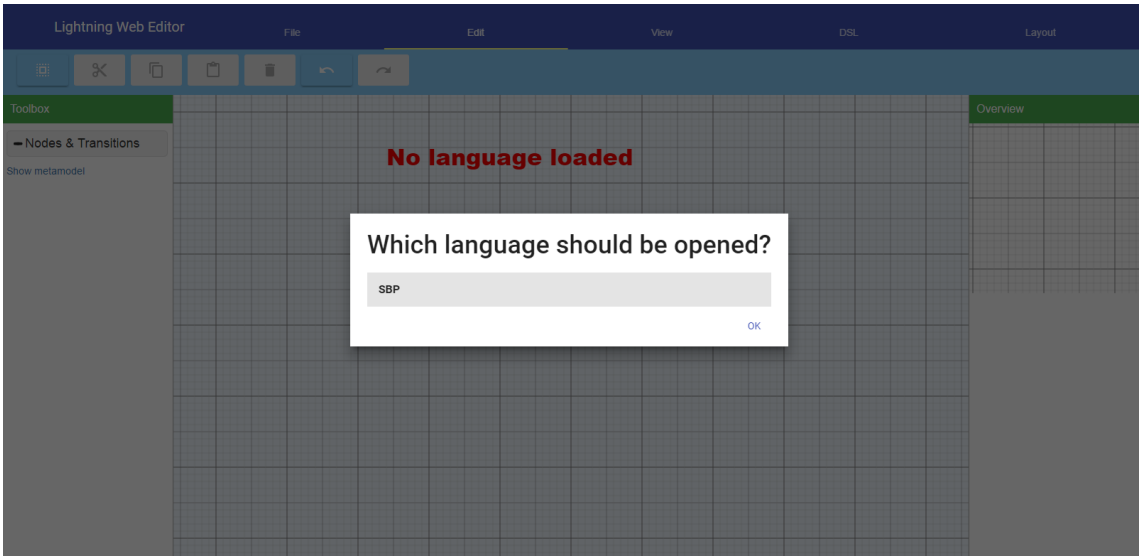

FIGURE 7.2: Opening SBP in the web editor

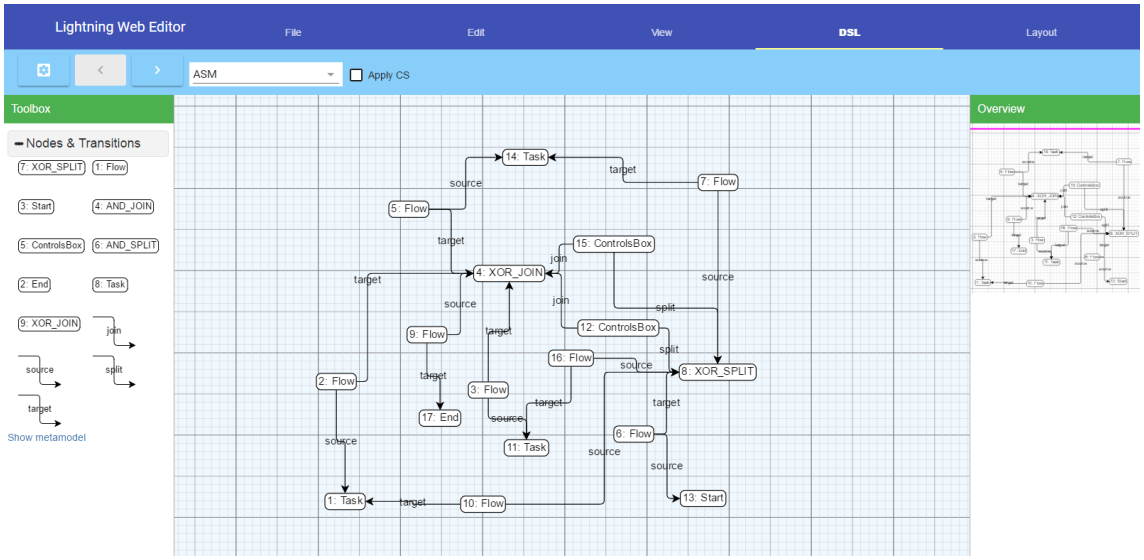

FIGURE 7.3: Case study SBP (abstract syntax)

After applying the concrete syntax and the layered digraph layouting algorithm we obtain the diagram from fig. 7.4. It uses the graphical representations defined in the Lightning definition of the language for the control nodes, the task nodes and start and end node. The flows are also correctly represented as links. Since all links flow from start to end, the layered digraph algorithm gives good positions for all nodes.

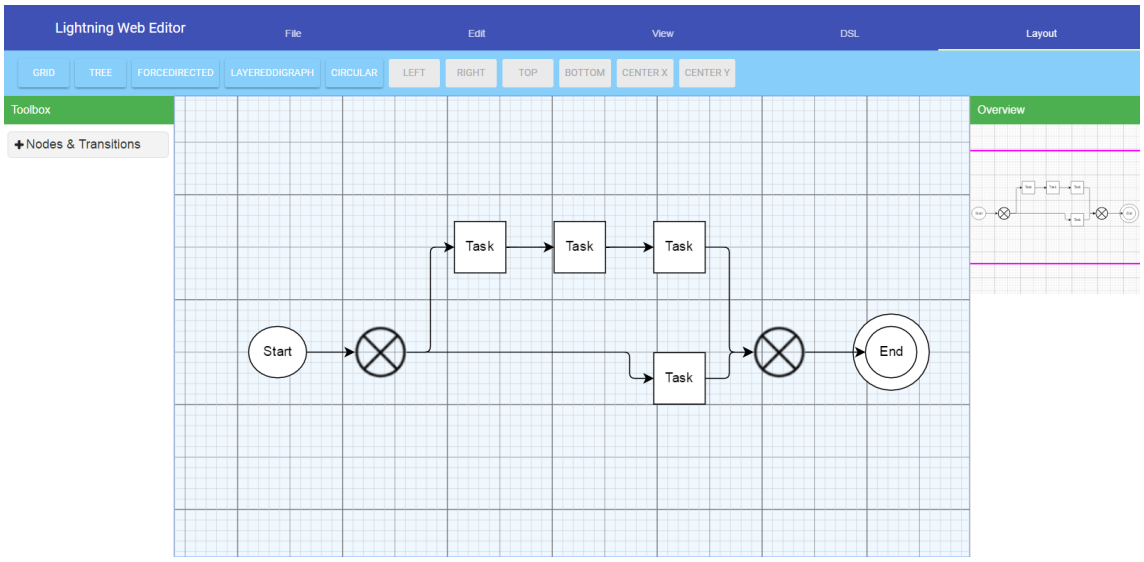

FIGURE 7.4: Case study SBP (concrete syntax)

7.2.2 The Ecoli metabolic network

The Lightning Web Editor can cope with large instances as demonstrated in fig. 7.5. It shows the metabolic network inside the Ecoli bacteria.

This instance has 168 nodes and 360 links in ASM mode, and 862 nodes and 1054 links in CSM mode without applying the concrete syntax and 334 nodes and 360 links with the concrete syntax applied. Applying a layout algorithm takes a few seconds. Zooming operations are not fluid but take less than a second. Scrolling and selecting are performed instantly. These operations are performed faster than in the Lightning plugin because the plugin's implementation for visualization using Draw2D is inefficient.

One significant drawback of this case is that analysis of the language and application of the concrete syntax takes 94 seconds. This time is the same for analysis and application of the concrete syntax in the Lightning plugin and the web editor.

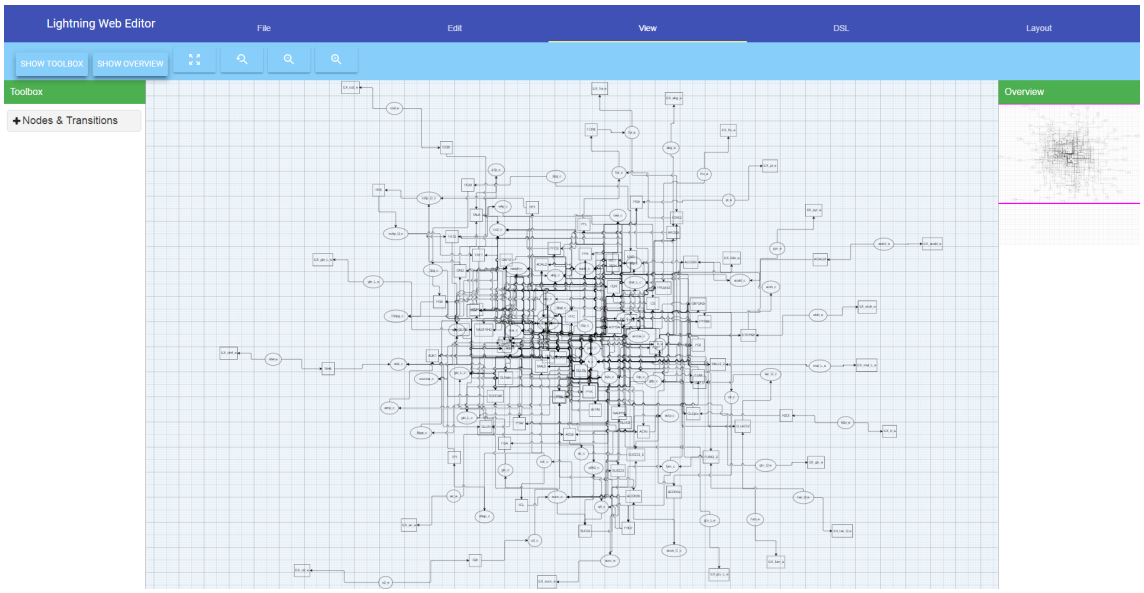

FIGURE 7.5: Case study Ecoli with concrete syntax applied

Chapter 8

Discussion & Related Work

In the first section, we discuss the answers to the research questions.

In the second section, we analyze existing tools for defining visual editors.

8.1 Research questions

The first research question we investigated in this work is whether or not the Lightning language definition contains enough information to generate editors allowing to create and modify language instances through their concrete syntax. An editor can be generated from a language definition if it is possible to generate a relevant palette from the language and if we can keep the ASM and CSM instances synchronized during the editing.

The palette consists of those graphical constructs representing one or a group of ASM elements. The mappings between ASM elements and their graphical representations are given in the ASM2VLM transformations. It is thus possible to obtain, given an ASM element the respective VLM element by simply executing the transformation on an instance containing solely this ASM element we are interested in. As an example, in the SBP case study, the task concept is represented by a rectangle containing text, an instance containing a single task will thus produce a single rectangle containing the expected text. This approach only works when the representation of given ASM elements is independent from the rest of the instance. As an example, consider the SBP case study where the concept of an active node is added with the intent that a task marked as an active node

is to be depicted by a green rectangle while other nodes stay white. If we provide in this case an instance containing a single task, then it will never produce a green rectangle.

It is thus needed to analyze more in depth the F-Alloy transformation to accurately foretell how a given ASM element is to be depicted.

A detailed look at the F-Alloy syntax allowed us to identify which syntactic constructs would lead to ambiguities in getting back the expected ASM instance from the edited VLM instance.

The first syntactic constructs leading to ambiguities are conditional rules that can be declared in value predicates. The image of the ASM concept defined by a mapping whose value predicate contains such a rule depends on a formula ranging over the whole instance (there's no syntax restriction on this formula so it can be any Alloy expression). As it is not possible to predict how this concept should be depicted without taking into consideration the whole instance, the previously described approach cannot be applied successfully to obtain from a single ASM element the expected graphical representation. The second syntactic constructs leading to ambiguities are indexed field references, that is, fields references which are suffixed by a box containing an Alloy expression. Should this expression be anything but the concept itself, then we have again introduced a dependency on a relation with other elements of the instance.

To correctly generate a palette component from ASM elements whose mapping feature one of those two ambiguous constructs, we propose the following solutions:

- Let the language engineer propose its own representation for the problematic ASM elements.
- Generate valid Alloy instances from the ASM and feed them to the ASM2VLM transformation with intent to identify those VLM elements that can be generated from the problematic ASM elements.

The effectiveness of the second solution proposed relies on the small scope hypothesis, as we want to claim that all possible VLM elements generatable from those problematic ASM elements can be found in small VLM instances obtained (via the ASM2VLM transformation) from small ASM instances. One would still need to identify which of the thus obtained VLM elements are desired palette items and which are to be composed in a single palette

item. Thus both solutions are semi-automatic. The second, requiring alloy analysis, is probably more time-consuming.

We can thus conclude already that palette generation fails, in some cases, to be performed fully automatically given a language definition. In those cases, the language engineer should define a palette item (one or a combination of VLM elements) to represent a given ASM element.

Considering the consistency between ASM and VLM instances, we have identified a set of operations performed on the VLM instances that are non-trivial to propagate to the ASM instance. Here is a list of the operations we want to discuss:

1. (Dis-)Connecting two VLM elements
2. (De-)Composing several VLM elements

In both cases, it is needed to identify which field is concerned by the connection or composed VLM elements. In the case of the connection, this information is generally present in guard predicates of mappings producing connectors and in the strict rules assigning values to the source and target of those connectors. In the case of composed VLM elements, this information is present in guard predicates of VLM elements, in the composition of other elements or in loose rules assigning those elements to composition of other VLM elements. As language engineers are free to write any Alloy expressions in those guards and rules, it can become unexpectedly hard to retrieve the field from those place.

We can get rid of those non-trivial cases in the synchronization between ASM and VLM instances by:

- Refining the syntax of F-Alloy so that the field(s) associated with a connection or composition are explicitly given.
- Letting the language engineer explicitly specify in a separate formalism which field(s) originate a connection or composition.

We have thus come to the conclusion that in some cases it is possible to generate editors from Lightning language definitions. In cases its is not possible due to ambiguities causing the palette automatic generation or the consistency between ASM and VLM instances to

fail, we have identified which information should be provided by the language engineer and provided some workarounds to facilitate these inputs.

8.2 Existing editors

8.2.1 AToMPM

AToMPM, short for "A Tool for Multi-Paradigm Modeling" [38], is a standalone web-based framework for generating DSL web-based tools. AToMPM is reflexive in the sense that the meta-models, menu items and operations possible in the tool are defined using AToMPM.

In AToMPM [39] everything is either a model or a model transformation. To develop a DSL first a model conforming to the SimpleClassDiagram meta-model (a class diagram meta-model defined in AToMPM) is created which is then promoted to be the meta-model of the new DSL. Similarly, a model conforming to the ConcreteSyntax meta-model is created and then promoted to be the meta-model of the concrete syntax of the new DSL. All editor operations are specified as model transformations.

A description of the new DSL contains a one-to-one mapping between the abstract syntax elements and the elements of the concrete syntax. Fig. 8.1 shows a simplified view of this meta-model.

