DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Model Validation in Graphical Cloud Based Editors

Simon Graband

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Model Validation in Graphical Cloud Based Editors

Modellvalidierung in grafischen cloudbasierten Editoren

Author:	Simon Graband
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Tri Huynh, M.Sc.
Submission Date:	13.11.2020

I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 13.11.2020 Simon Graband

Acknowledgments

I would, first and foremost, like to thank my advisor Tri Huynh for supporting me throughout this thesis with constructive feedback and continuous input on how to improve my work.

Additionally, i would like to thank the whole EclipseSource staff, especially Jonas Helming, for providing such a great environment for learning and working on this thesis.

Furthermore, i want to thank all of the participants of the evaluation for taking the time and providing me with useful feedback from many different perspectives.

I also would like to thank Prof. Dr. Florian Matthes for the feedback he provided and letting me pursue this topic at his chair.

Last but definitely not least, i would like to thank my friends and family, especially my girlfriend Sophie, for always supporting me throughout the last six months.

Abstract

Many modeling toolchains are currently in the transition process from desktop-based to cloud-based solutions. Model validation is a key value in many model toolchains and allows for early error detection in the design process. However, model validation in the context of client-server scenarios poses new challenges compared to desktop-based solutions. Some validation constraints should result in immediate feedback to the user, which suggests a client-side validation, since this saves a roundtrip to the server allowing for a quicker response. Other validation constraints need more computation time and access to the underlying model, which suggests a validation on the server, where the model is stored, and then, after validating the model, returning the validation result to the client. In this thesis, a model validation approach for cloud-based modeling tools will be researched. Furthermore, a concept to describe and execute model constraints on the client and server, depending on the specific use case, will be designed, implemented and evaluated. As a Proof of Concept (PoC) for the model validation in the cloud a Case Management Model and Notation (CMMN) editor will be implemented to test the functionality and usability of the framework.

Contents

Acknowledgments **iii**

Abstract **iv**

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Objectives and Research Questions . . . 1
1.3. Thesis Structure . . . 2

2. Background **3**
2.1. Terminology . . . 3
2.1.1. Model . . . 3
2.1.2. Metamodel . . . 3
2.1.3. Constraint . . . 4
2.1.4. Model Validation Framework . . . 4
2.1.5. Case Management Model and Notation . . . 4
2.1.6. EMF.cloud . . . 4
2.2. Related Work . . . 5
2.2.1. Object Constraint Language . . . 5
2.2.2. Java Bean Validation . . . 6
2.2.3. XML Schema . . . 11
2.2.4. Eclipse Modelling Framework . . . 14

3. Requirements and Challenges for a Model Validation Framework **17**
3.1. Requirements for Model Validation . . . 17
3.1.1. Functional Requirements . . . 17
3.1.2. Non-Functional Requirements . . . 18
3.2. Challenges of Model Validation . . . 20
3.2.1. Categorization of Constraints . . . 20
3.2.2. Persistence . . . 23

4. A Framework for Model Validation in cloud based Tools **24**
4.1. Architecture Goals . . . 24
4.1.1. Architecture . . . 24
4.1.2. Constraint Validation Stategy . . . 30
4.1.3. Persistence of Error Feedback . . . 31
4.1.4. Error Feedback . . . 31

4.2. Implementation of the Validation Framework . . . 31
4.2.1. Modelserver Validation . . . 32
4.2.2. Validation Framework . . . 35
4.2.3. Relevance of Validation Client . . . 40

5. Evaluation **43**
5.1. CMMN editor . . . 43
5.1.1. CMMN Constraints . . . 43
5.1.2. CMMN Metamodel . . . 44
5.1.3. Editor Implementation . . . 44
5.2. User Study . . . 49
5.2.1. Test Setup . . . 50
5.2.2. Results . . . 53
5.2.3. Benefits . . . 57
5.2.4. Limitations . . . 57

6. Conclusion and Future Work **59**

A. General Addenda **60**
A.1. Model Validation Framework Code . . . 60
A.2. Modelserver Code . . . 60
A.3. CMMN Editor Code . . . 60
A.4. Evaluation Results . . . 60
A.5. CMMN Metamodel . . . 60

List of Figures **62**

Acronyms **63**

Bibliography **65**

1. Introduction

1.1. Motivation

Model validation is a key factor in model-driven engineering (MDE), which shifts the priority of software development from coding first towards modelling first [1]. This enables the generation of model elements, which are essential to the analysis or development of software, such as code, dependency graphs, etc. Therefore, it is important to have consistent and correct models to maintain the accuracy of generated artifacts. To fulfil this requirement, model validation is considered as one of the two most common model quality improvement techniques in MDE, alongside with model verification [1]. In recent years, modelling systems are influenced by the rapid adaption of cloud based solutions for software [2]. In this trend, alongside with consumer products such as Microsoft Office or E-Mail clients, development tools also offer services on the cloud, with Visual Studio Code [3] and Eclipse Theia [4] as the typical examples. The advantages of a cloud-based approach are scalability, portability and zero installation time [5]. This is especially beneficial, when working with multiple or even external parties, since no installation effort is required by any party. One specific example is an external domain expert reviewing a model. If no time is spent for installing the modelling tool, the costs and overhead of the review process is reduced, while the high availability and portability improve the overall working experience. Since the main advantages of modelling and model validation are cost reduction and accurate planning [6], the combination with a cloud-based approach can lead to an even greater cost reduction for software development.

1.2. Objectives and Research Questions

The overall goal of this thesis is to create a framework for seamlessly integrating model validation in graphical cloud-based editors, that fosters and simplifies the system modelling task. To define the scope of this thesis, the following research questions were defined:

1. What are existing solutions for constraint validation?
2. How can those solutions be applied to a cloud-based tool?
3. What user benefits can be measured with a user study?

A special focus, while answering the second question, is the split between client and server in cloud-based tools and how the different components of the model validation framework are split between the two. For the user study a Proof of Concept (PoC) for the model validation is being implemented, to test the functionality as well as the usability of the framework.

1.3. Thesis Structure

In order to answer the previously defined research question, some of the existing model validation technologies will be researched first, before applying the learned concepts to a cloud-based architecture. After specifying the requirements, a model validation framework for graphical cloud based editors will be implemented. Last but not least, the advantages the model validation framework offers shall be measured, for users as well as other developers. After implementing the framework, it will be integrated and tested in a Case Management Model and Notation (CMMN) editor to prove its functionality, as well as figuring out the benefits and the limitations, and therefore possible future work, of the proposed solution.

2. Background

First of all, the most important terminology, regarding model validation, will be explained. After that, some of the strengths and weaknesses of existing solutions will be analyzed, to derive relevant concepts that will be adapted for the cloud-based approach proposed in this thesis.

2.1. Terminology

2.1.1. Model

First, to avoid confusion with similar terms, like mathematical models, the term model will be defined. Basically, a model provides an abstraction of a complex subject [7][8]. This abstraction should contain less complexity, than the subject, by only focusing on the essential aspects of the subject [7]. Essential aspects are the functionality to be implemented, while the non-essential aspects are, mainly but not limited to, the technical platform that the essential aspects are deployed to [7]. Simplified, a model is the building plan to a project, without defining the technical specifications of the project, for example which programming language the software will be coded in. However, this approach is not restricted to software, (business) processes may also be modeled before being implemented. For example, the language CMMN offers features to model the case management of a specific use case [9]. Models can be created in multiple ways. They can either be represented in a tree-view, plain text or, especially relevant for this thesis, graphically. For the graphical representation, a diagram is used to alter the model. To align the graphical representation with the underlying model, both components need to follow the specification of the modelling language.

2.1.2. Metamodel

This specification is defined in a so called metamodel [8]. A metamodel can be described, as a model for another model or a model of a modelling language [8]. For example, the specification of the language CMMN can be defined in a metamodel, which models the features of the CMMN language. The metamodel contains information like, which objects are available, which attributes these objects can have, how can they be connected and what rules the objects and their attributes need to follow. Therefore, the metamodel has the same relationship to a model as the model has to its instance. It describes the way a model can be build, without defining how exactly it is being build. The same relationship can be applied to describing metamodels. A meta-metamodel defines how a metamodel can be build.