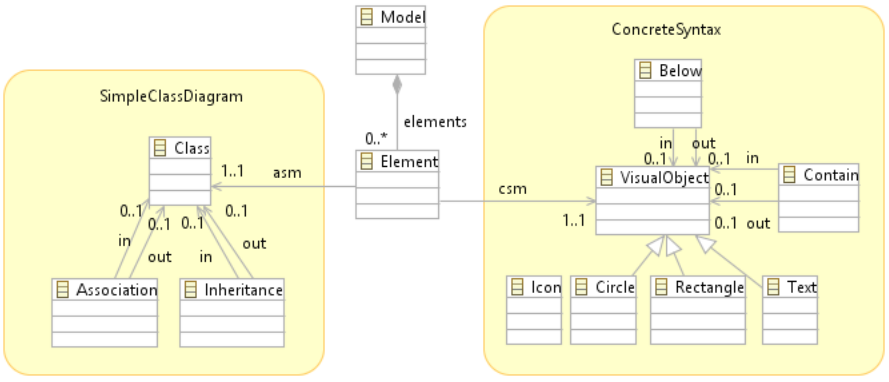

FIGURE 8.1: Simplified AToMPM meta-model

AToMPM is very similar to the tool developed in the context of this thesis as it is also a web-based editor relying on model driven approaches (models and model transformations) to define, edit and represent DSLs. While every operation users can perform on AToMPM

generated editors are to be specified, Lightning generated editors support most of those operations by default.

8.2.2 GMF / Eugenia

The Graphical Modeling Framework (GMF) [40] is an Eclipse plugin that provides a model-driven approach to generating graphical editors in Eclipse. Eugenia is an Eclipse plugin that builds on top of GMF to simplify the task of generating editors.

A simplified view on the types defining visual editors according to GMF is shown in fig. 8.2 [41]. To generate an editor with GMF, an Ecore meta-model is needed. From this meta-model the source code for editing models conforming to this meta-model is generated, as well as three models describing the available tools in the editor, the graphical elements in the editor and the mappings between tool, diagram element and meta-model element. It is important to note that at most one type or relation in the meta-model is mapped to one node or connection on the canvas. From these three models, a code generation model is created, which finally generates the code for the editor. These models can be edited to customize the editor.

The generation of these models is only a starting point. The customization is almost always needed to obtain a working editor. This customization is complicated and error prone since the engineer requires a good understanding of the complex GMF meta-model and the editor models use many cross-references between each other and the Ecore meta-model. Should the Ecore meta-model change, the developer either needs to propagate these changes manually to the editor models or regenerate the editor models and reapply the customization.

To handle these shortcomings of GMF, projects like Eugenia [42] were created. Eugenia uses annotations to the Ecore meta-model elements to describe how they should be displayed in the editor. Eugenia takes care of generating all intermediary models needed by GMF to generate the editor code. Limitations of Eugenia are that due to the annotations abstract syntax definition and concrete syntax definition as well as information necessary for the creation of the editor are mixed together in the meta-model detracting from its primary purpose (abstract syntax definition). Also, Eugenia allows access to only a subset of GMF features, for more control intermediary model transformations can be run on the generated GMF models.

One of the biggest drawbacks of Eugenia is that if this increased control is needed, then the developer still needs a sound understanding of GMF to write his own post-processing transformations. Thus Eugenia is more of an extension that provides a different way of working with GMF for complex editors.

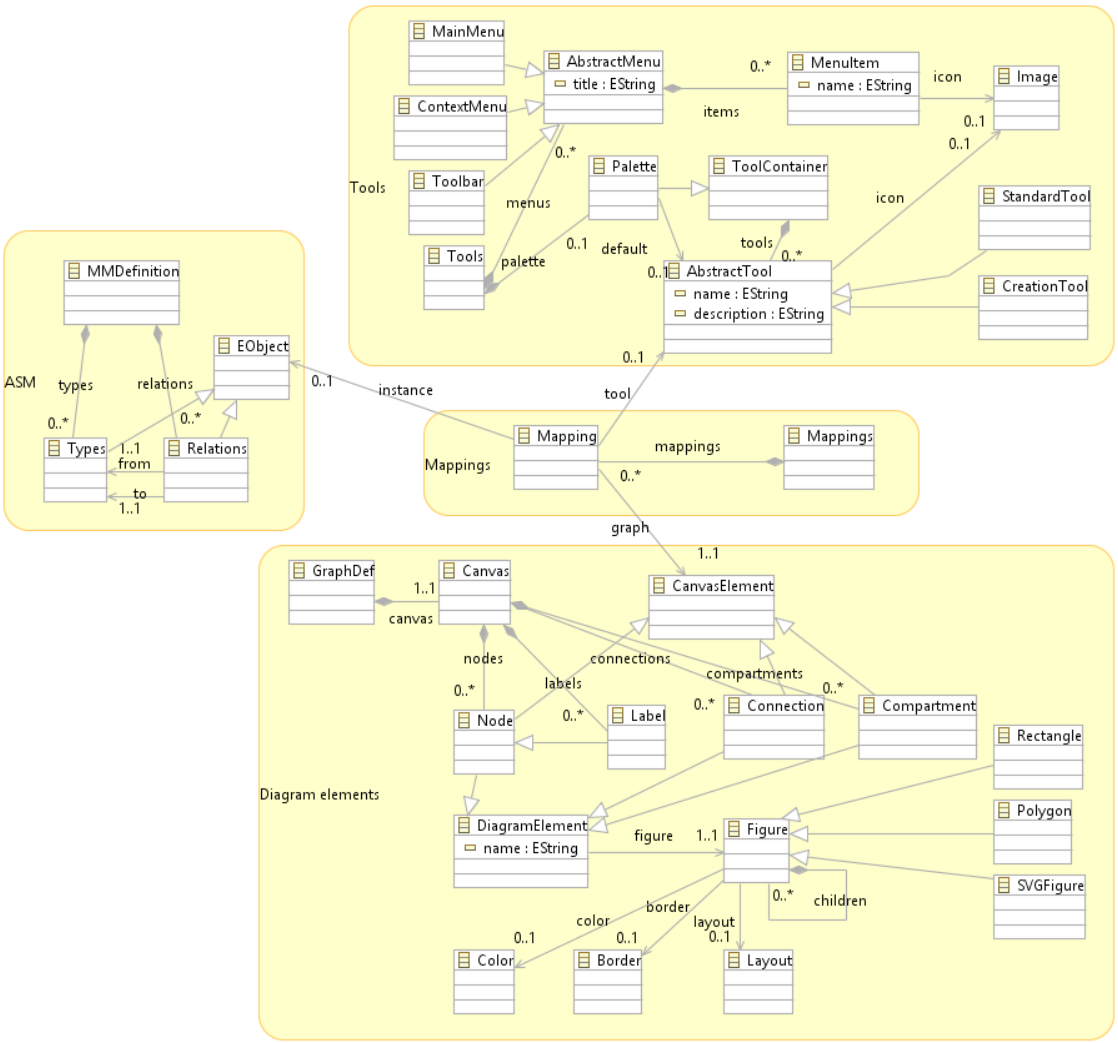

FIGURE 8.2: Simplified GMF meta-model

8.2.3 Marama

The Marama project is an Eclipse meta-toolset [43] that tries to fulfill all general requirements of a meta-tool (G1-4). It allows the specification of a domain-specific language and an editor for it.

Although Marama also has a web editor for its models [44], this editor is composed of a server component and a client component, mostly static HTML with some JavaScript.

The client triggers requests to the server when HTML buttons are pressed or mouse clicks on the diagram occur. The server processes how the model should be displayed in the UI and server-side transformations effect any changes to the model. Only scrolling and zooming is performed client-side. The Lightning Web Editor distinguishes itself from this Marama web editor by being more responsive, performing more operations client-side and supporting more editor functions.

On the Marama website [45], it is announced that development of Marama was ended in 2012. While no reason is cited, we can speculate that it could be because the scope of the project was bigger and the requirements more complex than the development team could tackle. Other causes could be too many bugs in the program or any number of common causes for project failure.

Chapter 9

Conclusion

9.1 Contributions

We have implemented a working web-editor for domain-specific languages specified in Lightning, called the Lightning Web Editor. This editor integrates many technologies used in web applications as well as existing code from the Lightning plugin that was adapted for use in this editor. We have extended the Lightning plugin to generate a graphical web editor for a Lightning language.

Instances edited in the Lightning Web Editor are type checked against their meta-model. As a first step the validation of instances only checks type and relation multiplicity constraints. We have though detailed in 5.7 approaches to fully support validation.

It is possible to edit instances in the editor with the concrete syntax applied. A language describing how the VLM should be displayed by GoJS was developed. For the default LightningVLM, this description was provided to the tool, but the option remains for language engineers to develop their own VLMs and describe how they should be displayed. While editing concrete syntax instances, an algorithm keeps the ASM and CSM instances synchronized.

While the deployment of the Lightning Web Editor is premature since critical security controls are missing, the design of many security features has been done.

9.2 Limitations

The limitations of the Lightning Web Editor are that it is dependent upon the Lightning Eclipse plugin for the specification of the DSLs since language engineers still need to install and use Eclipse and the Lightning plugin.

There is no visual editor for representations, which is a significant impediment for new language engineers since they either need to learn F-Alloy to write their ASM2VLM transformation and they need to be familiar with the given LightningVLM as output or they need to to find out how to describe the representations of their VLM in JSON.

The synchronization algorithm does not have enough information to copy elements of the concrete syntax obtained through the ASM2VLM transformation. These elements can still be deleted and new elements from the palette added.

At the moment there is no user access control, so everyone has access to all the languages and instances. If the Lightning Web Editor is to be used in the cloud, user access control needs to be implemented.

The user input to the API only has limited validation, i.e. most parameters sent to the backend are taken as is. Only in some cases, we check that references are to existing IDs of the correct type for that reference, or that a project has the required meta-models. This represents a security vulnerability since cross-site scripting might be possible.

There is no throttling of the API access, i.e. a restriction of the number of calls to the backend per user per time unit, so a malicious user can easily perform a denial of service attack by requesting many analysis or transformations.

9.3 Future Work

As future work we leave the implementation of the postponed and partially fulfilled requirements.

Complete validation of instances is to be implemented, such that on request from the user the instance is sent to the backend and checked against all meta-model constraints. Any elements violating a constraint need to be displayed in the editor with an error icon and a message describing the violated constraint.

We will extend the Lightning VLM and the description of how that VLM should be interpreted by GoJS to be able to customize edge symbols in the ASM2VLM transformation. Similar extensions of VLM and description are required for the capability to specify the link routing algorithm for every link.

A wizard or editor for the visual representations of abstract syntax concepts would be very useful since it will provide a more intuitive interface to define the CSM than in the current editor.

The challenges of the synchronization between CSM instance and ASM instance shows that this is an area where work is still required. Either the VLM and ASM2VLM are reworked to enable a bi-directional mapping between ASM and VLM like it is used in the other existing tools, or the synchronization algorithm needs to be enhanced.

Implementing security is a critical requirement for a web application.

Versioning of languages and instances is a step towards making the editor have features sought in industry.

The user should be allowed to specify the scope for the instance generation and a default layout algorithm to be applied to the generated instances.

Making the tool stand-alone, will require the implementation of editor for Alloy and a file management solution for the project files.

At the moment it is possible for multiple users to load and save instances, but not to work in real time on the same instance. Enabling this will further collaboration.

Other interesting future developments could be the implementation of view filters, where depending on a user query some of the nodes and links are either hidden or highlighted.

Also, a more thorough validation of the usability requirement is planned with a user experiment.

Appendix A

Lightning license

Appendix B

LightningVLM

Figure B.1 shows the meta-model of the default Lightning VLM.

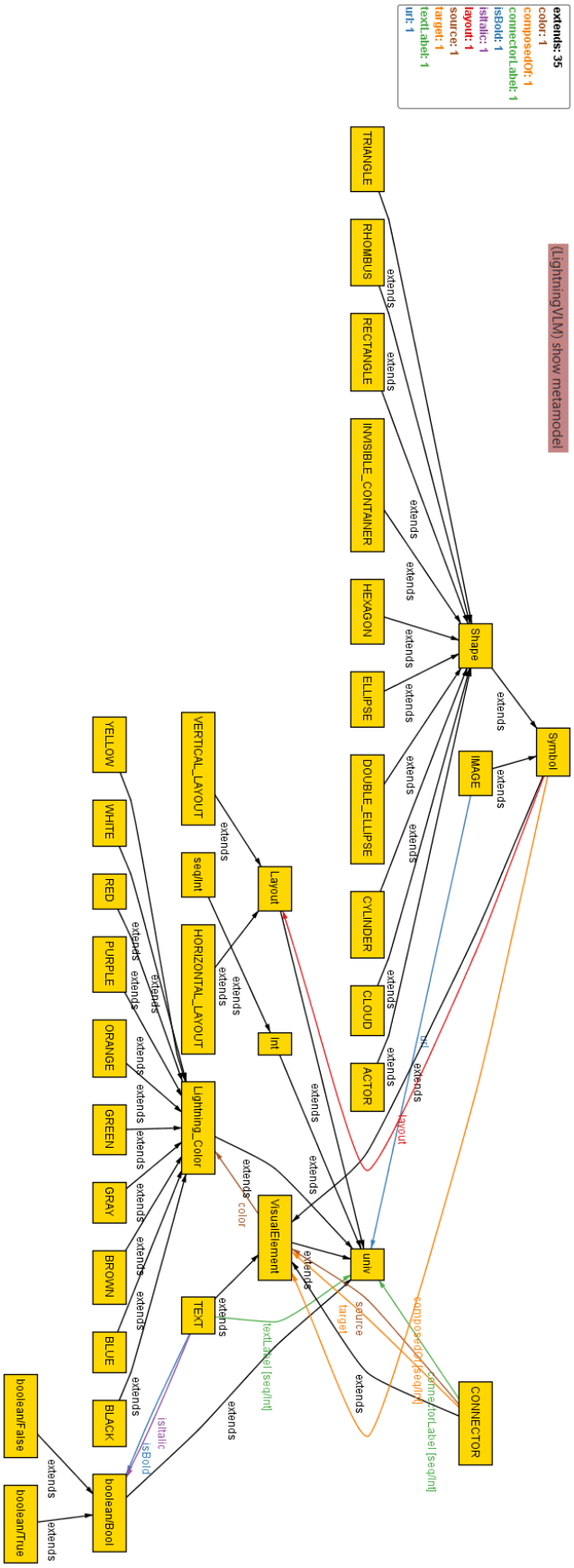

FIGURE B.1: The Lightning Visual Language Model

Appendix C

API configuration files

Listing C.1 for the file `pom.xml` defines the Maven dependencies of the backend. All dependencies except for the Alloy API are Maven dependencies. The Alloy API is included as a JAR in the `lib` folder.

Listing C.2 for the file `web.xml` is the main configuration file for the web application. It specifies how the context should be configured (by processing `applicationContext.xml`), how many servlets there are (one) and how they are configured, i.e. which Java class to load and any parameters (here the configuration is the class in `App.java`).

Listing C.3 for the file `applicationContext.xml` shows how the context is configured to load the Spring framework, to enable dependency injection, to look for more configuration in annotated Java classes and to scan the classes in the packages `lu.uni.lightning.webapi` and `lu.uni.lightning.webapi.endpoints` for more configuration (i.e. the `PersistenceJPAConfig` class) and components (the endpoints are components in order to receive repositories through dependency injection).

Listing C.4 shows the file `App.java`. This class configures the REST framework Jersey. It uses a filter to process logging. The Jackson library is used to map JSON to the method parameters of the endpoints and the returned objects of endpoint methods to JSON. Any exception thrown within the endpoint methods are processed by the `GenericExceptionMapper`. It scans the package `lu.uni.lightning.webapi.endpoints` for endpoint classes. See listing D.3 for an example of an endpoint.

One other configuration was for testing: A security feature against cross-site scripting in web browsers disallowed requests to URLs not on the same domain (even if everything except the ports matches), except if the API explicitly allowed it. To allow this the `AccessControlAllowOriginFilter` was added that favorably responded to requests whether a client was allowed access from origin `localhost`.

Listing C.5 for the file `PersistenceJPAConfig.java` shows how the persistence is configured. It uses Spring Data JPA with Hibernate as engine and connects either to an embedded H2 database or an external MySQL database. It also tells the entity manager to scan the package `lu.uni.lightning.webapi.system` for entity classes.

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
       /2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
       /maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4    <groupId>lu.uni.lightning</groupId>
5    <artifactId>LWebEditorAPI</artifactId>
6    <version>0.0.1-SNAPSHOT</version>
7    <packaging>war</packaging>
8    <name>LWebEditorAPI</name>
9    <description>Contains the server API for the Lightning Web Editor</description>
10   <organization>
11     <name>University of Luxembourg</name>
12     <url>www.uni.lu</url>
13   </organization>
14
15   <properties>
16     <jersey.version>2.22.2</jersey.version>
17     <jackson.version>2.7.3</jackson.version>
18     <spring.version>4.2.5.RELEASE</spring.version>
19     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
20   </properties>
21
22   <build>
23     <finalName>${project.artifactId}</finalName>
24     <sourceDirectory>src</sourceDirectory>
25     <plugins>
26       <plugin>
27         <artifactId>maven-war-plugin</artifactId>
28         <version>2.4</version>
29         <configuration>
30           <warSourceDirectory>WebContent</warSourceDirectory>
31           <failOnMissingWebXml>false</failOnMissingWebXml>
```

```
32          </configuration>
33        </plugin>
34        <plugin>
35          <groupId>org.apache.maven.plugins</groupId>
36          <artifactId>maven-compiler-plugin</artifactId>
37          <version>2.5.1</version>
38          <inherited>true</inherited>
39          <configuration>
40            <source>1.7</source>
41            <target>1.7</target>
42          </configuration>
43        </plugin>
44      </plugins>
45    </build>
46
47    <dependencyManagement>
48      <dependencies>
49        <dependency>
50          <groupId>org.glassfish.jersey</groupId>
51          <artifactId>jersey-bom</artifactId>
52          <version>${jersey.version}</version>
53          <type>pom</type>
54          <scope>import</scope>
55        </dependency>
56        <dependency>
57          <groupId>org.springframework</groupId>
58          <artifactId>spring-framework-bom</artifactId>
59          <version>${spring.version}</version>
60          <type>pom</type>
61          <scope>import</scope>
62        </dependency>
63      </dependencies>
64    </dependencyManagement>
65
66    <dependencies>
67      <dependency>
68        <groupId>org.glassfish.jersey.ext</groupId>
69        <artifactId>jersey-spring3</artifactId>
70        <exclusions>
71          <exclusion>
72            <groupId>org.glassfish.hk2.external</groupId>
73            <artifactId>bean-validator</artifactId>
74          </exclusion>
75        </exclusions>
76      </dependency>
77      <dependency>
78        <groupId>org.apache.commons</groupId>
79        <artifactId>commons-dbcp2</artifactId>
```

```
80        <version>2.1.1</version>
81      </dependency>
82      <dependency>
83        <groupId>org.apache.commons</groupId>
84        <artifactId>commons-pool2</artifactId>
85        <version>2.4.2</version>
86      </dependency>
87      <dependency>
88        <groupId>commons-codec</groupId>
89        <artifactId>commons-codec</artifactId>
90        <version>1.10</version>
91      </dependency>
92      <dependency>
93        <groupId>org.springframework</groupId>
94        <artifactId>spring-core</artifactId>
95      </dependency>
96      <dependency>
97        <groupId>org.springframework</groupId>
98        <artifactId>spring-context</artifactId>
99      </dependency>
100     <dependency>
101       <groupId>org.springframework</groupId>
102       <artifactId>spring-beans</artifactId>
103     </dependency>
104     <dependency>
105           <groupId>org.springframework</groupId>
106           <artifactId>spring-web</artifactId>
107     </dependency>
108     <dependency>
109       <groupId>org.springframework</groupId>
110       <artifactId>spring-aspects</artifactId>
111     </dependency>
112     <dependency>
113       <groupId>org.springframework</groupId>
114       <artifactId>spring-orm</artifactId>
115     </dependency>
116     <dependency>
117       <groupId>org.springframework.data</groupId>
118       <artifactId>spring-data-commons</artifactId>
119       <version>1.11.4.RELEASE</version>
120     </dependency>
121     <dependency>
122       <groupId>org.springframework.data</groupId>
123       <artifactId>spring-data-jpa</artifactId>
124       <version>1.9.4.RELEASE</version>
125     </dependency>
126     <dependency>
127       <groupId>org.hibernate</groupId>
```

```
128       <artifactId>hibernate-core</artifactId>
129       <version>5.1.0.Final</version>
130     </dependency>
131     <dependency>
132       <groupId>org.hibernate.common</groupId>
133       <artifactId>hibernate-commons-annotations</artifactId>
134       <version>5.0.1.Final</version>
135     </dependency>
136     <dependency>
137       <groupId>org.hibernate</groupId>
138       <artifactId>hibernate-entitymanager</artifactId>
139       <version>5.1.0.Final</version>
140     </dependency>
141       <dependency>
142         <groupId>mysql</groupId>
143         <artifactId>mysql-connector-java</artifactId>
144         <version>5.1.32</version>
145       <scope>runtime</scope>
146     </dependency>
147     <dependency>
148       <groupId>javax.servlet</groupId>
149       <artifactId>javax.servlet-api</artifactId>
150       <version>3.1.0</version>
151     </dependency>
152     <dependency>
153       <groupId>org.slf4j</groupId>
154       <artifactId>slf4j-api</artifactId>
155       <version>1.7.20</version>
156     </dependency>
157     <dependency>
158       <groupId>org.slf4j</groupId>
159       <artifactId>slf4j-log4j12</artifactId>
160       <version>1.7.20</version>
161     </dependency>
162     <dependency>
163       <groupId>org.glassfish.jersey.containers</groupId>
164       <artifactId>jersey-container-servlet</artifactId>
165     </dependency>
166     <dependency>
167         <groupId>org.glassfish.jersey.media</groupId>
168         <artifactId>jersey-media-json-jackson</artifactId>
169             <exclusions>
170             <exclusion>
171               <groupId>com.fasterxml.jackson.core</groupId>
172               <artifactId>jackson-annotations</artifactId>
173             </exclusion>
174             <exclusion>
175               <groupId>com.fasterxml.jackson.jaxrs</groupId>
```

```
176             <artifactId>jackson-jaxrs-base</artifactId>
177           </exclusion>
178           <exclusion>
179             <groupId>com.fasterxml.jackson.jaxrs</groupId>
180             <artifactId>jackson-jaxrs-json-provider</artifactId>
181           </exclusion>
182       </exclusions>
183   </dependency>
184   <dependency>
185       <groupId>org.glassfish.jersey.ext</groupId>
186       <artifactId>jersey-bean-validation</artifactId>
187   </dependency>
188   <dependency>
189     <groupId>com.fasterxml.jackson.core</groupId>
190     <artifactId>jackson-annotations</artifactId>
191     <version>${jackson.version}</version>
192   </dependency>
193   <dependency>
194     <groupId>com.fasterxml.jackson.jaxrs</groupId>
195     <artifactId>jackson-jaxrs-base</artifactId>
196     <version>${jackson.version}</version>
197   </dependency>
198   <dependency>
199     <groupId>com.fasterxml.jackson.jaxrs</groupId>
200     <artifactId>jackson-jaxrs-json-provider</artifactId>
201     <version>${jackson.version}</version>
202   </dependency>
203   <dependency>
204     <groupId>com.fasterxml.jackson.datatype</groupId>
205     <artifactId>jackson-datatype-hibernate5</artifactId>
206     <version>${jackson.version}</version>
207   </dependency>
208   <dependency>
209       <groupId>com.h2database</groupId>
210       <artifactId>h2</artifactId>
211       <version>1.4.191</version>
212   </dependency>
213 </dependencies>
214 <repositories>
215       <repository>
216           <id>spring-releases</id>
217           <name>Spring Releases</name>
218           <url>https://repo.spring.io/libs-release</url>
219       </repository>
220       <repository>
221           <id>org.jboss.repository.releases</id>
222           <name>JBoss Maven Release Repository</name>
```

```
223             <url>https://repository.jboss.org/nexus/content/repositories/releases</
    url>
224         </repository>
225     </repositories>
226 </project>
```

Listing C.1: API configuration file pom.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.
       sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0"
       >
3    <display-name>lu.uni.lightning.webapi</display-name>
4    <listener>
5      <listener-class>org.springframework.web.context.ContextLoaderListener</listener
       -class>
6    </listener>
7    <context-param>
8      <param-name>contextConfigLocation</param-name>
9      <param-value>classpath:applicationContext.xml</param-value>
10   </context-param>
11   <servlet>
12     <servlet-name>Jersey REST Service</servlet-name>
13     <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
14     <init-param>
15       <param-name>javax.ws.rs.Application</param-name>
16       <param-value>lu.uni.lightning.webapi.App</param-value>
17     </init-param>
18     <load-on-startup>1</load-on-startup>
19   </servlet>
20   <servlet-mapping>
21     <servlet-name>Jersey REST Service</servlet-name>
22     <url-pattern>/v1/*</url-pattern>
23   </servlet-mapping>
24 </web-app>
```

Listing C.2: API configuration file web.xml

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:p="http://www.springframework.org/schema/p" xmlns:tx="http://www.
       springframework.org/schema/tx"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      xmlns:jpa="http://www.springframework.org/schema/data/jpa"
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9  http://www.springframework.org/schema/tx
10 http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
11 http://www.springframework.org/schema/data/jpa
12 http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
13 http://www.springframework.org/schema/context
14 http://www.springframework.org/schema/context/spring-context-3.0.xsd">
15
16     <context:component-scan base-package="lu.uni.lightning.webapi" />
17     <context:component-scan base-package="lu.uni.lightning.webapi.endpoints" />
18
19     <context:spring-configured />
20     <context:annotation-config />
21 </beans>
```

Listing C.3: API configuration file applicationContext.xml

```java
package lu.uni.lightning.webapi;

import org.glassfish.jersey.jackson.JacksonFeature;
import org.glassfish.jersey.server.ResourceConfig;
import org.springframework.stereotype.Component;

@Component
public class App extends ResourceConfig {
  public App() {
    super();
    packages("lu.uni.lightning.webapi.endpoints");
    // Use slf4j logging:
    register(CustomLoggingFilter.class);
    // Add the Access-Control-Allow-Origin header
    register(AccessControlAllowOriginFilter.class);
    // Use Jackson JSON (de)serializer :
    register(ObjectMapperProvider.class);
    register(JacksonFeature.class);
    register(GenericExceptionMapper.class);
  }
}
```

Listing C.4: API configuration file App.java

```java
package lu.uni.lightning.webapi;

import java.util.Properties;
import javax.persistence.EntityManagerFactory;
import javax.servlet.ServletContext;
import javax.sql.DataSource;
import org.apache.commons.dbcp2.BasicDataSource;
import org.hibernate.jpa.HibernatePersistenceProvider;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.dao.annotation.
    PersistenceExceptionTranslationPostProcessor;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
```

```
24 @EnableJpaRepositories
25 @EnableTransactionManagement
26 public class PersistenceJPAConfig {
27   private Logger log = LoggerFactory.getLogger(PersistenceJPAConfig.class);
28
29   @Autowired
30   private ServletContext ctx;
31
32   @Bean
33   public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
34     Boolean isEmbedded = (Boolean) ctx.getAttribute("embedded");
35     if(isEmbedded==null) {
36       log.info("Embedded flag not set. defaulting to false");
37       isEmbedded = false;
38     }
39     log.info("Selecting the embedded dataSource: " + isEmbedded);
40     LocalContainerEntityManagerFactoryBean em = new
       LocalContainerEntityManagerFactoryBean();
41     if(isEmbedded)
42       em.setDataSource(embeddedDataSource());
43     else
44       em.setDataSource(externalDataSource());
45     em.setPackagesToScan(new String[] { "lu.uni.lightning.webapi.system" });
46     em.setPersistenceProvider(new HibernatePersistenceProvider());
47     JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
48     em.setJpaVendorAdapter(vendorAdapter);
49     if(isEmbedded)
50       em.setJpaProperties(additionalPropertiesEmbedded());
51     else
52       em.setJpaProperties(additionalPropertiesExternal());
53     return em;
54   }
55
56   @Bean
57   public DataSource externalDataSource() {
58     //DriverManagerDataSource dataSource = new DriverManagerDataSource();
59       // Only ok for testing:
60       // Spring DriverManagerDataSource does not implement connection pooling,
61       // creating new connections on every call
62     BasicDataSource dataSource = new BasicDataSource(); // ok for production
63     dataSource.setDriverClassName("com.mysql.jdbc.Driver");
64     dataSource.setUrl("jdbc:mysql://localhost/lwebeditor");
65     dataSource.setUsername( "lightning" );
66     dataSource.setPassword( "XXX" );
67     return dataSource;
68   }
69
70   @Bean
```

```
71    public DataSource embeddedDataSource() {
72      BasicDataSource dataSource = new BasicDataSource(); // ok for production
73      dataSource.setDriverClassName("org.h2.Driver");
74      dataSource.setUrl("jdbc:h2:mem:lwebeditor");
75      //dataSource.setUsername( "lightning" );
76      //dataSource.setPassword( "XXX" );
77      return dataSource;
78    }
79
80    @Bean
81    public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {
82      JpaTransactionManager transactionManager = new JpaTransactionManager();
83      transactionManager.setEntityManagerFactory(emf);
84      return transactionManager;
85    }
86
87    @Bean
88    public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
89      return new PersistenceExceptionTranslationPostProcessor();
90    }
91
92    Properties additionalPropertiesExternal() {
93      return new Properties() {
94        private static final long serialVersionUID = 1L;
95        { // Hibernate Specific:
96          //setProperty("hibernate.hbm2ddl.auto", "create-drop");
97          setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
98          //setProperty("javax.persistence.schema-generation.database.action", "
      create");
99        }
100     };
101   }
102
103   Properties additionalPropertiesEmbedded() {
104     return new Properties() {
105       private static final long serialVersionUID = 1L;
106       { // Hibernate Specific:
107         //setProperty("hibernate.hbm2ddl.auto", "create-drop");
108         setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
109         setProperty("javax.persistence.schema-generation.database.action", "create"
      );
110       }
111     };
112   }
113 }
```

Listing C.5: API configuration file PersistenceJPAConfig.java

Appendix D

Entity, repository and endpoint example

Listing D.1 shows the generated entity class for projects. For the most part, the file is derived from the UML diagram of the database schema. There is a section reserved to add additional methods, which is used to handle associations from JSON references by ID.

Listing D.2 has the code for defining a repository for project entities. It defines methods to retrieve or save project instances from the database. Spring Data provides the implementation.

Listing D.3 defines the REST endpoint which handles requests concerning projects, i.e. everything for the URL `/api/v1/project/*`.