2.1.3. Constraint

A constraint is a condition that must be true in order for the model to still be valid. For example, a constraint could be added to the model of a house, stating that a house needs to have at least one room. If this is the case, then any model that contains a house, which does not contain a room, should be flagged as invalid. Since model validation validates the different constraints, that were coded for a model, the terminology model validation and constraint validation can be used interchangeable. In general, constraints can be divided into two different categories: language-specific and user-defined. Language-specific constraints are all the rules, that are a part of the documentation of a specific language, for example the definition of valid connections between different objects in the model. On the other hand, user-defined constraints are rules that are defined by the user of the model validation itself. For example, the user may wish to add some restrictions to the value of an attribute.

2.1.4. Model Validation Framework

The goal of a model validation framework, is to validate all the constraints defined for a model and return the results in a usable format, for example an error message. Therefore, the model validation framework consists of different components. The first component is the model or constraint validator, which is the component, that validates the model against the defined constraints. Additionally, the model validation framework needs a component that displays the feedback of the model validator to the user. Last but not least, the framework needs an editor, where the model can be altered. In this thesis the editor will be a graphical modeling editor.

2.1.5. Case Management Model and Notation

The Case Management Model and Notation (CMMN) language will be used as a PoC later on in this thesis. CMMN is a complementary language to Business Process Model and Notation (BPMN) [10], but focuses, as the name suggests, on Case Management [11]. Some of its use cases contain "diagnosis in healthcare", "permitting in government" and "claim processing in insurance" [9]. The specifics of CMMN relevant to this thesis will be described in 5.1, when the CMMN editor will be implemented. CMMN offers a variety of language-specific constraints and a need for user-defined constraints, as its application are widely spread. Therefore, the customization of the language and its constraints is important, to ensure proper support for these use cases. Furthermore, the graphical representation of a CMMN model, influences the model validation, as some elements can be nested within each other. This is relevant for this thesis, as graphical aspects of the model need to be validated as well.

2.1.6. EMF.cloud

Since the goal of this thesis is to enable model validation in the cloud, the EMF.cloud project [12] will be helpful for the implementation of this thesis. The EMF.cloud project contains "components and technologies" that aim at "making the Eclipse Modelling Framework (EMF)

and its benefits available in the web and cloud" [12]. The goal is not to re-implement, but to use the existing technologies and deploying them in a cloud-based environment [12]. Similar, the goal for this thesis is not to implement a model validation framework from scratch, but reusing existing technologies and offer a way to use them in the cloud. Whether or not the Eclipse Modelling Framework (EMF) is fitting for this use case will be discussed in section 2.2. If the EMF is applicable for this use case, the EMF.cloud components can be used and extended as needed.

2.2. Related Work

2.2.1. Object Constraint Language

The Object Constraint Language (OCL) is a constraint language for the Unified Modelling Language (UML) standard [13]. OCL allows the software architect to specify constraints in a purely declarative way, since OCL specifies constraints on a conceptual level and does not specify the implementation of these constraints.

Language

The language is designed in a way, that it closely resembles object-oriented programming languages. It supports attributes of objects, navigation between linked objects and operation calls. OCL constraints can be associated with any kind of *ModelElement* in an UML diagram [14]. In OCL a constraint consists of a body, which consists of any valid OCL expression. Those expressions are typed. The type specifies the result type and valid operations. Data types include the basic types *Boolean*, *Integer*, *Real* and *String* [14]. Valid operations for those types contain logical (*and*, *or*, *not*), arithmetic (+, -, *) and string manipulation (*concat*, *substring*) operations. Classes, data types and interfaces are also available inside OCL, if they are described in the underlying UML diagram. Those are called *Classifiers* and operations on those need to be defined in the UML model. There are three different data types for collections in OCL, *Set(T)* for mathematical sets, *Bag(T)* for multisets and *Sequence(T)* for lists. T is a placeholder for the type of values in the collection and needs to be defined in a concrete collection type, e.g., *Set(Integer)* [14].
Each constraint needs a context that defines, which model element the constraint is attached to. For example, invariants need a *Classifier* as a context, whereas pre- and postconditions need a *Classifier* and an operation as a context. Invariants are conditions that need to be true during the lifetime of a system for all instances of a given *Classifier* and need to return a *Boolean* as a result [14]. In the following example, the number of rooms in a house shall be greater than zero.

```
context h : House inv:
    self.rooms > 0
```

With *self.rooms* the expression refers to the attribute *rooms* of an object of the context, in this case the context is *House*.
OCL also allows validation of states before and after operation calls. In the following example, the method *expand(roomsAmount:Integer)* of the *Classifier House* is validated. Before the method is called, the number of rooms needs to be greater than zero and after the operation has been called, the number of rooms needs to be bigger than before.

```
context House::expand(roomsAmount : Integer) : Integer
    pre: roomsAmount > 0
    post: self.rooms = self.rooms@pre + roomsAmount
```

The *@pre* notation is only allowed in postconditions, since those are the only place where more than one system state, before and after the operation, can be referenced [14].
To summarize, OCL is a very powerful language to describe constraints for UML models. But there are interpreters for other languages as well, like Eclipse OCL for EMF-based models [15]. However, this also leads to some disadvantages. Since OCL is only declarative and does not specify the implementation approach, a lot of tools have been developed, that try to validate OCL constraints against many different languages. There are various tools available, all with different functionalities. This can be problematic, because a tool for a modelling language will not offer all of the functionalities, that are needed. To solve this problem Richters [14] proposed a metamodel for OCL, that is deeply tied to the UML metamodel, which allows for better implementation of an OCL tool, as it defines all of the abstract syntax, to better generalize the rules of OCL. The disadvantage to this approach is, a metamodel for every modelling language needs to be created, since the OCL metamodel needs to be deeply tied with the metamodel of the modelling language. This means two metamodels need to be created in order to design a model editor. One for the language itself and one for the OCL interpreter. Creating the editor as well as the OCL interpreter means additional work, which is the biggest downside to using OCL as the constraint language for a model validation framework.

2.2.2. Java Bean Validation

To validate data in a Java programs Bean Validation for Java was introduced [16]. In Java Bean Validation, "annotations can be defined for types, methods, fields or other constraint annotations in case of composition" [16]. These annotations are used to define the parameters of the constraints, that should be validated. Apart from the standard annotation, custom annotations with custom coded constraints can be created. [16].

Language

Like mentioned before, Java Bean Validation uses annotations to define constraints. The constraint can either be taken from the standard Java specification request (JSR) annotations [16] or be created by coding the constraint in Java. The standard constraints contain annotations like *@NotNull*, *@AssertTrue*, *@Size*, *@Min*, *@Max*, *@Email*, *@NotEmpty* and *@NotBlank*.

```
@ZipCodeCityCoherenceChecker
public class Address {
    @NotNull @Size(max=30)
    private String addressline1;

    @Size(max=30)
    private String addressline2;

    private String zipCode;

    private String city;

    //Other getters and setters left our

    @Size(max=30) @NotNull
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}
```

In this example, taken from the Java Bean proposal [16], the field *addressline*1 needs to be not null and the size may not be longer then 30 characters. The same is applied for *addressline2*, however this field can be null, it just cannot exceed the 30 character limit. The result of the method *getCity()* is also validated for being not null and not containing more than 30 characters. Last but not least the whole class has a *@ZipCodeCityCoherenceChecker* annotation. This means, that a created Address object is validated with the *isValid()* method of the *@ZipCodeCityCoherenceChecker* constraint annotation, which is not a JSR Standard annotation [16]. Such a constraint annotation can be expressed with a constraint definition, which needs to be annotated with *javax.validation.Constraint* and contain *RUNTIME*, as its retention policy [16]. To describe the constraint definition, attributes may be added, which values can be specified once the annotation is applied to an element. There are three reserved names, which cannot be used, for these attributes: *message*, *groups* and *payload*. Furthermore, attribute names may not start with *valid*. Apart from those every other name is useable [16]. The *message* attribute defines the message that is returned if the constraint criteria is not met. *Groups* can be used to alter the order in which the constraints are evaluated and *payload* is used to offer some more information about the error, for example the severity [16]. All of the attributes can be assigned a default value, if no other value is set when the annotation is assigned to an element [16].

```
@Constraint(validatedBy = CreditCardNumberValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface CreditCardNumber {
    String message() default
        "{org.eclipse.constraints.CreditCardNumber.error}";
    Class<?>[] groups() default {};
    Class<? extends ConstraintPayload>[] payload() default {};
}
```