```
1  package lu.uni.lightning.webapi.system;
2  /*
3   *  Do not place import/include statements above this comment, just below.
4   *  @FILE-ID : (_4Ph-wPppEeWWa71teavVrg)
5   */
6  import java.io.Serializable;
7  import java.util.Date;
8  /*
9   *  Do not place import/include statements above this comment, just below.
10  *  @FILE-ID : (_4Ph-wPppEeWWa71teavVrg)
11  */
12 import java.util.UUID;
13 import javax.persistence.Column;
14 import javax.persistence.Entity;
```

```
15 import javax.persistence.GeneratedValue;
16 import javax.persistence.Id;
17 import javax.persistence.ManyToMany;
18 import javax.persistence.ManyToOne;
19 import javax.persistence.OneToMany;
20 import javax.persistence.Table;
21 import javax.persistence.Version;
22 import org.hibernate.annotations.GenericGenerator;
23 import com.fasterxml.jackson.annotation.JsonProperty;
24
25 /**
26  * Please describe the responsibility of your class in your modeling tool.
27  */
28 @Entity
29 @Table(name = "lwe_Project")
30 public class Project implements Serializable {
31   /** Stores the linked object of association '<em><b>owner</b></em>' */
32   @ManyToOne(cascade = {})
33   private User owner;
34
35   /** Stores all linked objects of association '<em><b>editors</b></em>' */
36   @ManyToMany(cascade = {}, mappedBy = "editableProjects")
37   private java.util.Set<User> editors = new java.util.HashSet<User>();
38
39   /** Stores all linked objects of association '<em><b>users</b></em>' */
40   @ManyToMany(cascade = {}, mappedBy = "usableProjects")
41   private java.util.Set<User> users = new java.util.HashSet<User>();
42
43   /** Stores all linked objects of association '<em><b>content</b></em>' */
44   @OneToMany(cascade = {}, mappedBy = "project")
45   private java.util.Set<RawProject> content = new java.util.HashSet<RawProject>();
46
47   /** Stores the linked object of association '<em><b>current</b></em>' */
48   @ManyToOne(cascade = {})
49   private RawProject current;
50
51   @Id @GeneratedValue(generator = "system-uuid")
52   @GenericGenerator(strategy = "uuid2", name = "system-uuid")
53   @Column(name = "id", length = 16)
54   private UUID id;
55   private Date created;
56   private Date lastModified;
57   private boolean isPublic;
58   private String name;
59   private ProjectType type;
60   private String description;
61
62   @Version
```

```
63    private int version;
64
65      /* ... generated constructor, getters and setters here ... */
66
67      /* PROTECTED REGION ID(java.class.own.code.implementation._4Ph-
        wPppEeWWa71teavVrg) ENABLED START */
68    // TODO: put your own implementation code here
69    private static final long serialVersionUID = 1L;
70    /**
71     * Establishes a link to the specified object for association '<em><b>current</b
        ></em>'.
72    * @param  current the id of the object to associate.
73     */
74    @JsonProperty("current")
75    public void setCurrentById(String current) {
76      RawProject p = null;
77      if (current != null) {
78        p = new RawProject();
79        p.setId(UUID.fromString(current));
80      }
81      setCurrent(p);
82    }
83    /* PROTECTED REGION END */
84  }
```

Listing D.1: Generated entity class: Project

```
1  package lu.uni.lightning.webapi.repositories;
2
3  import java.util.UUID;
4  import org.springframework.data.repository.PagingAndSortingRepository;
5  import lu.uni.lightning.webapi.system.Project;
6
7  public interface ProjectRepository
8      extends PagingAndSortingRepository<Project, UUID> {
9  }
```

Listing D.2: Interface for the repository: project

```
1  package lu.uni.lightning.webapi.endpoints;
2
3  import java.io.UnsupportedEncodingException;
4  import java.net.URLDecoder;
5  import java.util.Iterator;
6  import java.util.List;
```

```
7  import java.util.UUID;
8
9  import javax.ws.rs.Consumes;
10 import javax.ws.rs.DELETE;
11 import javax.ws.rs.GET;
12 import javax.ws.rs.NotFoundException;
13 import javax.ws.rs.POST;
14 import javax.ws.rs.Path;
15 import javax.ws.rs.PathParam;
16 import javax.ws.rs.Produces;
17 import javax.ws.rs.core.MediaType;
18 import javax.ws.rs.core.Response;
19
20 import org.slf4j.Logger;
21 import org.slf4j.LoggerFactory;
22 import org.springframework.beans.factory.annotation.Autowired;
23 import org.springframework.stereotype.Component;
24 import org.springframework.transaction.annotation.Transactional;
25
26 import jersey.repackaged.com.google.common.collect.Lists;
27 import lu.uni.lightning.webapi.repositories.ProjectRepository;
28 import lu.uni.lightning.webapi.system.Project;
29
30 @Path("project")
31 @Consumes(MediaType.APPLICATION_JSON)
32 @Produces(MediaType.APPLICATION_JSON)
33 @Component
34 public class ProjectEndPoint {
35   private Logger log = LoggerFactory.getLogger(ProjectEndPoint.class);
36
37   @Autowired
38   private ProjectRepository projects;
39
40   @GET
41   public List<Project> getAll() {
42     return Lists.newArrayList(projects.findAll());
43   }
44
45   @GET
46   @Transactional
47   @Path("{id: [a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0
       -9]{12}}")
48   public Project get(@PathParam("id") final UUID id) {
49     log.info("Requesting project for id: " + id.toString());
50     Project result = projects.findOne(id);
51     if(result==null)
52       throw new NotFoundException();
53     lazyLoad(result);
```

```
54      return result;
55    }
56
57    @GET
58    @Transactional
59    @Path("name/{name: .+}")
60    public Project getByName(@PathParam("name") final String _name) throws
        UnsupportedEncodingException {
61      String name = URLDecoder.decode(_name, "UTF-8");
62      log.info("Requesting project with name=" + name);
63      Project result = null;
64      Iterator<Project> iter = projects.findAll().iterator();
65      while(iter.hasNext()) {
66        Project p = iter.next();
67        if(p.getName().equals(name)) {
68          result = p;
69          break;
70        }
71      }
72      if(result==null)
73        throw new NotFoundException();
74      lazyLoad(result);
75      return result;
76    }
77
78    @DELETE
79    @Path("{id: [a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0
        -9]{12}}")
80    public void remove(@PathParam("id") final UUID id) {
81      projects.delete(id);
82    }
83
84    @DELETE
85    public void removeAll() {
86      projects.deleteAll();
87    }
88
89    @POST
90    public Response store(final Project newProject) {
91      return Response.status(201).entity(projects.save(newProject)).build();
92    }
93
94    @POST
95    @Path("multiple")
96    public Response store(final List<Project> newProjects) {
97      return Response.status(201).entity(Lists.newArrayList(projects.save(newProjects
        ))).build();
98    }
```

```
99
100   private void lazyLoad(Project p) {
101     p.getContent().size();
102     p.getCurrent();
103     p.getEditors().size();
104     p.getOwner();
105     p.getUsers().size();
106   }
107 }
```

Listing D.3: API Endpoint: project

Appendix E

DB schema

Figure E.1 shows the Server DB schema.

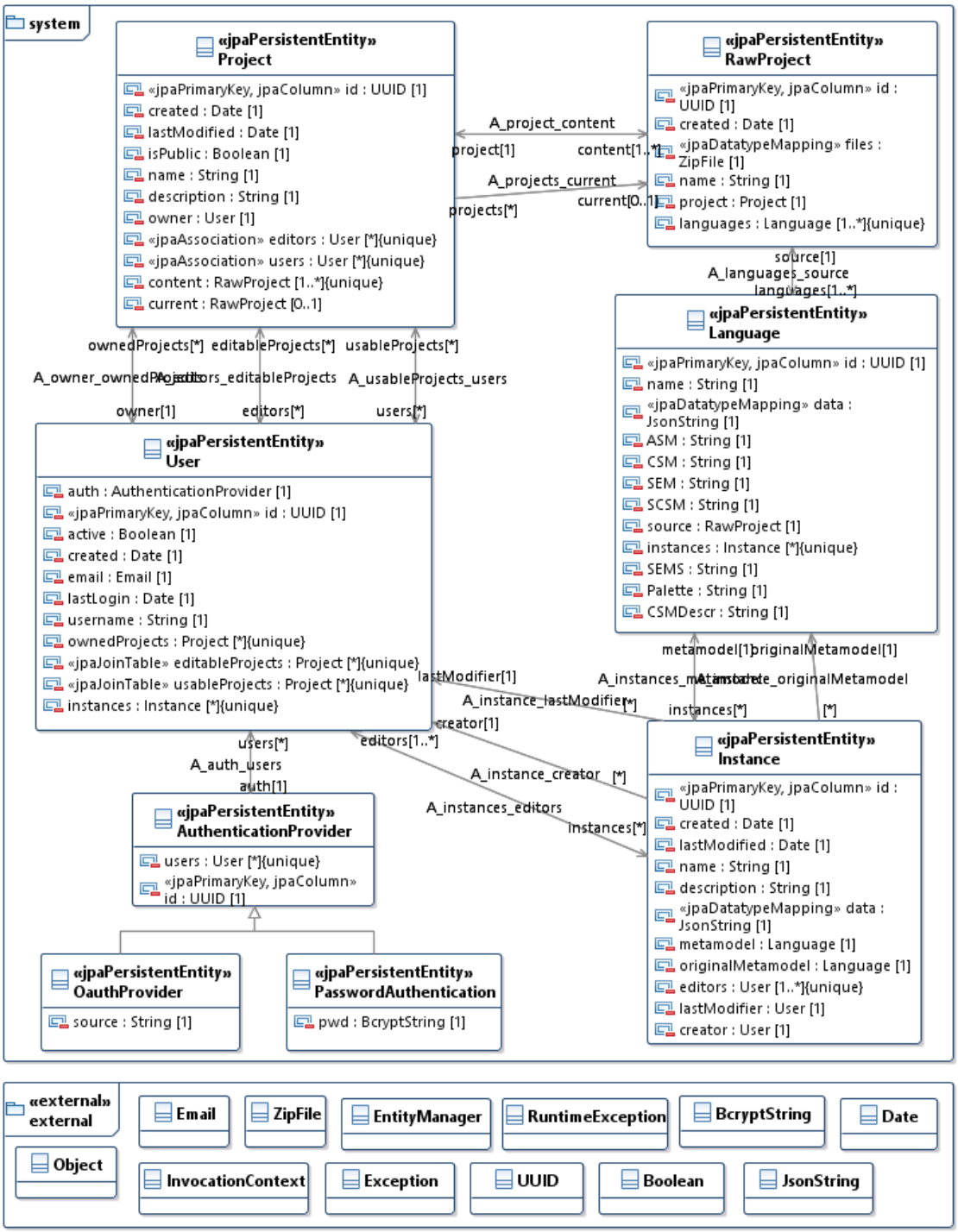

FIGURE E.1: Server DB schema

Appendix F

Description of GoJS interpretation of LightningVLM

Listing F.1 shows the description how GoJS should interpret the default LightningVLM.

```
{
  variables: {
    '{{shapeConfig}}': { // used as variable in the templates
      portId: "",
      fromLinkable: true, toLinkable: true,
      fromLinkableDuplicates: true, toLinkableDuplicates: true,
      fromLinkableSelfNode: true, toLinkableSelfNode: true,
      minSize: ['Size', { height: 50, width: 50 }],
      cursor: "pointer"
    },
    '{{shapeConfigN}}': {
      desiredSize: ['Size', { height: 50, width: 50 }]
    }
  },
  groupTemplateMap: {
    ':Group': ['Group', 'Auto', { // type used by the template to regroup CSM
    objects for one palette item
      locationSpot: ['go.Spot.Center'],
      ungroupable: true
    }, ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
    stringify: 'go.Point.stringify' }], ['Placeholder', { padding: 5 }]],
    TRIANGLE: ['Group', // class to create
      'Auto', {
        locationSpot: ['go.Spot.Center'],
        layout: [['ck.DynamicLayout'], {}]
```

```
24        }, ['Shape', 'Triangle', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
      parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
      'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
      padding: 5 }]
25      ],
26      RHOMBUS: ['Group', // class to create
27        'Auto', {
28          locationSpot: ['go.Spot.Center'],
29          layout: [['ck.DynamicLayout'], {}]
30        }, ['Shape', 'Diamond', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
      parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
      'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
      padding: 5 }]
31      ],
32      RECTANGLE: ['Group', // class to create
33        'Auto', {
34          locationSpot: ['go.Spot.Center'],
35          layout: [['ck.DynamicLayout'], {}]
36        }, ['Shape', 'Rectangle', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
      parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
      'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
      padding: 5 }]
37      ],
38      INVISIBLE_CONTAINER: ['Group', // class to create
39        'Auto', {
40          locationSpot: ['go.Spot.Center'],
41          layout: [['ck.DynamicLayout'], {}]
42        }, ['Shape', 'Rectangle', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }], {
43          stroke: "LightGrey",
44          strokeDashArray: [[4, 2]]
45        }], ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.
      negate' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.
      parse', stringify: 'go.Point.stringify' }], ['Placeholder', { padding: 5 }]
46      ],
47      HEXAGON: ['Group', // class to create
48        'Auto', {
49          locationSpot: ['go.Spot.Center'],
50          layout: [['ck.DynamicLayout'], {}]
51        }, ['Shape', 'Hexagon', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
      parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
      'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
      padding: 5 }]
52      ],
```

```
53      ELLIPSE: ['Group', // class to create
54        'Auto', {
55          locationSpot: ['go.Spot.Center'],
56          layout: [['ck.DynamicLayout'], {}]
57        }, ['Shape', 'Ellipse', ['{{shapeConfig}}'], ['Binding', { target: "fill",
        source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
        parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
        'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
        padding: 5 }]
58      ],
59      DOUBLE_ELLIPSE: ['Group', // class to create
60        'Auto', {
61          locationSpot: ['go.Spot.Center'],
62          layout: [['ck.DynamicLayout'], {}]
63        }, ['Shape', 'Ellipse', ['{{shapeConfig}}'], ['Binding', { target: "fill",
        source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
        parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
        'go.Point.parse', stringify: 'go.Point.stringify' }], ['Panel', 'Auto', ['Shape
        ', 'Ellipse', ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color
        " }]], ['Placeholder', { padding: 5 }]
64      ]],
65      CYLINDER: ['Group', // class to create
66        'Auto', {
67          locationSpot: ['go.Spot.Center'],
68          layout: [['ck.DynamicLayout'], {}]
69        }, ['Shape', 'Cylinder1', ['{{shapeConfig}}'], ['Binding', { target: "fill",
        source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
        parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
        'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
        padding: 5 }]
70      ],
71      CLOUD: ['Group', // class to create
72        'Auto', {
73          locationSpot: ['go.Spot.Center'],
74          layout: [['ck.DynamicLayout'], {}]
75        }, ['Shape', 'Cloud', ['{{shapeConfig}}'], ['Binding', { target: "fill",
        source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
        parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
        'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
        padding: 5 }]
76      ],
77      ACTOR: ['Group', // class to create
78        'Auto', {
79          locationSpot: ['go.Spot.Center'],
80          layout: [['ck.DynamicLayout'], {}]
```

```
81        }, ['Shape', 'Actor', ['{{shapeConfig}}'], ['Binding', { target: "fill",
      source: "color" }]], ['Binding', { target: 'selectable', source: 'parentKey',
      parse: 'ck.negate' }], ['Binding', { target: 'location', source: 'loc', parse:
      'go.Point.parse', stringify: 'go.Point.stringify' }], ['Placeholder', {
      padding: 5 }]
82      ],
83      IMAGE: ['Group', // class to create
84        'Auto', {
85          locationSpot: ['go.Spot.Center'],
86          layout: [['ck.DynamicLayout'], {}]
87        }, ['Picture', ['{{shapeConfig}}'], ['Binding', { target: "source", source: "
      url" }], ['Binding', { target: "background", source: "color" }]], ['Binding', {
       target: 'selectable', source: 'parentKey', parse: 'ck.negate' }], ['Binding',
      { target: 'location', source: 'loc', parse: 'go.Point.parse', stringify: 'go.
      Point.stringify' }], ['Placeholder', { padding: 5 }]
88      ]
89    },
90    nodeTemplateMap: {
91      TRIANGLE: ['Node', // class to create
92        {
93          locationSpot: ['go.Spot.Center']
94        }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Triangle', ['{{shapeConfigN}}'
      ], ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }]]
95      ],
96      RHOMBUS: ['Node', // class to create
97        {
98          locationSpot: ['go.Spot.Center']
99        }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Diamond', ['{{shapeConfigN}}'],
       ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }]]
100     ],
101     RECTANGLE: ['Node', // class to create
102       {
103         locationSpot: ['go.Spot.Center']
104       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Rectangle', ['{{shapeConfigN}}'
      ], ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }]]
105     ],
106     INVISIBLE_CONTAINER: ['Node', // class to create
107       {
108         locationSpot: ['go.Spot.Center']
```

```
109       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Rectangle', ['{{shapeConfigN}}'
      ], ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }], {
110         stroke: "LightGrey",
111         strokeDashArray: [[4, 2]]
112       }]
113     ],
114     HEXAGON: ['Node', // class to create
115       {
116         locationSpot: ['go.Spot.Center']
117       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Hexagon', ['{{shapeConfigN}}'],
       ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }]]
118     ],
119     ELLIPSE: ['Node', // class to create
120       {
121         locationSpot: ['go.Spot.Center']
122       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Ellipse', ['{{shapeConfigN}}'],
       ['{{shapeConfig}}'], ['Binding', { target: "fill", source: "color" }]]
123     ],
124     DOUBLE_ELLIPSE: ['Node', // class to create
125       'Auto', {
126         locationSpot: ['go.Spot.Center']
127       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Ellipse', ['{{shapeConfig}}'],
      { geometryStretch: ['go.GraphObject.Uniform'] }, ['Binding', { target: "fill",
      source: "color" }]], ['Panel', ['Shape', 'Ellipse', ['{{shapeConfig}}'], ['{{
      shapeConfigN}}'], ['Binding', { target: "fill", source: "color" }]]
128     ]],
129     CYLINDER: ['Node', // class to create
130       {
131         locationSpot: ['go.Spot.Center']
132       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Cylinder1', ['{{shapeConfig}}'
      ], ['{{shapeConfigN}}'], ['Binding', { target: "fill", source: "color" }]]
133     ],
134     CLOUD: ['Node', // class to create
135       {
136         locationSpot: ['go.Spot.Center']
```

```
137       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Cloud', ['{{shapeConfig}}'], ['
      {{shapeConfigN}}'], ['Binding', { target: "fill", source: "color" }]]
138     ],
139     ACTOR: ['Node', // class to create
140       {
141         locationSpot: ['go.Spot.Center']
142       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Shape', 'Actor', ['{{shapeConfig}}'], ['
      {{shapeConfigN}}'], ['Binding', { target: "fill", source: "color" }]]
143     ],
144     IMAGE: ['Node', // class to create
145       {
146         locationSpot: ['go.Spot.Center']
147       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['Picture', ['{{shapeConfig}}'], ['{{
      shapeConfigN}}'], ['Binding', { target: "source", source: "url" }], ['Binding',
       { target: "background", source: "color" }]]
148     ],
149     TEXT: ['Node', // class to create
150       {
151         locationSpot: ['go.Spot.Center']
152       }, ['Binding', { target: 'selectable', source: 'parentKey', parse: 'ck.negate
      ' }], ['Binding', { target: 'location', source: 'loc', parse: 'go.Point.parse',
       stringify: 'go.Point.stringify' }], ['TextBlock', ['Binding', { target: "font"
      , source: "font" }], ['Binding', { target: "text", source: "label" }], ['
      Binding', { target: "stroke", source: "color" }]]
153     ]
154   },
155   linkTemplateMap: {
156     CONNECTOR: ['Link', { relinkableFrom: true,
157       relinkableTo: true,
158       reshapable: true,
159       routing: ['go.Link.AvoidsNodes'],
160       curve: ['go.Link.JumpOver'],
161       corner: 5,
162       toShortLength: 4
163     }, ['Binding', { target: "points" }], ['Shape', ['Binding', { target: "stroke",
       source: "color" }]], ['Shape', { toArrow: "Standard" }, ['Binding', { target:
      "stroke", source: "color" }], ['Binding', { target: "fill", source: "color"
      }]], ['TextBlock', ['Binding', { target: "text", source: "text" }], ['Binding',
       { target: "stroke", source: "color" }]]]
164   },
165   //linkTypes: ['CONNECTOR'], // redundant: we have it as key in linkTemplateMap
166   containmentLinks: [{
```

```
167     fromType: 'Symbol',
168     toType: 'VisualElement',
169     connection: 'composedOf',
170     ordered: true
171   }],
172   nodeAttributeMap: {
173     Symbol: [{
174       connection: 'layout',
175       toType: 'Layout',
176       to: 'layout', // target attribute
177       map: {
178         HORIZONTAL_LAYOUT: 'horizontal',
179         '': 'vertical' // default
180       }
181     }],
182     IMAGE: [{
183       connection: 'url',
184       toType: 'univ',
185       from: 'label',
186       to: 'url',
187       conversion: 'ck.cleanQuotes'
188     }],
189     INVISIBLE_CONTAINER: [{
190       connection: 'color',
191       fromType: 'VisualElement',
192       toType: 'Lightning_Color',
193       to: 'color',
194       map: {
195         '': 'transparent' // the invisible container is always transparent
196       }
197     }],
198     CONNECTOR: [{
199         connection: 'source',
200         toType: 'VisualElement',
201         from: 'key',
202         to: 'from',
203         type: Number
204       }, {
205         connection: 'target',
206         toType: 'VisualElement',
207         from: 'key',
208         to: 'to',
209         type: Number
210       }, {
211         connection: 'connectorLabel',
212         toType: 'univ',
213         from: 'label',
214         to: 'text',
```

```
215         conversion: 'ck.cleanSuffix',
216         ordered: true // will add [ seq/Int ] to label and order by that
217       }, {
218         connection: 'color',
219         fromType: 'VisualElement',
220         toType: 'Lightning_Color',
221         to: 'color',
222         map: {
223           'YELLOW': 'yellow',
224           'WHITE': 'white',
225           'RED': 'red',
226           'PURPLE': 'purple',
227           'ORANGE': 'orange',
228           'GREEN': 'green',
229           'GRAY': 'gray',
230           'BROWN': 'brown',
231           'BLUE': 'blue',
232           'BLACK': 'black',
233           '': 'black' // default
234         }
235       }
236     ],
237     TEXT: [{
238         connection: 'textLabel',
239         toType: 'univ',
240         from: 'label',
241         to: 'label',
242         conversion: 'ck.cleanSuffix',
243         ordered: true
244       }, {
245         connection: 'isItalic',
246         toType: 'Bool',
247         to: 'isItalic',
248         map: {
249           'True': true,
250           '': false
251         }
252       }, {
253         connection: 'isBold',
254         toType: 'Bool',
255         to: 'isBold',
256         map: {
257           'True': true,
258           '': false
259         }
260       }, {
261         to: 'font',
262         postProcessor: true,
```

```
263         conversion: 'ck.fontCSS'
264       }, {
265         connection: 'color',
266         fromType: 'VisualElement',
267         toType: 'Lightning_Color',
268         to: 'color',
269         map: {
270           'YELLOW': 'yellow',
271           'WHITE': 'white',
272           'RED': 'red',
273           'PURPLE': 'purple',
274           'ORANGE': 'orange',
275           'GREEN': 'green',
276           'GRAY': 'gray',
277           'BROWN': 'brown',
278           'BLUE': 'blue',
279           'BLACK': 'black',
280           '': 'black' // default
281         }
282       }
283     ],
284     VisualElement: [{
285       connection: 'color',
286       toType: 'Lightning_Color',
287       to: 'color',
288       map: {
289         'YELLOW': 'yellow',
290         'WHITE': 'white',
291         'RED': 'red',
292         'PURPLE': 'purple',
293         'ORANGE': 'orange',
294         'GREEN': 'green',
295         'GRAY': 'gray',
296         'BROWN': 'brown',
297         'BLUE': 'blue',
298         'BLACK': 'black',
299         '': 'transparent' // default
300       }
301     }]
302   }
303 }
```

Listing F.1: Description how GoJS should interpret the LightningVLM

Appendix G

Documentation of API

This appendix documents the endpoints of the backend shown in table G.1.

Index	URL	Method
G.1.1	`/project`	GET
G.1.2	`/project/:uid`	GET
G.1.3	`/project/name/:name`	GET
G.1.4	`/project`	DELETE
G.1.5	`/project/:uid`	DELETE
G.1.6	`/project`	POST
G.1.7	`/project/multiple`	POST
G.2.1	`/raw-project`	GET
G.2.2	`/raw-project/:uid`	GET
G.2.3	`/raw-project`	DELETE
G.2.4	`/raw-project/:uid`	DELETE
G.2.5	`/raw-project`	POST
G.2.6	`/raw-project/multiple`	POST
G.3.1	`/language`	GET
G.3.2	`/language/current`	GET
G.3.3	`/language/:uid`	GET
G.3.4	`/language/:uid/instances`	GET
G.3.5	`/language`	DELETE
G.3.6	`/language/:uid`	DELETE
G.3.7	`/language`	POST
G.4.1	`/instance`	GET
G.4.2	`/instance/:uid`	GET
G.4.3	`/instance`	DELETE
G.4.4	`/instance/:uid`	DELETE
G.4.5	`/instance`	POST
G.4.6	`/instance/multiple`	POST
G.5.1	`/analyzer/:uid`	POST
G.5.2	`/analyzer/:id/:index`	POST
G.5.3	`/analyzer/:id`	DELETE
G.6.1	`/transform/:uid/csm`	POST
G.6.2	`/transform/:uid/sem`	POST
G.6.3	`/transform/:uid/scsm`	POST
G.7.1	`/palette`	POST

TABLE G.1: API endpoints

G.1 Project endpoint

G.1.1

Title	Return all projects
URL	`/project`
Method	GET
URL parameters	None
Data parameters	None
Success response	

```
[
  {
    "id": "42d14bd3-d530-4748-9ec5-6b9df5b27203",
    "owner": null,
    "editors": null,
    "users": null,
    "content": null,
    "current": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "created": null,
    "lastModified": null,
    "isPublic": false,
    "name": "PetriNet",
    "type": null,
    "description": null,
    "version": 1
  }
]
```

Error response	None

G.1.2

Title	Return the project with the given ID
URL	`/project/:uid`
Method	GET
URL parameters	**Required:** `uid=[UUID]` example: uid=42d14bd3-d530-4748-9ec5-6b9df5b27203
Data parameters	None

Success response

```
{
  "id": "42d14bd3-d530-4748-9ec5-6b9df5b27203",
  "owner": null,
  "editors": [],
  "users": [],
  "content": [
    "6c2a74f9-7bd1-4dd9-838e-e7a730559267"
  ],
  "current": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
  "created": null,
  "lastModified": null,
  "isPublic": false,
  "name": "PetriNet",
  "type": null,
  "description": null,
  "version": 1
}
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.1.3

Title	Return the project with the given name
URL	`/project/name/:name`
Method	GET
URL parameters	**Required:** `name=[String]` example: name=PetriNet
Data parameters	None
Success response	

```
{
  "id": "42d14bd3-d530-4748-9ec5-6b9df5b27203",
  "owner": null,
  "editors": [],
  "users": [],
  "content": [
    "6c2a74f9-7bd1-4dd9-838e-e7a730559267"
  ],
  "current": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
  "created": null,
  "lastModified": null,
  "isPublic": false,
  "name": "PetriNet",
  "type": null,
  "description": null,
  "version": 1
}
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.1.4

Title	Delete all projects
URL	`/project`
Method	DELETE
URL parameters	None
Data parameters	None
Success response	Code: 200
Error response	None

G.1.5

Title	Delete the project with the given ID
URL	`/project/:uid`
Method	DELETE
URL parameters	**Required:** `uid=[UUID]` example: uid=42d14bd3-d530-4748-9ec5-6b9df5b27203
Data parameters	None
Success response	Code: 200
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.1.6

Title	Insert a new project
URL	`/project`
Method	POST
URL parameters	None
Data parameters	

```
{
    "id": null,
    "owner": [UUID],
    "editors": [list of UUID],
    "users": [list of UUID],
    "content": [list of UUID],
    "current": [UUID],
    "created": [date],
    "lastModified": [date],
    "isPublic": [boolean],
    "name": [string],
    "type": null,
    "description": [string],
    "version": 1
  }
```

Success response	Code: 201 Returns the data parameter with ID set to a generated UUID.
Error response	None

G.1.7

Title	Insert new projects
URL	`/project/multiple`
Method	POST
URL parameters	None
Data parameters	

```
[ {
    "id": null,
    "owner": [UUID],
    "editors": [list of UUID],
    "users": [list of UUID],
    "content": [list of UUID],
    "current": [UUID],
    "created": [date],
    "lastModified": [date],
    "isPublic": [boolean],
    "name": [string],
    "type": null,
    "description": [string],
    "version": 1
  } ]
```

Success response	Code: 201 Returns the data parameter with the IDs set to a generated UUID.
Error response	None

G.2 RawProject endpoint

G.2.1

Title	Return all raw projects
URL	`/raw-project`
Method	GET
URL parameters	None
Data parameters	None
Success response	

```
[
  {
    "id": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "project": "42d14bd3-d530-4748-9ec5-6b9df5b27203",
    "languages": null,
    "created": null,
    "files": "...",
    "name": "PetriNet",
    "version": 0
  }
]
```

Error response	None
Notes	The files field contains a base64 encoded zip file of the Alloy, f-Alloy and JSON files of the project.

G.2.2

Title	Return the raw project with the given ID
URL	`/raw-project/:uid`
Method	GET
URL parameters	**Required:** `uid=[UUID]` example: uid=6c2a74f9-7bd1-4dd9-838e-e7a730559267
Data parameters	None

Success response

```
{
    "id": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "project": "42d14bd3-d530-4748-9ec5-6b9df5b27203",
    "languages": null,
    "created": null,
    "files": "...",
    "name": "PetriNet",
    "version": 0
  }
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.2.3