In the example, a constraint definition is set. The *@Constraint* annotation defines which class contains the code for the validation and *@target* specifies which elements the constraint can be applied to. Also for *message*, *groups* and *payload* the default options are defined. For the message a resource is defined, which allows for a more consistent way of storing error message and an easy way to offer different language packs (internationalisation) [17] for the error messages [16]. This lays out a good foundation for a model validation framework as the error messages can be managed in an easy way and also be internationalized [17].
Another feature of Java Bean Validation is the composition of constraints. A constraint may be annotated with another constraint [16] . In the following example, the constraint defined in *CreditCardNumberValidator.class* is composed with the *@Pattern* and *@Size* constraints.

```
@Pattern(regexp="[0-9]*")
@Size(min=16, max=16)
@Constraint(validatedBy = CreditCardNumberValidator.class)
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface CreditCardNumber {
    String message() default "Wrong credit card format";
    Class<?>[] groups() default {};
    Class<? extends ConstraintPayload>[] payload() default {};
}
```

This means that the value entered does not only need to match the specified constraint but also the composed, additionally annotated constraints. In this case the credit card number needs to apply to the specified rules in *CreditCardNumberValidator.class*, contain only of numbers, and have a fixed length of 16. This allows the description to combine standard annotations with coded constraints, which eases the use of Java Bean Validation. The values of composed annotations can also be set via a attribute when the annotation, that is composed, is set [16]. Similar to before, default values can be set in case no value was assigned.

```
@Pattern(regexp="[0-9]*")
@Size
@Constraint(validatedBy = CreditCardNumberValidator.class)
```

```
@Documented
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface CreditCardNumber {
    String message() default "Wrong credit card format";
    Class<?>[] groups() default {};
    Class<? extends ConstraintPayload>[] payload() default {};

    @OverridesAttribute.List( {
           @OverridesAttribute(constraint=Size.class, name="min"),
           @OverridesAttribute(constraint=Size.class, name="max") } )
    int size() default 16;

    @OverridesAttribute(constraint=Size.class, name="message")
    String sizeMessage() default "Credit card number needs to be 16 character
        long";

    @OverridesAttribute(constraint=Pattern.class, name="message")
    String numberMessage() default "Credit card number should only contain of
        numbers";

    @Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        FrenchZipcode[] value();
    }
}
```

In this case the size and the messages can still be defined in the annotation, however if they are not set, they will default to the values specified in the code above. To validate a constraint a validation implementation for that constraint is needed. Such a validator implements the following *ConstraintValidator* class [16]:

```
/**
* Defines the logic to validate a given constraint A
* for a given object type T.
* Implementations must comply to the following restriction:
* T must resolve to a non parameterized type
*
* @author Emmanuel Bernard
* @author Hardy Ferentschik
*/
```

```
public interface ConstraintValidator<A extends Annotation, T> {
    /**
    * Initialize the validator in preparation for isValid calls.
    * The constraint annotation for a given constraint declaration
    * is passed.
    * <p/>
    * This method is guaranteed to be called before any of the other Constraint
    * implementation methods
    *
    * @param constraintAnnotation annotation instance for a given constraint
        declaration
    */
    void initialize(A constraintAnnotation);
    /**
    * Implement the validation logic.
    * <code>value</code> state must not be altered.
    *
    * @param value object to validate
    * @param constraintValidatorContext context in which the constraint is
        evaluated
    *
    * @return false if <code>value</code> does not pass the constraint
    */
    boolean isValid(T value, ConstraintValidatorContext
        constraintValidatorContext);
}
```

The *isValid()* method should return true if the value is valid. In the following example the an empty check for the String *value* is coded.

```
public boolean isValid(String value, ConstraintValidatorContext context) {
  if ( value == null ) return false;
  return value.length() > 0;
}
```

One thing to note here is that best practices suggest to split different constraints into different validators. So in the case above the null check should not be validated together with the *length* and rather be validated with the *@NotNull* annotation. This allows for better error message handling and helps to keep the constraints separated from each other.
To summarize, Java Bean Validation offers a variety of functionalities to validate constraints for a model validation framework. Constraints are defined with annotations, which makes it easy to see the constraints applied to a specific field. Furthermore, constraints can be composed together while keeping separate messages, for better error feedback. One disadvantage of

this approach is that the model needs to stored in Java itself, for the framework to work properly. One workaround for this maybe to create the elements in Java first and validating them, before storing them in another format. However, this solution requires some additional work, and if the stack of the editor allows for easier model saving, this approach might be more laborious than a native solution.

2.2.3. XML Schema

When storing models in eXtensible Markup Language (XML) format, there are quite a few tools for model validation by describing data structures and constraints. A selection fo those tools are XML Schema Definition (XSD) or XML Schema [18], DTD, XDR, SOX, Schematron, DSD and DCD [19]. However, this thesis will not focus on each of these languages, but rather focus on XSD, and its features, to see whether or not it is a good fit for model validation in graphical cloud based editors or not. If XSD is indeed a good fit a decision for one of the other languages may be applicable as well, based on the features of that language.

Language

In general, the syntax of XSD is XML syntax. This is the case for all of the above mentioned languages, except for DTD [19]. This is advantageous for the user, as there is no need to learn a new syntax, because the syntax is the same as for the model file itself. Additionally, the schema file also has the same properties as a XML file and can therefore be read in existing XML applications, be stored in a XML storage system with other XML documents (like the model for example), as well as being extensible [19]. XSD supports namespaces, which help to reuse previously defined elements and therefore further enable extensibility. Similar to namespaces XSD can include or import other schema fragments and consume them for better maintenance and readability [19]. The keyword *include* references a schema fragment that is within the same namespace, whereas *import* refers to a fragment outside the current namespace. XSD even allows imports from multiple namespaces [19]. In the example below the complex type *person* is defined. A *person* consists of a *name* and a *birthday*, the later has the type *b:birthday*, the prefix *b* points to the *NamespaceB*. Additionally, XSD allows the creation of a general namespace without the definition of a prefix, in this case *NamespaceA*.

```
<schema xmlns='NamespaceA'
          xmlns:b='NamespaceB'>
    <element name='person'>
        <complexType>
            <element name='name' type='name'>
            <element name='birthday' type='b:birthday'>
        </complexType>
    </element>
</schema>
```

For data types XSD supports built-in types, user-defined types, domain constraints and nullable objects. For built-int types it supports 37 of the most commonly used programming languages types. Additionally, the user is able to create new data types by deriving a new type from the built-in or other user-defined types [19]. A domain constraint for a datatype can be set in XSD by applying facets or regular expressions to a specific to the datatype [19]. Facets are predefined options for restricting the value of a type. A full list of these facets can be found on the W3Schools website [20]. An example for a facet would be the length which restricts the length of the datatype value by the value that is set to the facet. Lastly, XSD has a nullable tag for attributes that, if set to true, allows null values for the attribute [19]. Below, two examples for restricting a simple data type to limit the content of the defined elements are implemented. The first example defines a set of values (*enumeration*) that can be used for the type, in that case *color*.

```
<xs:element name="color">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Red"/>
      <xs:enumeration value="Green"/>
      <xs:enumeration value="Blue"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The second example restricts a string to be a mobile phone number (a number with at most 15 digits).