Title	Delete all raw projects
URL	`/raw-project`
Method	DELETE
URL parameters	None
Data parameters	None
Success response	Code: 200
Error response	None

G.2.4

Title	Delete the raw project with the given ID
URL	`/raw-project/:uid`
Method	DELETE
URL parameters	**Required:** `uid=[UUID]` example: uid=6c2a74f9-7bd1-4dd9-838e-e7a730559267
Data parameters	None
Success response	Code: 200
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.2.5

Title	Insert a new raw project
URL	`/raw-project`
Method	POST
URL parameters	None
Data parameters	

```
{
    "id": null,
    "project": [UUID],
    "languages": null,
    "created": null,
    "files": [alphanumeric],
    "name": [string],
    "version": 0
  }
```

Success response	Code: 201 Returns the data parameter with ID set to a generated UUID.
Error response	None

G.2.6

Title	Insert new raw projects
URL	`/raw-project/multiple`
Method	POST
URL parameters	None
Data parameters	

```
[ {
    "id": null,
    "project": [UUID],
    "languages": null,
    "created": null,
    "files": [alphanumeric],
    "name": [string],
    "version": 0
  } ]
```

Success response	Code: 201 Returns the data parameter with the IDs set to a generated UUID.
Error response	None

G.3 Language endpoint

G.3.1

Title	Return all languages
URL	`/language`
Method	GET
URL parameters	None
Data parameters	None
Success response	


```
[
  {
    "id": "32e332e2-9510-4cff-b080-08eb07797365",
    "source": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "instances": null,
    "name": "PetriNet",
    "data": "...",
    "version": 0,
    "asm": "PetriNet/AbstractSyntax/ASM.als",
    "csm": "PetriNet/ConcreteSyntax/ASM2VLM.fals",
    "sem": "PetriNet/Semantics/init.als",
    "palette": "PetriNet/ConcreteSyntax/palette.json",
    "scsm": "PetriNet/ConcreteSyntax/SEM2VLM.fals",
    "csmdescr": null,
    "sems": "PetriNet/Semantics/step.fals"
  }
]
```


Error response	None

G.3.2

Title	Return all current languages
URL	`/language/current`
Method	GET
URL parameters	None
Data parameters	None
Success response	


```
[
  {
    "id": "32e332e2-9510-4cff-b080-08eb07797365",
    "source": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "instances": null,
    "name": "PetriNet",
    "data": "...",
    "version": 0,
    "asm": "PetriNet/AbstractSyntax/ASM.als",
    "csm": "PetriNet/ConcreteSyntax/ASM2VLM.fals",
    "sem": "PetriNet/Semantics/init.als",
    "palette": "PetriNet/ConcreteSyntax/palette.json",
    "scsm": "PetriNet/ConcreteSyntax/SEM2VLM.fals",
    "csmdescr": null,
    "sems": "PetriNet/Semantics/step.fals"
  }
]
```


Error response	None
Notes	A language is current if it is contained in a raw-project that a project references as current.

G.3.3

Title	Return the language with the given ID
URL	`/language/:uid`
Method	GET
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	None

Success response


```
{
    "id": "32e332e2-9510-4cff-b080-08eb07797365",
    "source": "6c2a74f9-7bd1-4dd9-838e-e7a730559267",
    "instances": null,
    "name": "PetriNet",
    "data": "...",
    "version": 0,
    "asm": "PetriNet/AbstractSyntax/ASM.als",
    "csm": "PetriNet/ConcreteSyntax/ASM2VLM.fals",
    "sem": "PetriNet/Semantics/init.als",
    "palette": "PetriNet/ConcreteSyntax/palette.json",
    "scsm": "PetriNet/ConcreteSyntax/SEM2VLM.fals",
    "csmdescr": null,
    "sems": "PetriNet/Semantics/step.fals"
  }
```


Error response


```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.3.4

Title	Return the instances of the language with the given ID
URL	`/language/:uid/instances`
Method	GET
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	None
Success response	

```
[ {
    "id": "a1d89005-c0ca-43c0-9580-dd99d31dfeb4",
    "metamodel": "32e332e2-9510-4cff-b080-08eb07797365",
    "originalMetamodel": null,
    "editors": null,
    "lastModifier": null,
    "creator": null,
    "created": null,
    "lastModified": null,
    "name": "Test",
    "description": null,
    "data": "...",
    "version": 0
  } ]
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.3.5

Title	Delete all languages
URL	`/language`
Method	DELETE
URL parameters	None
Data parameters	None
Success response	Code: 200
Error response	None

G.3.6

Title	Delete the language with the given ID
URL	`/language/:uid`
Method	DELETE
URL parameters	**Required:** `uid=[UUID]` example: uid=2d6e88ad-7f0d-4ff9-905a-caa30f693031
Data parameters	None
Success response	Code: 200
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.3.7

Title	Insert a new language
URL	`/language`
Method	POST
URL parameters	None
Data parameters	

```
{
    "id": null,
    "source": [UUID],
    "instances": null,
    "name": [string],
    "data": null,
    "version": 0,
    "asm": [string],
    "csm": [string],
    "sem": [string],
    "palette": [string],
    "scsm": [string],
    "csmdescr": [string],
    "sems": [string]
  }
```

Success response	Code: 201 Returns the data parameter with ID set to a generated UUID and data to the JSON transformed meta-models.
Error response	

```
{
  "message": "HTTP 400 Bad Request",
  "developerMessage": "...",
  "status": 400
}
```

Notes	It is checked that the referenced raw-project exists and that the language contains an ASM.

G.4 Instance endpoint

G.4.1

Title	Return all instances
URL	`/instance`
Method	GET
URL parameters	None
Data parameters	None
Success response	


```
[
  {
    "id": "a1d89005-c0ca-43c0-9580-dd99d31dfeb4",
    "metamodel": "32e332e2-9510-4cff-b080-08eb07797365",
    "originalMetamodel": null,
    "editors": null,
    "lastModifier": null,
    "creator": null,
    "created": null,
    "lastModified": null,
    "name": "Test",
    "description": null,
    "data": "...",
    "version": 0
  }
]
```


Error response	None
Notes	To get only the instances from a specific language refer to `/language/:uid/instances`

G.4.2

Title	Return the instance with the given ID
URL	`/instance/:uid`
Method	GET
URL parameters	**Required:** `uid=[UUID]` example: uid=a1d89005-c0ca-43c0-9580-dd99d31dfeb4
Data parameters	None
Success response	

```
{
    "id": "a1d89005-c0ca-43c0-9580-dd99d31dfeb4",
    "metamodel": "32e332e2-9510-4cff-b080-08eb07797365",
    "originalMetamodel": null,
    "editors": null,
    "lastModifier": null,
    "creator": null,
    "created": null,
    "lastModified": null,
    "name": "Test",
    "description": null,
    "data": "...",
    "version": 0
  }
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.4.3

Title	Delete all instances
URL	`/instance`
Method	DELETE
URL parameters	None
Data parameters	None
Success response	Code: 200
Error response	None

G.4.4

Title	Delete the instance with the given ID
URL	`/instance/:uid`
Method	DELETE
URL parameters	**Required:** `uid=[UUID]` example: uid=a1d89005-c0ca-43c0-9580-dd99d31dfeb4
Data parameters	None
Success response	Code: 200
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.4.5

Title	Insert a new instance
URL	`/instance`
Method	POST
URL parameters	None
Data parameters	

```
{
    "id": null,
    "metamodel": [UUID],
    "originalMetamodel": null,
    "editors": [list of UUID],
    "lastModifier": [UUID],
    "creator": [UUID],
    "created": [date],
    "lastModified": [date],
    "name": [string],
    "description": [string],
    "data": "...",
    "version": 0
}
```

Success response	Code: 201 Returns the data parameter with ID set to a generated UUID.
Error response	None
Notes	The data field is a JSON serialization of the Lightning instance.

G.4.6

Title	Insert new instances
URL	`/instance/multiple`
Method	POST
URL parameters	None
Data parameters	

```
[ {
    "id": null,
    "metamodel": [UUID],
    "originalMetamodel": null,
    "editors": [list of UUID],
    "lastModifier": [UUID],
    "creator": [UUID],
    "created": [date],
    "lastModified": [date],
    "name": [string],
    "description": [string],
    "data": "...",
    "version": 0
  } ]
```

Success response	Code: 201 Returns the data parameter with the IDs set to a generated UUID.
Error response	None

G.5 Analyzer endpoint

G.5.1

Title	Analyzes the language with the given ID
URL	`/analyzer/:uid`
Method	POST
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	None
Success response	Code: 200 Returns an integer as the session ID for the current analysis.
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.5.2

Title	Return the analysis result depending on the given index from the analysis session with the given ID
URL	`/analyzer/:id/:index`
Method	POST
URL parameters	**Required:** `id=[Integer]`, `index=[Integer]` example: id=4, index=0
Data parameters	None
Success response	Code: 200

```
{
  "solution":{ /* ASM instance */ },
  "isSatisfiable":true,
  "hasNext":true,
  "index":0
}
```

Error response

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

Notes	The endpoint does not actually cache all results. It returns the last analysis result if the asked index is less or equal to the index of the last result. If the asked index is greater then it calculates the next analysis result, caches it and returns it.

G.5.3

Title	Delete the analysis session with the given ID
URL	`/analyzer/:id`
Method	DELETE
URL parameters	**Required:** `id=[Integer]` example: id=4
Data parameters	None
Success response	Code: 200
Error response	None

G.6 Transformation endpoint

G.6.1

Title	Apply the ASM2VLM transformation of the given language
URL	`/transform/:uid/csm`
Method	POST
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	An ASM instance of the given language.

```
{ "class": "LightningInstanceModel",
  "nodeCategoryProperty": "type",
  "linkCategoryProperty": "type",
  "nodeIsGroupProperty": "hasChildren",
  "nodeGroupKeyProperty": "parentKey",
  "nodeDataArray": [ {"type":"Transition", "key":1,
     "label":"Transition$1"},
     ... ],
  "linkDataArray": [
{"type":"Node-Node-nextNode", "from":5, "to":2}, ...
 ]}
```

Success response	Returns the transformed instance.
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.6.2

Title	Apply the semantic step transformation of the given language
URL	`/transform/:uid/sem`
Method	POST
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	A semantic instance of the given language.

```
{ "class": "LightningInstanceModel",
  "nodeCategoryProperty": "type",
  "linkCategoryProperty": "type",
  "nodeIsGroupProperty": "hasChildren",
  "nodeGroupKeyProperty": "parentKey",
  "nodeDataArray": [ {"type":"Transition", "key":1,
     "label":"Transition$1"},
     ... ],
  "linkDataArray": [
{"type":"Node-Node-nextNode", "from":5, "to":2}, ...
 ]}
```

Success response	Returns the transformed instance.
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.6.3

Title	Apply the SEM2VLM transformation of the given language
URL	`/transform/:uid/scsm`
Method	POST
URL parameters	**Required:** `uid=[UUID]` example: uid=32e332e2-9510-4cff-b080-08eb07797365
Data parameters	A semantic instance of the given language.

```
{ "class": "LightningInstanceModel",
  "nodeCategoryProperty": "type",
  "linkCategoryProperty": "type",
  "nodeIsGroupProperty": "hasChildren",
  "nodeGroupKeyProperty": "parentKey",
  "nodeDataArray": [ {"type":"Transition", "key":1,
     "label":"Transition$1"},
     ... ],
  "linkDataArray": [
{"type":"Node-Node-nextNode", "from":5, "to":2}, ...
 ]}
```

Success response	Returns the transformed instance.
Error response	

```
{
  "message": "HTTP 404 Not Found",
  "developerMessage": "...",
  "status": 404
}
```

G.7 Palette endpoint

G.7.1

Title Analyzes the given raw project and language and returns the palette entries.

URL `/palette`

Method POST

URL parameters None

Data parameters

```
{
  "project": { /* a raw-project */ },
  "language": { /* a language */ }
}
```

Success response Code: 200

```
[ /* a list of palette entries */
  {
    "asm": { /* an ASM instance */ },
    "csm": { /* an CSM instance */ }
  }, /* ... */
]
```

Error response