```
<xs:element name="mobilePhoneNumber">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]"/>
      <xs:maxLength value="15"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

All of these features for data types set a very good foundation for a model validation framework as the facets and regular expressions can be used to define complex data types for the features of the modelling language and can then be validated once a model is created.
In XSD a default value can also be set for attributes [19]. Which can be beneficial to a model validation framework as it gives the model a valid value for the attribute that can be set when the attribute is created to ensure the model stays valid. Furthermore, a required tag for attributes can be set [19]. If the tag is set, the parent element needs a value for that attribute otherwise the model is not valid. Whereas if the tag is not set, the attribute must not be set.

For model validation that is important, because it is an easy way to check whether or not the element is missing a required attribute or not.

```
<xs:attribute name="mobilePhoneNumber" default="000000000000000"
    use="required" />
```

XSD additionally can define facets for a specific attribute, if the facets were not already defined in a used user-defined type, to only allow it to have certain values. For elements, XSD can define whether or not sub-elements need to be ordered or can be unordered [19]. This is also an important feature to have for a model validation framework as the sequence of sub-elements might alter the validness of a element. For example, a person has a first name and a last name. If those are not ordered, some elements might initialize the first name first or the other way around, hence the validator might return wrong results, for example when checking for duplicate last names. Elements can also be assigned a minimum and a maximum occurrence, which indicate, that there should be at least the minimum amount and not more than the maximum amount of sub-elements nested in a element [19]. This can help to validate relationships between different elements as there may be restrictions about how many references to other elements are allowed. For example, a person may have not more than one spouse.

```
<xs:element name="person">
    <xs:attribute name="spouse" maxOccurs="1"/>
</xs:element>
```

XSD allows for inheritance to create sub-types. To be precise it allows inheritance to restrict simple and complex types and allows for extension of complex types [19]. When restricting via inheritance the possible values for the base type are reduced. For example, a simple type like a String can be restricted to some specific values like different kinds of colors [19]. For complex types the attribute occurrence might be changed to higher or lower value. When extending a complex type, a new attribute might be added. For example, a *Graduate* might be based on the type *Person* but will have an additional attribute called *Degree*. On top of that, attributes and elements can be defined as unique, which will prevent the same value from being used again. The example below ensures, that the address of a person contains a unique *mobilePhoneNumber* [19].

```
<unique>
  <selector>person/addr</selector>
  <field>mobilePhoneNumber</field>
</unique>
```

Overall XSD is a very useful tool to describe validation rules for models. It supports a lot of features that can be used to define constraints for the modelling language and the constraints are defined in the same format as the model itself, which makes the development easier. The fact, that there are multiple languages available that work similar to XSD, might also improve

the usage of XML schema as a language may be chosen that trades some functionality, that is not needed, for some easier syntax. However, the validation of the constraints defined in XSD needs a implementation of how these rules are validated against a XML model. The implementation might not be to complicated, since both the model and the constraints are stored in the same format, but the implementation is still required.

2.2.4. Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) offers a native way to validate created EMF models. EMF uses the Essential MOF (EMOF) specification, which is a variant of the Meta Object Facility (MOF) specification [21]. To be precise EMF uses the Ecore meta-metamodel [21]. The Ecore meta-metamodel is used to describe the metamodel of the modelling language being used. The created meta-metamodel can then be used to create Java code for the metamodel, which then can be used to create models for the modelling language with an existing EMF editor or to create a custom editor for the modelling language. Ecore has some functionalities that let you assign specific constraints to elements. This in combination with the functionality to create code for custom editor, makes an interesting use case for model validation as the metamodel for the modelling language is sufficient to create a custom editor and also apply the validation rules. Additionally, EMF offers the functionality to validate its models against the metamodel, that the model is based on. This is called EMF validation.

Ecore

To further describe the features of the EMF approach, this section will focus on its meta-metamodel language Ecore. To illustrate the Ecore language the following image is a simplified version of the its metamodel [22].

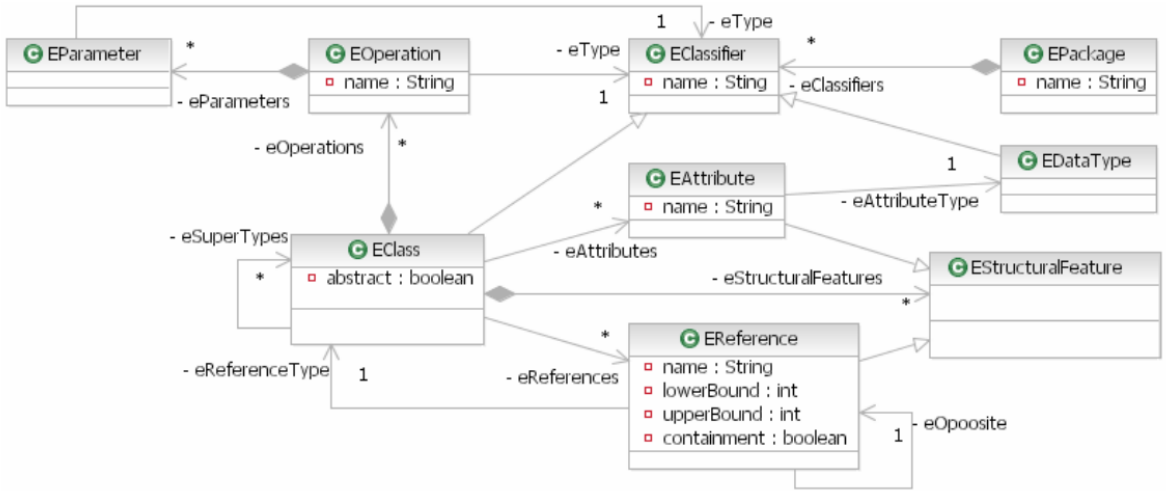

Figure 2.1.: Simplified Ecore Metamodel [22]

All of the elements in the metamodel are of the type *EModelElement*, which was left out to improve the visibilty of the model [22]. The root object of an Ecore model is an *EPackage* which contains a list of *EClassifiers* [23]. An *EClassifier* is either an *EClass*, or an *EDataType*. An *EDataType* can be used to create new data types that can be used inside the Ecore model. There is a list of predefined *EDataTypes*, like *EInteger*, *EString* and *EBoolean* [23], but custom data types can be added as well. For example, a data type can be defined, which is based on the Java specification of a String, which is then limited to eight characters, named *String8Char*. This *EDataType* can then be applied to attributes in the Ecore model.

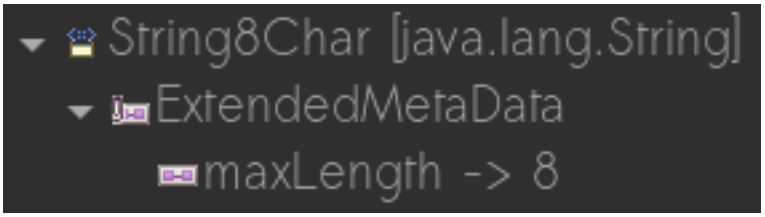

Figure 2.2.: Custom EDataType

The restriction of the String is done by applying an *ExtendedMetaData* object to the data type and setting the *maxLength* facet to 8 [24]. Ecore supports a variety of these facets, which eases the creation of custom data types. If a data type with specific values should be created, an *EEnum* can be created. A *EEnum* defines several values that are part of the new data type. For example, an *EEnum Color* with the values *Red*, *Green* and *Blue* can be defined. This can then be applied as a type to other attributes in the model.