```
{
  "message": "HTTP 500 Internal server error",
  "developerMessage": "...",
  "status": 500
}
```

Appendix H

Documentation of GUI

Before the Lightning Web Editor can be used to edit instances, a language needs to be loaded. The 'Open language dialog' is shown in fig. H.1.

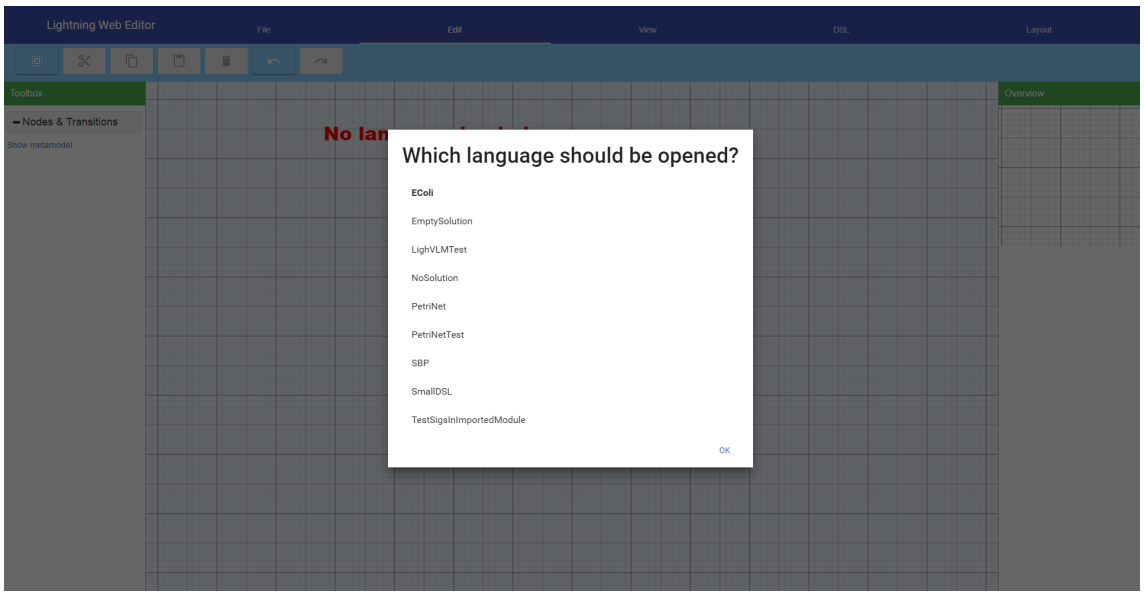

FIGURE H.1: Open language dialog

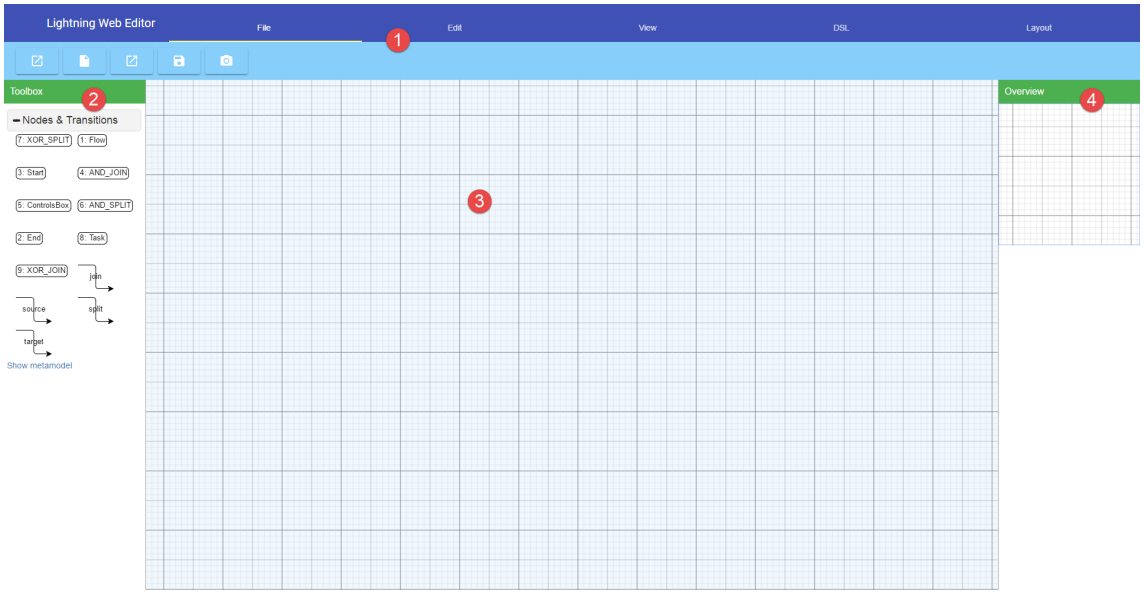

FIGURE H.2: Screenshot of GUI

Fig. H.2 shows the GUI with a language loaded. It has the following parts:

1. The menu bar shows different tabs each with its own menu items. Cf. H.1 for a more detailed description of the menu items.

2. The left sidebar contains the palette. This can be used to edit the displayed instance by dragging and dropping nodes and links onto the canvas, where they are added to the instance.

3. The canvas displays the current instance.

4. The right sidebar displays an overview of the whole instance as well as the current viewport of the canvas.

   Whether a sidebar is shown can be toggled in the view menu bar. The width of a sidebar can be resized by dragging the border of the side bar. The resize action is cancellable by pressing Escape.

When executing operations that may take longer to complete, a busy overlay is shown (fig. H.3).

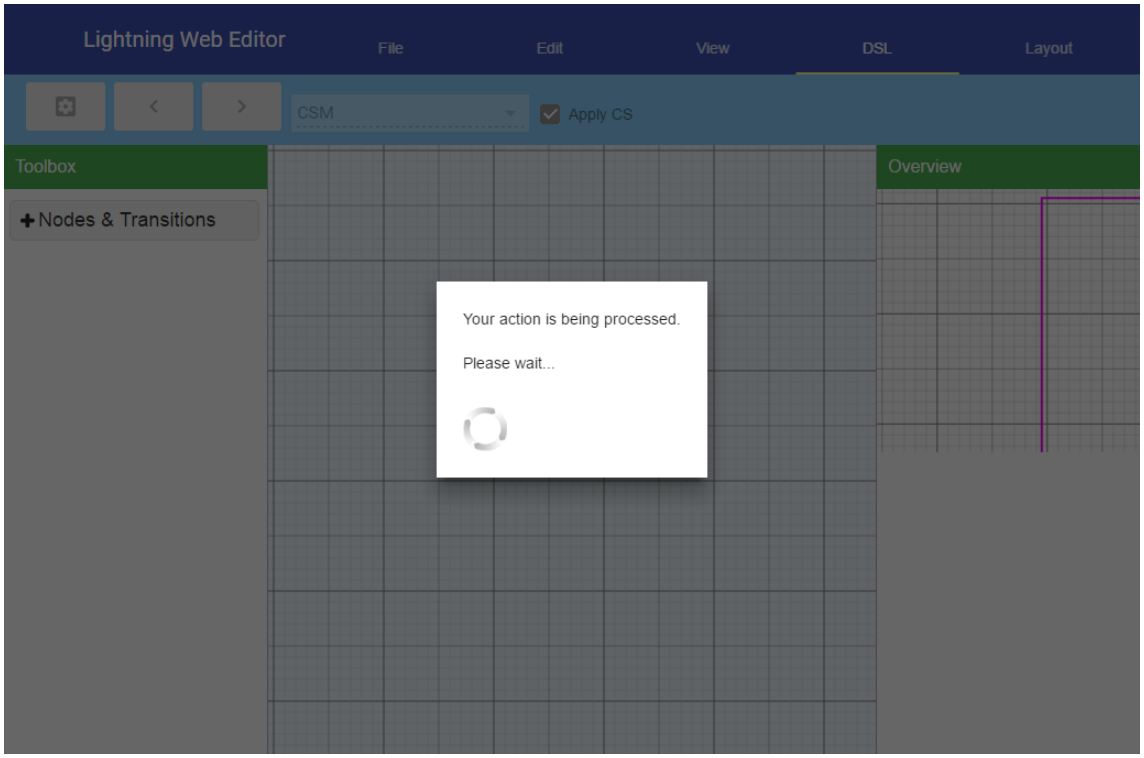

FIGURE H.3: Busy overlay

H.1 Menu bars

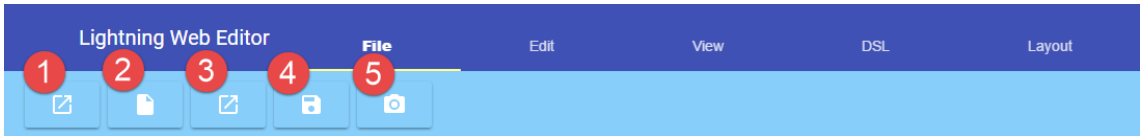

FIGURE H.4: Menu bar File

The File menu bar (fig. H.4) has the following items:

1. 'Open language' opens the 'Open language dialog'.
2. 'New instance' will reset the diagram to a new instance of the currently loaded language.
3. 'Open instance' will open the 'Open instance dialog'.
4. 'Save instance as' will open the 'Save instance as dialog'.

5. 'Export as image' will save an image of the current diagram. It will try to capture the whole instance at zoom level 100%. It will crop the picture if it is bigger than 3000x3000 pixels. It will not show the background.

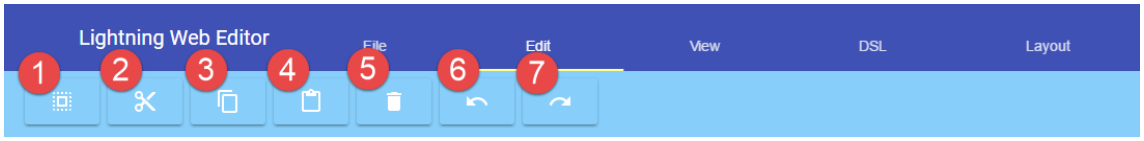

FIGURE H.5: Menu bar Edit

The Edit menu bar (fig. H.5) has the following items:

1. 'Select all' will select all parts on the canvas.
2. 'Cut' will perform the cut clipboard action.
3. 'Copy' will perform the copy clipboard action.
4. 'Paste' will perform the paste clipboard action.
5. 'Delete' will delete the current selection.
6. 'Undo' will undo the last action.
7. 'Redo' will redo the last undone action.

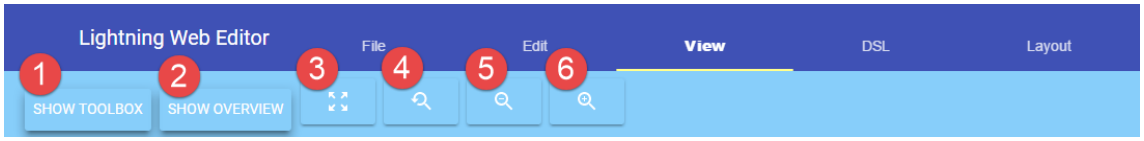

FIGURE H.6: Menu bar View

The View menu bar (fig. H.6) has the following items:

1. 'Show toolbox' allows toggling the visibility of the left sidebar.
2. 'Show overview' allows toggling the visibility of the right sidebar.
3. 'Zoom to fit' will zoom the diagram to show the whole instance.
4. 'Reset zoom' will set the zoom to 100 %.
5. 'Zoom out' will decrease the zoom.

6. 'Zoom in' will increase the zoom.

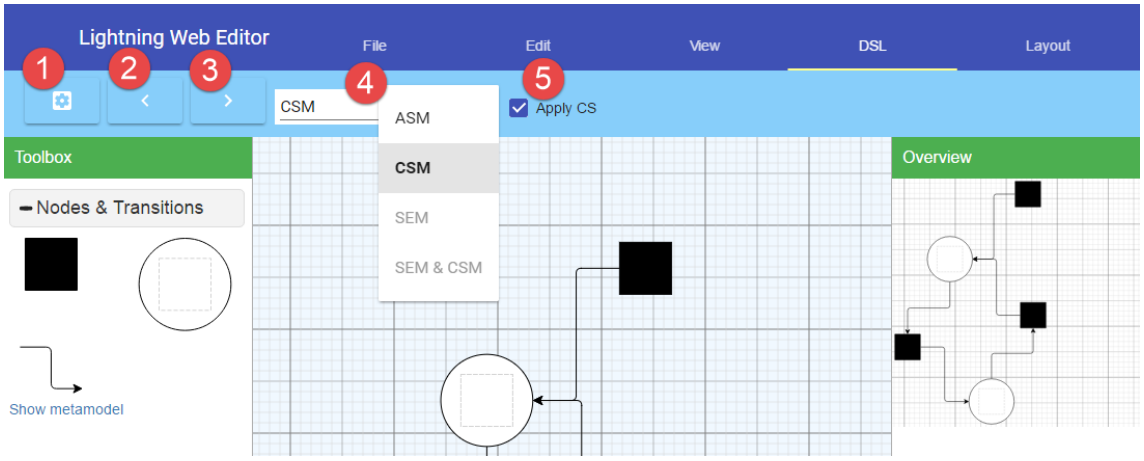

FIGURE H.7: Menu bar DSL

The DSL menu bar (fig. H.7) has the following items:

1. 'Generate instance' will analyze the current language and generate a series of instances and display the first one.

2. 'Previous instance' will go to the previous instance, if instances have been generated and there is a previous one).

3. 'Next instance' will go to the next instance, if instances have been generated and there is a next one).

4. 'UI mode' will allow switching between ASM and CSM mode. In the future, it will also be possible to access semantic and semantic with CSM with it.

5. 'Apply concrete syntax' exposes an intermediary step in the application of concrete syntax for debugging. With this disabled, it will show the abstract representation of the CSM instance.

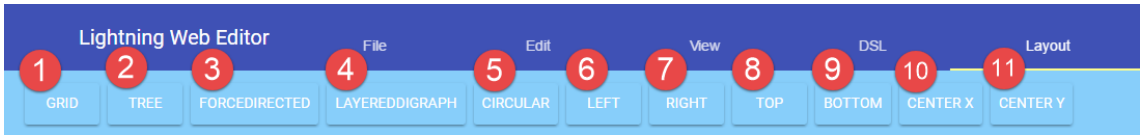

FIGURE H.8: Menu bar Layout

The layout algorithms apply to a target. This target is the selection if the selection is not empty or the diagram if the selection is empty. Align commands are only available if the

selection contains at least two parts. The Layout menu bar (fig. H.8) has the following items:

1. 'Grid' will layout the target in a grid.
2. 'Tree' will layout the target as a tree. This can cope in a limited fashion with instances that are not quite a tree.
3. 'Force directed' will layout the target applying an attractive force on links and repulsive forces between nodes.
4. 'Layered digraph' will layout the target in layers.
5. 'Circular' will layout the target in a circle.
6. 'Left' will align the selection using their left-most edge.
7. 'Right' will align the selection using their right-most edge.
8. 'Top' will align the selection using their top-most edge.
9. 'Bottom' will align the selection using their bottom-most edge.
10. 'Center X' will align the selection using their center's X coordinate.
11. 'Center Y' will align the selection using their center's Y coordinate.

H.2 Dialogs

Fig. H.9 shows the 'Save instance dialog'. It asks for a name under which the current instance will be saved. After confirming the dialog, the current instance is sent to the backend and persisted in the DB under the entered name.

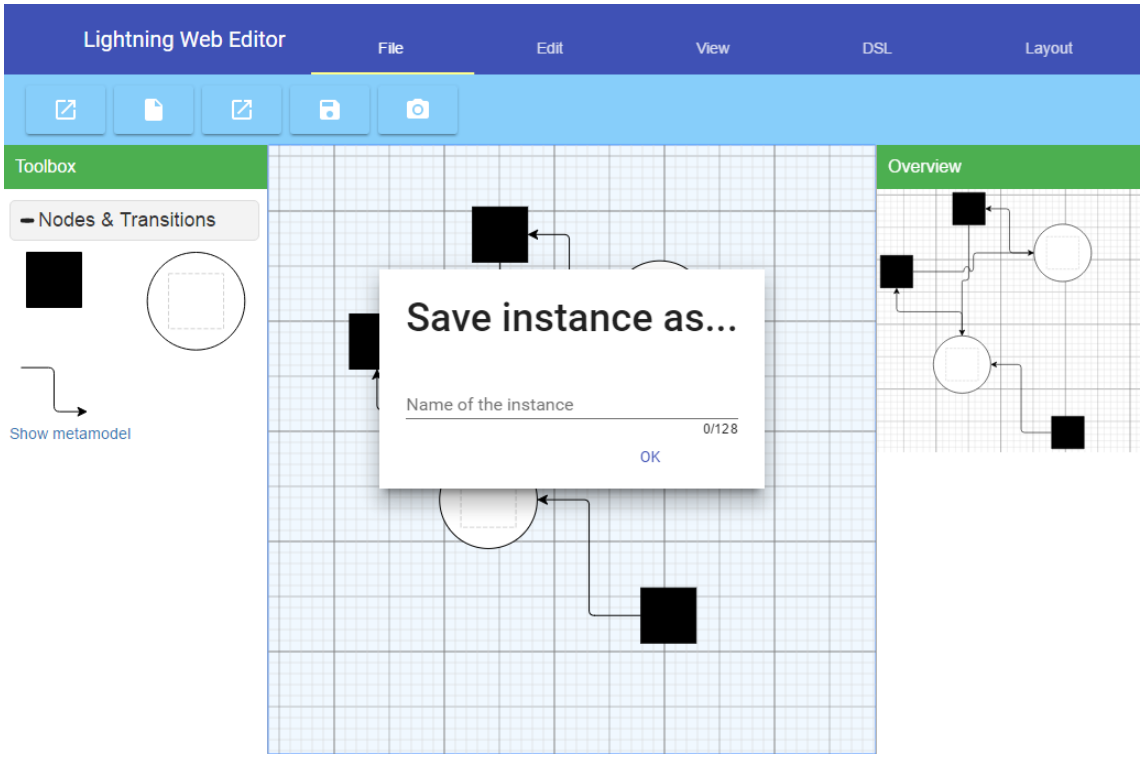

FIGURE H.9: Save instance dialog

Fig. H.10 shows the 'Open instance dialog'. It shows the available instances for the current language. When the selection of an instance is confirmed, that instance is opened in the editor.

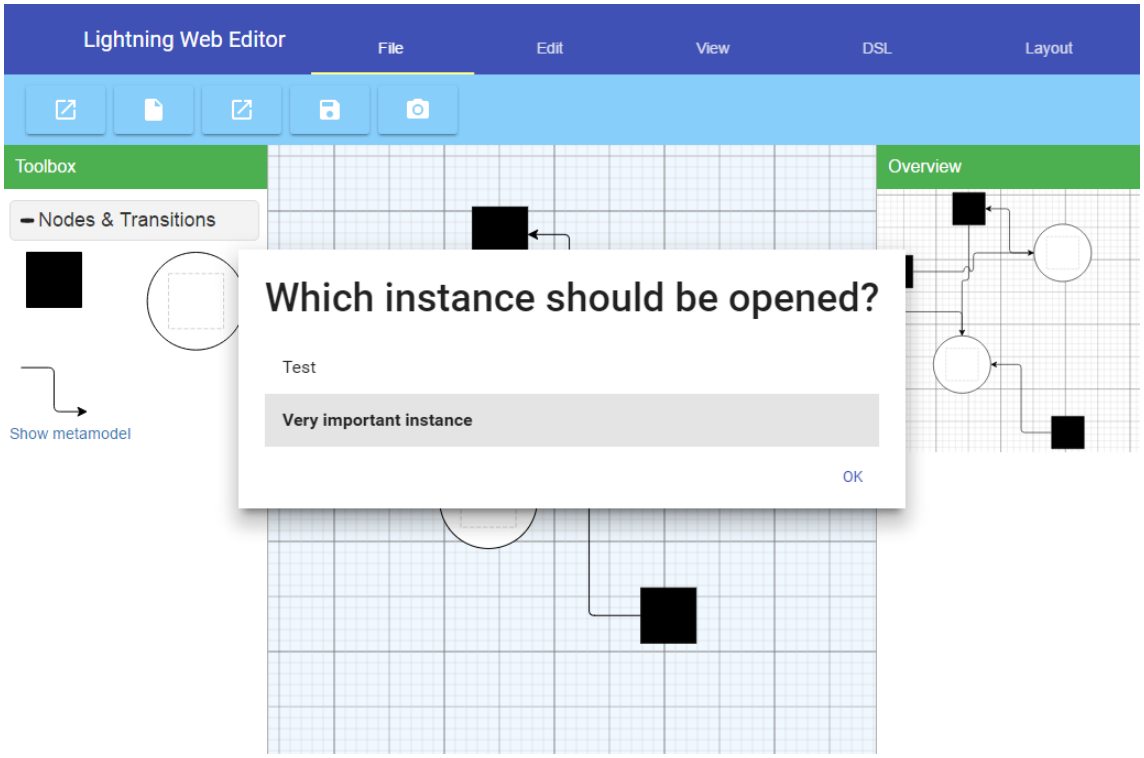

FIGURE H.10: Open instance dialog

H.3 Keyboard commands

The diagramming library supports the following default shortcuts [46]:

- Del & Backspace delete the selection.
- Ctrl-X & Shift-Del cuts the selection, i.e. it puts the selection into the clipboard and deletes the selection.
- Ctrl-C & Ctrl-Insert copies the selection, i.e. it puts the selection into the clipboard.
- Ctrl-V & Shift-Insert pastes any content of the clipboard onto the canvas.
- Ctrl-A selects all nodes and links.
- Ctrl-Z & Alt-Backspace undoes the last action.
- Ctrl-Y & Alt-Shift-Backspace redoes the next undone action.
- Up & Down & Left & Right (arrow keys) scrolls the diagram if nothing is selected. Otherwise, it moves the selection.

- PageUp & PageDown scrolls the diagram.
- Home & End call scrolls the diagram.
- Space scrolls to the selected part.
- Keypad– (minus) decreases zoom.
- Keypad-+ (plus) increases zoom.
- Ctrl-0 resets the zoom.
- Shift-Z zooms to fit the whole diagram; repeat to return to the original scale and position.
- Esc stops any command currently being performed.

H.4 Mouse actions

By clicking on nodes and links they are selected and any previous selection is deselected. Clicking on the background deselects everything. Pressing the Ctrl key toggles the selection of the target node or link. Pressing the Shift key only adds nodes and links to the selection.

Selected nodes can be rotated by dragging the Rotate adornment (fig. H.11). Selected parts can be moved with the mouse by dragging them.

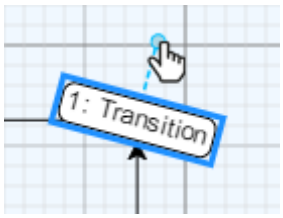

FIGURE H.11: Rotate adornment

Dragging the background of the diagram will scroll the diagram.

Clicking the background without moving the mouse and then dragging the mouse will select everything within the displayed rectangle (fig. H.12).

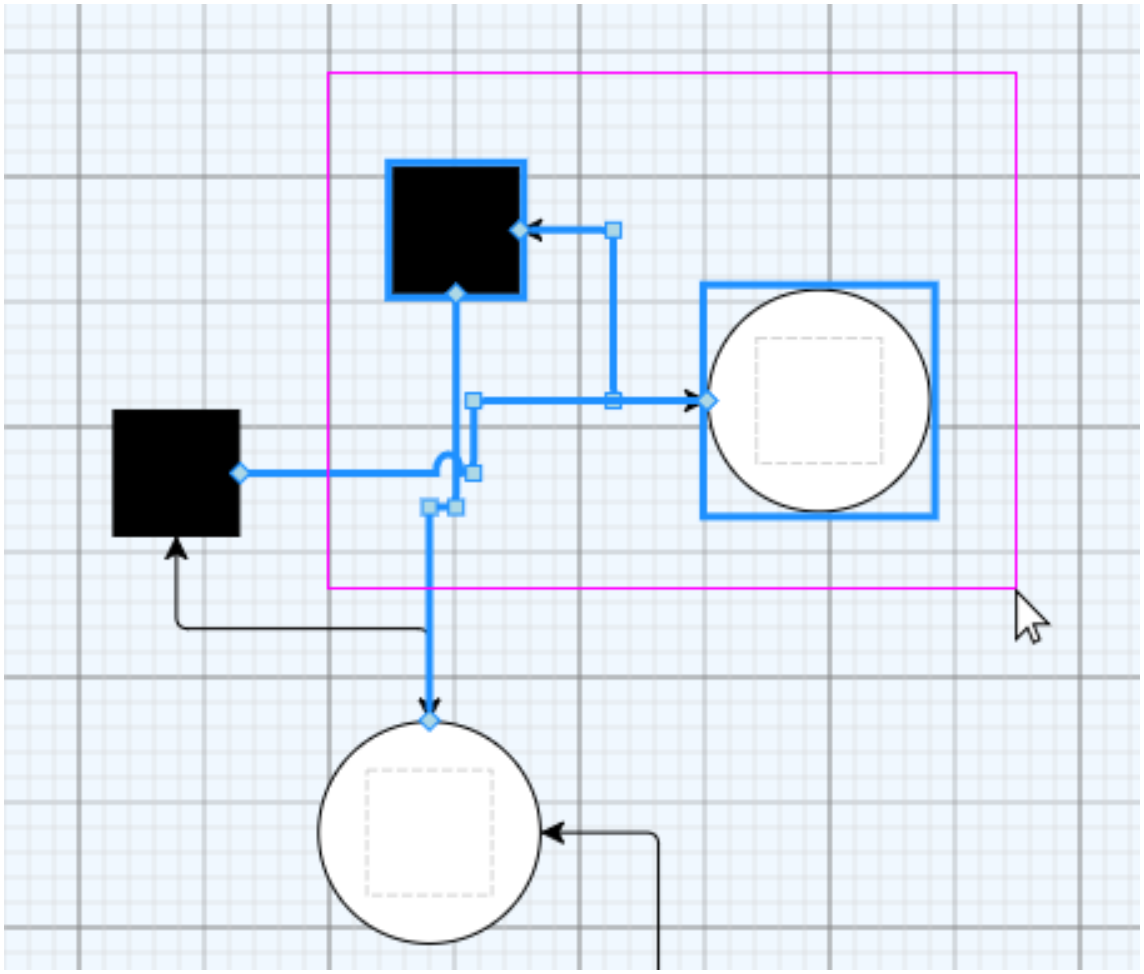

FIGURE H.12: Drag selection zone

Using the mouse wheel will scroll vertically. Using the mouse wheel while pressing Shift will scroll horizontally. Using the mouse wheel while pressing Ctrl will zoom.

The endpoints of links can be dragged to connect them to a node. Doing this in the CSM UI mode will also synchronize the ASM instance. Whether a link can be connected to a node depends on whether the type of the node is compatible with the start, respectively end point type of the link. Compatibility means that the link point type is the same or an ancestor of the node type. See fig. H.13 for a compatible link and node and fig. H.14 for an incompatible link and node.

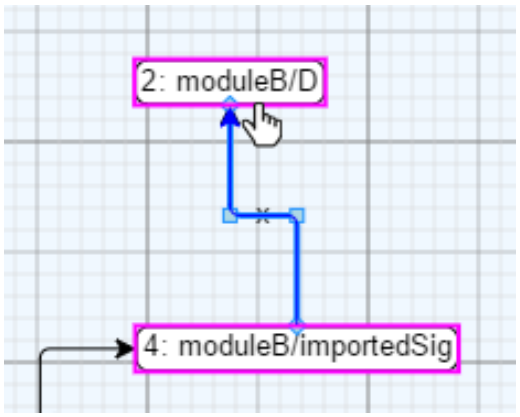

FIGURE H.13: Compatible link and node types

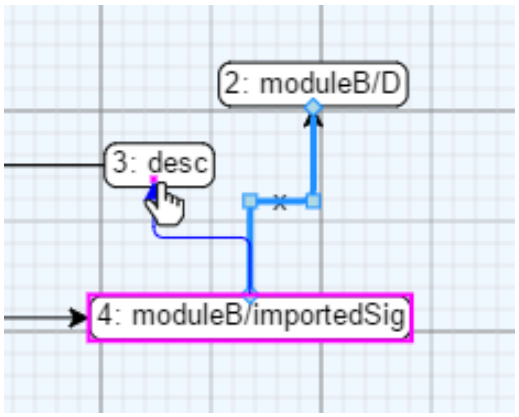

FIGURE H.14: Incompatible link and node types

Bibliography

[1] Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt. Designing languages using lightning. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 77–82. ACM, 2015.

[2] Alistair Sutcliffe and AG Sutcliffe. *The Domain Theory: Patterns for knowledge and software reuse*. CRC Press, 2002.

[3] Anneke G. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008. ISBN 0-321-55345-4 978-0-321-55345-4.

[4] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[5] Alloy. http://alloy.mit.edu/alloy/.

[6] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[7] Loïc Gammaitoni. Lightning. https://lightning.gforge.uni.lu/.

[8] Loïc Gammaitoni and Pierre Kelsen. F-Alloy: An Alloy Based Model Transformation Language. In *Theory and Practice of Model Transformations*, pages 166–180. Springer, 2015.

[9] John C Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li. Generating domain-specific visual language tools from abstract visual specifications. *Software Engineering, IEEE Transactions on*, 39(4):487–515, 2013.

[10] Hanns-Alexander Dietrich, Dominic Breuker, Matthias Steinhorst, Patrick Delfmann, and Jörg Becker. Developing Graphical Model Editors for Meta-Modelling Tools. 2013.

[11] Ramesh Nagappan, Robert Skoczylas, and Rima Patel Sriganesh. *Developing Java web services: architecting and developing secure web services using Java*. John Wiley & Sons, 2003.

[12] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly Media, Inc., 2008.

[13] Steve Francia. REST vs SOAP, the difference between soap and rest : spf13.com. http://spf13.com/post/soap-vs-rest, January 2010.

[14] HTTP/1.1: Method Definitions. https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[15] Dragan Gaić. Top 8 Java RESTful Micro Frameworks. http://www.gajotres.net/best-available-java-restful-micro-frameworks/.

[16] GeneSEZ: Welcome to GeneSEZ. http://www.genesez.org/.

[17] 14.2. The GeneSEZ JPA UML Profile. http://documentation.genesez.org/en/de.genesez.uml.profile.jpa.html.

[18] RequireJS. http://requirejs.org/, .

[19] lodash. https://lodash.com/.

[20] jQuery Foundation jquery.org. jQuery, .

[21] Page.js by visionmedia. https://visionmedia.github.io/page.js/.

[22] Bower. https://bower.io/, .

[23] Jade - Template Engine. http://jade-lang.com/, .

[24] Grunt: The JavaScript Task Runner. http://gruntjs.com/, .

[25] The web's scaffolding tool for modern webapps | Yeoman. http://yeoman.io/, .

[26] zguillez/generator-base-polymer. https://github.com/zguillez/generator-base-polymer.

[27] Bootstrap · The world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com/, .

[28] jQuery Foundation jquery.org. jQuery Mobile, .

[29] AngularJS — Superheroic JavaScript MVW Framework. https://angularjs.org/, .

[30] Welcome - Polymer Project. https://www.polymer-project.org/1.0/, .

[31] A JavaScript library for building user interfaces | React. https://facebook.github.io/react/, .

[32] React-Bootstrap. https://react-bootstrap.github.io/, .

[33] Angular vs. React - the tie breaker. https://www.airpair.com/angularjs/posts/angular-vs-react-the-tie-breaker, .

[34] GoJS Diagrams for JavaScript and HTML, by Northwoods Software. http://gojs.net/latest/index.html, .

[35] Northwoods Academic Use. http://www.nwoods.com/sales/academic-use.html, .

[36] NORTHWOODS SOFTWARE CORPORATION. http://gojs.net/latest/doc/license.html, .

[37] Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey. Verifying modelling languages using lightning: a case study. *MoDeVVa 2014: Model-Driven Engineering, Verification and Validation*, pages 19–28, 2014.

[38] Eugene Syriani. AToMPM. http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm.

[39] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25. Citeseer, 2013.

[40] Graphical editors based on models: GMF/EUGENIA/Sirius, .

[41] org.eclipse.gmf-tooling.git - gmf tooling project repository. http://git.eclipse.org/c/gmf-tooling/org.eclipse.gmf-tooling.git/tree/plugins?id=2acb3fe7775eca8223318d590350282250975 63b.

[42] Dimitrios S Kolovos, Antonio García-Domínguez, Louis M Rose, and Richard F Paige. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, pages 1–27, 2015.

[43] John Grundy, John Hosking, Jun Huh, and Karen Na-Liu Li. Marama: An Eclipse Meta-toolset for Generating Multi-view Environments. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 819–822, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368210.

[44] John Grundy, John Hosking, Shuping Cao, Denjin Zhao, Nianping Zhu, Ewan Tempero, and Hermann Stoeckle. Experiences developing architectures for realizing thin-client diagram editing tools. *Software: Practice and Experience*, 37(12):1245–1283, 2007.

[45] Welcome - Marama - Wiki. https://wiki.auckland.ac.nz/display/csidst/Welcome, 2013.

[46] GoJS Commands – Northwoods Software. http://gojs.net/latest/intro/commands.html, .