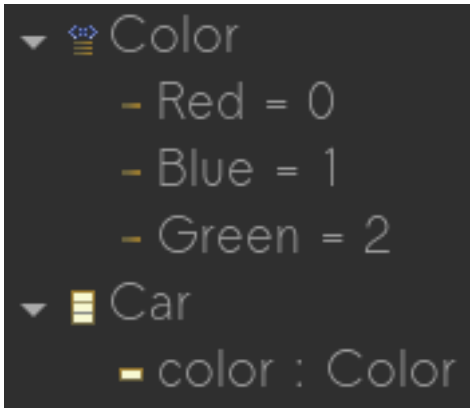

Figure 2.3.: Assigning an EEnum as a Type

EClasses on the other hand define concepts of the modelled language, For example, a property in UML [22]. Apart from a basic *EClass*, they can also be declared abstract or an interface. Therefore, *EClasses* can be part of inheritance hierarchies [23]. To further describe *EClasses*, they contain two additional lists, one consisting of *EStructuralFeatures* the other one of *EOperations*. *ESructuralFeatures* are either *EAttributes* or *EReferences* and, like *EClassifiers*, have a name. *EAttributes* are typed with a *EDataType*, and therefore can be assigned any

default or custom created *DataType* and *EEnum*. *EReferences* are assigned an *EClass* as a type. Therefore, *EStructuralFeatures* help define properties of the *EClass* they are assigned to. Whereas *EOperations* define the behavior of the *EClass* [22]. *EOperations* return *EClassifiers* as a type and have a list of *EParameters*. So a full Ecore model for *Vehicles* containing a *Color* and *tyreCount* is pictured in 2.4.

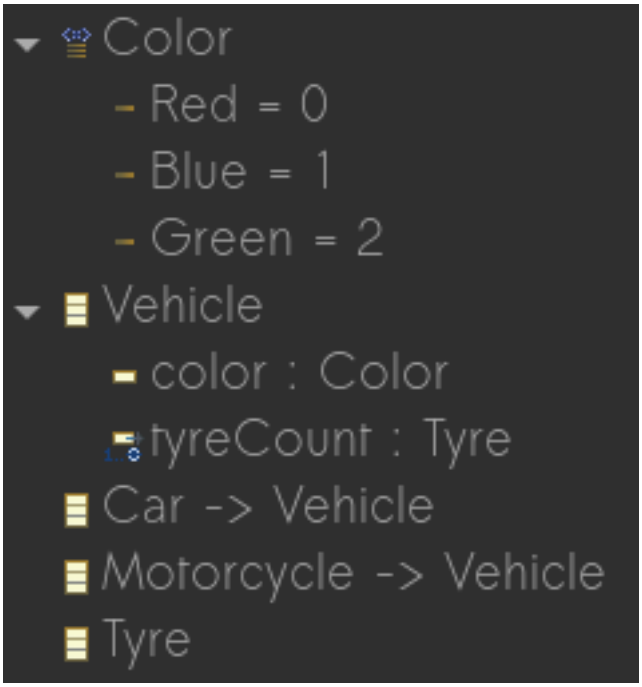

Figure 2.4.: Vehicle Model Example in Ecore

For the *tyreCount* attribute the multiplicity was set. This way Ecore can validate the amount of occurrences relationships between different objects. In this case, a *Vehicle* needs to have at least one tyre, but the maximum is not set. This could be done by moving the *tyreCount* to *Car* and *Motorcycle* and setting the maximum of the *Car* to four and for the *Motorcycle* to two. Ecore supports constraints that are coded in Java, if other, more complicated, constraints need to be added to the metamodel [25]. The biggest advantage of EMF is the fact, that all of the constraints defined in the Ecore metamodel of the modelling language can be validated using EMF validation. Therefore, the validation does not need to be implemented. Additionally, a EMF model can be altered using Java source code and can be represented using XML schema [26]. To summarize EMF offers a solution, that can alter, store and validate created models, based on an Ecore metamodel. This can be leveraged for the model validation framework, as all the needed functionalities on the model side are represented in EMF. Additionally, the EMF.cloud project already contains solutions for some of these components to run in the cloud. Therefore, a model validation framework can be created using EMF without the need of re-implementing already existing solutions.

3. Requirements and Challenges for a Model Validation Framework

This chapter will focus on defining the requirements and challenges of creating a model validation framework for graphical cloud-based editors.

3.1. Requirements for Model Validation

First, the functional and non-functional requirements for the model validation framework will be defined.

3.1.1. Functional Requirements

Validation of Models

First and foremost, the model validation framework needs to deliver functionalities to validate a model, either automatically or on a user request. For this to work, the framework needs to know about the model as well as the constraints that are relevant to the model. As discussed before, the definition of the constraints may be different depending on the constraint language. Some languages store the constraints within the model, while others store the constraints separately. Additionally, the framework needs to be able to validate the model against the rules that are defined. Last but not least, the model framework needs a component, that returns feedback to the user if an error is detected.

Input Validation

To prevent erroneous information to be entered into the system in the first place, the model validation framework should offer a way to validate the inputs right when they are entered. This way errors can be prevented rather than detected.

Custom Constraints

The last functional requirement the model validation framework should offer, is the handling of user-defined constraints. The user should be able to filter certain constraints from the validation result, to enable user freedom. Additionally, a way to add own constraints should be added. This can either be achieved by offering a way to turn pre-defined constraints on or off or by letting the user code new constraints himself and adding them to the validated rule-set.

3.1.2. Non-Functional Requirements

Client-Server Architecture

The first non-functional requirement results from the cloud-based approached. Because of the nature of cloud software the model validation needs to offer strict rules on what will be computed on the client vs. the server. For example, like discussed in Section 3.2.1, the type of the operation plays a role in where the operation should be computed. Another aspect, that the model validation needs to handle, is the communication between the client and the server. Since the validation might be computed on the server, but the result of the validation should be displayed on the client, there needs to be communication between the client and the server, and vice versa. This communication also needs to define a format of requests and responses. These need to be tailored to the information that is needed on both sides in order to fulfill their assigned tasks. If the provided information is not enough the task cannot be fulfilled, and if there is too much information, the deserialization is getting more complicated and computation might take longer.

Model validation Framework

If the application is using a Language Server Protocol (LSP)-like server, that contains only information about the language, without containing any information about the implementation of the specific editor of the language. The server should be designed highly reusable [27]. This allows other editors to use the same server, by providing the language they want to use. In that case, and even in normal Client-Server application, it makes sense to describe a generic model validation framework, that can read out the validation results from the server and provide them to the different clients. Furthermore, such a framework can offer methods to trigger the different endpoints on the server.

Server endpoints

Of course, these endpoints need to be defined by the model validation as well. Endpoints are entry points to the server that let the client trigger actions on the server. So there might be a endpoint for validating the model and another one for retrieving a list of constraints for the input validation. The list of endpoints is heavily influenced by the capabilities of the server and the actions that a client should be able to trigger. Furthermore, the endpoints should be designed in a way, that if they are used for another client, the responses are still applicable.

Generic Server Responses

The overall design goal for the server, more on that in the next chapter, should be to make it as generic as possible without having any information about the client implementation. This allows to decouple [27] the server from one or many clients. If the server calculates a generic response, and this response is then read out by the generic model validation framework. It gives the application the chance to use the result like it intends to, for example for error

marker creation. Or another application might want to show the error message in a tree and not in a diagram. If the server would already return the error message rendered in a specific format, the server would need to be adjusted if another application wants to display the feedback differently. But when the server response is generic, the application can decide how to use the validation feedback, without changing the server.

Usability

Last but not least, usability should be an important requirement for requirement for a model validation framework. The most consistent validation framework is not useful, if the results are not presented in the right place or even at the wrong time. However, the placement of the results is not the only consideration here. It is also important, that the user can understand the error messages and the process for triggering the validation is useable. Nielsens usability heuristics [28] can be used to define the most important rules, with regard to usability, for a model validation framework. Nielsens 10 usability heuristics are [28]:

1. Visibility of system status
2. Match between system and the real world
3. User control and freedom
4. Consistency and standards
5. Error Prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalist design
9. Help users recognize, diagnose, and recover from errors
10. Help and documentation

These heuristics will play a role in Section 3.2.1. Since a model validation framework is not a standalone application but will rather extend an existing graphical model editor, the framework as well as the application will require functionalities to confirm to a certain heuristic. Since this thesis is about the framework it will focus on the functionalities of the framework. Therefore, only the heuristics that are more reliant on the framework side will be discussed, as the others are mainly just good usability design in general. The rules that are more reliant on the application itself, and therefore are not discussed, are visibility of system status, match between system and real world, consistency and standards, flexibility and efficiency of use and help and documentation. The first heuristic to be discussed is user control and freedom. This will be discussed more in detail in Section 3.2.1. Sometimes the user needs to be able to enter wrongful information, because otherwise they would not be able to

create a valid element in the first place. For example, if a e-mail field is filled out and from the first character on the e-mail format is checked, the input will not be valid. This means, that the validation framework should not prevent the entering of new characters if it is not creating a valid e-mail. It should only validate and prevent wrongful information, once the user finished typing. However according to heuristic number 5 errors should be prevented. So the model validation needs to define what errors are prevented, and which are validated and then are shown as errors. In principle, all the errors that can be prevented should be prevented. To apply with heuristic number 4 consistency and standards, the validation framework should be consistent throughout all parts and cohere with connected standards. Some examples for this would be the usage of similar error messages for similar errors or the correct designation of warnings and errors according to the language specification. Aesthetic and minimalist design might be more interesting for the modelling application, since the presentation for the errors should be handled by the application, however the error messages should also follow these principles. Meaning, that they should not contain to much information to confuse the user. Also the aesthetic part might be suitable for the format of the message. Last but not least, the model validation framework should "help users recognize, diagnose and recover from errors". This should not be a surprise because this is the main task of the framework. However this is implying some advanced features, like quick-fixes to let the user fix errors more easily.

3.2. Challenges of Model Validation

3.2.1. Categorization of Constraints

The biggest challenge of a model validation framework is the handling of different types of constraints. Therefore, the following will look into what kind of constraints exist an how those can be categorized into dimensions. This will allow a pretty good overview over different types of constraints and what the requirements are to validate such constraints. What can be a challenge here is, that some dimensions might influence the decision on other dimensions.

Time of validation

The first dimension to be named here is the time the validation is happening. To be more precise, whether or not the validation is happening during run-time (live) or at a specific point (batch), for example when saving [25]. Batch validation can be further described by the state of when the validation is triggered. This thesis will name the time of validation accordingly to the state, where the validation is triggered. The first time at which the model can be validated is pre-enter, meaning that the user is prevented from entering wrong information. For example, if a user needs to edit a field that should only contain letters, a pre-enter validation, will not let the user enter anything except for letters. For example, numbers will be blocked. The goal of this is to never let the model enter an erroneous state. While this can be very useful the limits of those approach are, that there may be erroneous states that are required to be reached in order to make the model editor viable. For example, if there

are nodes that require a child to be valid, there needs to be the option to create the node first, without the child, in order to then add the child and reaching the valid state. Another restriction, that needs to be considered, is that a pre-enter validation requires knowledge about the model on the client, as the client handles the inputs. Why this may not be intended, will be discussed further in Section 3.2.1. The next step, where the input of a user may be validated is pre-confirm, meaning the user is able to enter false information like a number for a letter only field, but he is not allowed to confirm it, for example by leaving the text field. The properties of pre-confirm are pretty similar to the ones of pre-enter, just that the enforcement happens at a different time. Instead of prohibiting the wrong input the input cannot be confirmed. This is of course only applicable to inputs that have a confirm mechanism, so this is very useful for text fields, like described before, but may not be as useful for creating model elements, as those may not have a confirm mechanisms. The last time of validation for batch validation is on-save. When validating on-save the user can enter the wrongful information into the system, however if the user tries to save, it will not be possible. This will give the user freedom over the system state without letting him save the erroneous state, because he needs to fix the diagram before saving. Opposed to the previous discussed batch validation, live validation is not happening at a specific point, but rather during run-time. While batch validation can try to filter out as many errors as possible, filtering out all errors will be hard to achieve. This might be due to limitations of the input validation, because the whole model is not available, or due to human failures, like opening a diagram that has wrongful information because it was opened in another editor that does not have input validation.

Scope of the constraint

The next dimension is the scope of a specific constraint or in other words, what is needed to validate the constraint. So what is the context the validator needs to know about. In the simplest case the constraint can be validated by only looking at an attribute of an object. For example, if the constraint is, that the name of the validated object can not have more than n characters, the constraint can be validated, only with knowledge about the name attribute of the object. Contrary to that, if the name of the object should not be equal to the name of any other object in the model the validator needs to now about the whole diagram, in order to check all of the names. Some examples of this dimension would be local to an attribute, local to a class or global. Of course this categorization is deeply tied to the specifics of the model language as well as the language of the editor, which uses the model validation, as these two are defining what scopes there are. However a exact definition for each constraint is not necessary, it is just important, to be aware of the scope as this will have implications of the decision the developer will make in regard to the other dimensions. For example, a constraint that requires global knowledge about the model, will not be able to be validated on the client, as the whole model needs to be accessed. However, if only one attribute needs to be checked, like the character limit name example from before, the editor might be able to validate such constraints on the client. Another thing to note here is that other than the previously talked about time of validation, the scope of the constraint is not a design decision of the developer, but is given by the specifics of the model and development language.

Feedback to the User

Another dimension is the way the feedback is presented to the user. In Section 3.2.1 the decision, between blocking the input or validating the input after entering it, was discussed. This is also applicable here as this decision should take usability into consideration. According to Nielsen [28] a program should prevent errors as much a possible (Heuristic 5), but on the same time should allow for user control and freedom (Heuristic 3). This makes for an interesting decision as, like described before, it could be that some erroneous state needs to be reached in order to guarantee user control and freedom, so that the user can for example create a node, which normally requires sub-nodes, without the sub-node, because he wants to create the node later on. Furthermore, the feedback should contain enough information to let the user understand the error and fix it. Therefore, different constraints may need different ways of displaying the feedback. While it might be sufficient for live validation to just show an error marker, that can show more information once the user hovers it, input validation needs immediate feedback that the user can understand so he can fix the input.

Location

Last but not least is the location of the validation. To be more precise, the decision between validating on the client or the server. Unlike the other dimensions named in this thesis this dimension is not a part of traditional desktop solutions, but rather is the result of the cloud-based approach in this thesis. Before looking at what influences this decision, it should be looked at, what the result of this decision will influence. First, is the concept of time. If a constraint is being validated on the client, it can be validated much faster as no server round trip is required to get the validation result. So the time factor heavily favors a client validation. However there are other factors that need to be considered. One of them is the general rule of a LSP/Graphical Language Server Platform (GLSP): making the client as slim as possible [27]. This allows all of the logic to be on the server in order to make it more maintainable. If something is changing for the language or the GLSP itself, only the server needs to be updated and not the, possible multiple, clients [27]. As validating a model might require knowledge about the whole model, which was discussed in Section 3.2.1, the validation cannot be done fully on the client as this would contradict the rule that was just discussed. However there are definitely some constraints that can be validated on the client. Input validation is a good example for this, as it is mostly local to the field that is edited with the input, and therefore, the input constraints for the specific field may have been loaded from the server at the start of the application and can then be validated on the client. This is, of course, requires for the rule to be fixed and not be dependable on other fields in the model. Therefore, checking whether or not two variables have the same name would be more suited for a server validation as it requires knowledge about the whole mode
To summarize, the model validation framework needs to offer functionalities, which can validate constraints, that fit into the different dimensions of the categorization that was just proposed. For example, the model validation needs to offer input validation as well as live validation, to prevent as much errors as possible, while still maintaining a viable editor.

3.2.2. Persistence

Another challenge of model validation is the persistence of the validation results. To persist the results of the validation it might be enough to just store the last validation result, but the model could be altered outside an editor with model validation enabled. This would lead to a difference between the model that was lastly validated and the newer version of this model. If this would then be opened again in a editor with the model validation, the model validation framework would return a wrongful validation. To prevent this the proposed model validation framework will define, what should happen, if a model is reopened in an editor with the model validation enabled.

4. A Framework for Model Validation in cloud based Tools

In this chapter the architecture for the model validation framework in a cloud based environment will be proposed. After that the implementation of the proposed architecture within EMF will be described.

4.1. Architecture Goals

One of the most significant differences between desktop application and cloud based approaches is the separation of client and server. Therefore, the first decision that needs to be made is how the functionalities are split between client and server. A client-side validation will be faster, than a server-side validation, as not server roundtrip is required. But, like mentioned before, the validation might require full access to the model, which is located on the server, hence requiring a server-side validation. Furthermore, validating on the server enables reusing the validation for other clients. For this decision the attributes of different constraints, like described in Section 3.2.1, can be used. The model validation framework will extend existing tools that offer functionalities that will be used for the framework. These tools will be described and discussed first. Finally, to provide a useable interaction with the framework, the goals for the User Experience (UX) design will be defined in Section 4.1.4.

4.1.1. Architecture

When considering the requirements previously discussed in Chapter 3 the basic architecture in Figure 4.1. The blue components are existing modules and do not need to be implemented by the model validation framework. The orange components are the ones that need to be customized according to the application itself. The features are derived from the modelling language, that the editor is for. These components are not part of the model validation, but they are the ones that are using the model validation. Therefore, they need to be customized to the modelling language as well as the model validation framework. Depending on the availability of an existing solution, they have to be created from scratch or just be adjusted. Last, the green components denote newly developed components, which are integrated into the model validation framework. They might be based on existing solutions, but they need to be implemented in a client server architecture. Of course, it could be argued, that the metamodel needs to be created, but it might as well already exist. Therefore, the metamodel is denoted as existing.
At the core of the architecture is the editor, that shall be extended with model validation

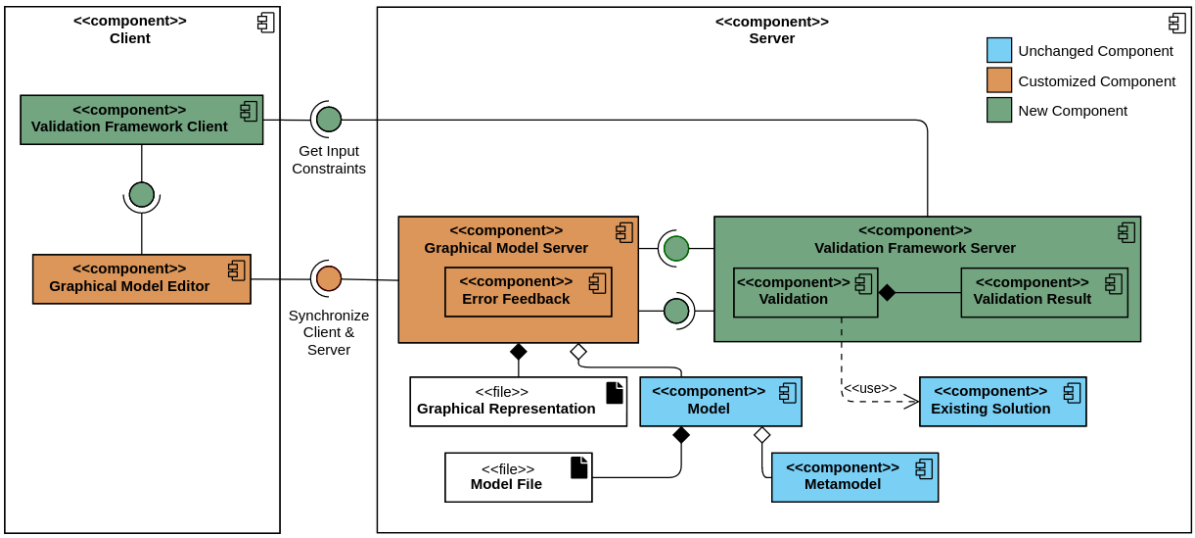

Figure 4.1.: Proposed Architecture

functionalities, for any language. As it is a cloud-based editor, it is communicating with a server, that handles all operations on the created model itself, in order to keep the client as slim as possible [29]. In this architecture the usage of a graphical model server is suggested. This is derived from the concept of Graphical Language Server Protocol, which will be described later on. In short, this allows the server to handle most of the graphical operations, like moving or resizing nodes, on the model as well, making the client even slimmer and managing all the (graphical) logic of the editor on the server. Obviously, a standard client server architecture would also work, however, needs to offer a solution to synchronize the state of the client with the server and vice versa. Either way, all of the logic is handled on the server, therefore it needs to know about the specification of the language that is being modeled in order to correctly handle the requests from the client. This information is all stored in the metamodel of the language. For simplification in the component diagram, the metamodel is part of the model. This is true for some languages like EMF, since the model itself contains a reference to its metamodel [26]. Therefore, this is a legitimate abstraction. However, there are other concepts that do not have the metamodel connected to the model. For example, OCL constraints, which should be a part of the metamodel, are not part of the model itself [14]. The metamodel should also contain all constraints for the specific language. How these constraints are defined is depending on the metamodelling language. Because of this, the server automatically knows about all of the constraints. To validate these constraints, the graphical model server is communicating with the server-side of the validation framework to synchronize the model state between the two. Since the metamodel is referenced in the model itself it is sufficient to just synchronize the model. If the metamodel would not be a part of the model itself, the metamodel would need to be shared between the graphical model server and the validation framework server as well. Furthermore, the graphical model server is able to trigger the validation and receiving the results. The validation framework

servers functionality might be using an existing solution for model validation to avoid the re-implementation of existing features and returns a validation result, which should be generic, so that the validation framework can be used with other editors as well.
The graphical model editor is connected to the validation framework client, which is able to retrieve a list of input constraints, that should be validated before being entered into the system. This ensures proper feedback to the user without the latency caused by a additional server round trip. The graphical model editor can then apply these rules to the input fields it offers to the user.
Additionally, the graphical model server needs a component, that can create graphical elements for the error feedback. When a new validation result is received by the validation framework server, the graphical model server can create the error feedback based on the validation result, which will then be displayed on the client. Now, this architecture can be improved by using some existing technologies to make the architecture more flexible and reusable.

GLSP

The first concept that will be discussed is the Graphical Language Server Platform (GLSP) [30]. GLSP consists of three different components: a diagram client, a protocol and a graphical language server [30]. Like already mentioned the graphical language server enables the server

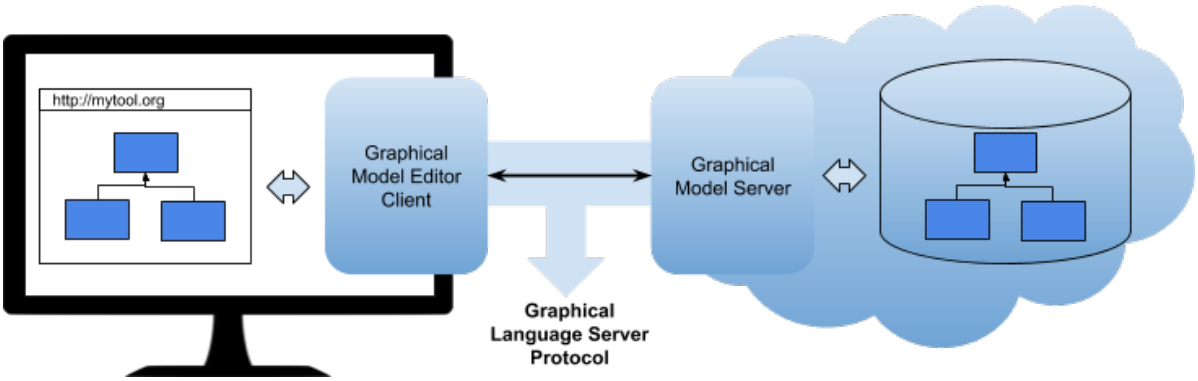

Figure 4.2.: Overview GLSP [30]

to handle the graphical operations of a model. The Graphical Language Server Protocol between the client and the server is derived from the LSP [30]. This allows the client and the server to be decoupled and be developed independently, while reusing existing clients and servers in order to reuse existing solutions. This eases the complexity of development. while offering full flexibility [30]. Lastly, the diagram client is responsible for rendering the diagrams, by using HTML5, Typescript, Sprotty [31] and Scalable Vector Graphics (SVG) [30]. Creating a language-specific editor can be as straightforward as using the generic GLSP server, which is supports basic functionalities, depending on the language [30]. However, features can be added to support advanced language-specific features [30]. Applying GLSP yields considerable advantages to answering research questions in this thesis. First, it offers

a "flexible and extensible" [30] way to create graphical cloud-based modelling editors. This concept does not restrict the technical implementation of both the client and the server. Hence, each application could have a different technical implementation, as long as they both are using the Graphical Language Server Protocol. Furthermore, it shifts the graphical operations to the server, which stores and validates the model. This also shifts the creation of the graphical representation for validation results to the server. Therefore, the validation is running on the server and is only displayed on the client, keeping the client slim and avoid having to code some validation again in JavaScript/TypeScript, or any other language for that matter, on the client [32]. Lastly, GLSP allows for "integrations with EMF and other modelling technologies" [30].

Eclipse Modelling Framework

The advantages of the Eclipse Modelling Framework (EMF) were already discussed in Section 2.2.4. To summarize, EMF offers the meta-metamodelling language Ecore, which enables the creation of metamodels for a modelling language and offers the possibility of adding constraints to the metamodel. Therefore, the features as well as the constraints for the specific language are organized in one place and can be retrieved from the metamodel. Additionally, EMF offers a Java application programming interface (API) for creating and altering EMF models using an EMF metamodel. Furthermore, EMF offers a native way of validating EMF models using EMF validation. Therefore, the validation does not need to be implemented from scratch, as the default EMF validation can be used and, if needed, extended. This, together with the usage if GLSP, offers a flexible and extensible way to create of an graphical editor with model validation enabled.

Eclipse Theia

To minimize the effort of creating a Graphical User Interface (GUI) on the client-side of the modelling editor, the usage of Eclipse Theia [4] is suggested. Eclipse Theia is an IDE framework, that is designed to work in a cloud-based environment. Theia offers basic functionalities and a User Interface (UI) for text editors, as well as modelling editors via sprotty, a tool that helps to create modelling environments for graphical modelling editor [31]. There is a theia-integration [33] from GLSP available, that creates such an environment inside Theia, to work with GLSP. Using this extension offers a wide range of of functionalities without the need of implementing them, for example, a file explorer, a problems view, quick-access bar for commands, etc [34]. Furthermore, these functionalities can be changed or extended if needed.

The updated architecture, when using GLSP, EMF and Theia, is displayed in Figure 4.3. The usage of GLSP changes the graphical model server to a custom GLSP server, since the generic GLSP server only needs to be customized to fit the needs of the modelling language. Also, the protocol between the client and the server, no longer needs to be implemented, because the graphical model server protocol can just be used instead. The implementation effort for the client is reduced as well, as the client is very slim and can be reused for

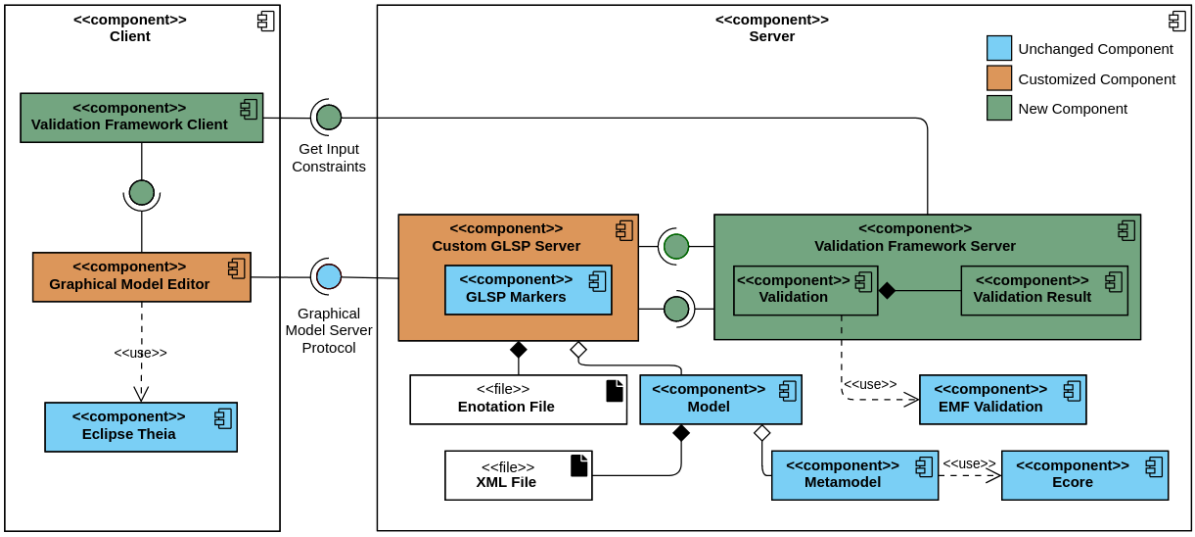

Figure 4.3.: Proposed Architecture with EMF

multiple editors, since the configuration for the language specification is defined on the server. Therefore, the same client is theoretically capable of supporting different languages, since it only contains the graphical representation of the model. The usage of Eclipse Theia also decreases the implementation effort for the client, as it already offers basic functionalities for an editor. Another advantage of the usage of GLSP is, that it already offers the functionality to create markers for errors, meaning the implementation of the error feedback is minimized as well. Additionally, EMF can use EMF validation to validate EMF models, which makes an implementation of the model validation obsolete. Furthermore, XML is used to store the model to a file.

Modelserver

Another component of the EMF.cloud project is the modelserver. [12] The modelserver "manages the runtime of loaded models ("shared editing domain")" [12], this means that the modelserver keeps track of changes on a specific model and can synchronize the state between different clients working on the same model. Changes can be applied on the modelserver with command patterns and clients can subscribe to changes on a specific model. When a model is changed all subscribed clients will be notified and can adapt those changes. This allows for synchronisation between different views of a model. A graphical editor might also have a tree view or a property view in the application in order to offer more precise ways of editing the model [32]. These views can also be synchronized using the modelserver. Integrating the validation to the modelserver would lead to two advantages. First, the validation result would be available for all clients connected to the modelserver and does not need to be calculated for any client (Note: client might be a graphical language server, that is displaying the model). As a result, the validation result can be displayed in all views, that are connected to the modelserver, and also do not calculate the result by themselves.

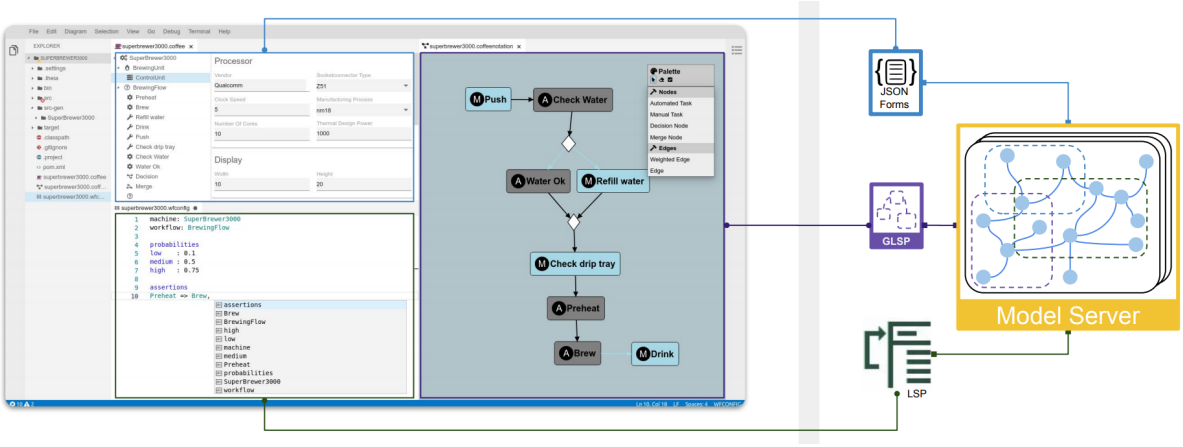

Figure 4.4.: Synchronizing different Views with the Modelserver [32]

Furthermore, it is still possible to alter the way the error is displayed, depending on the view, since the modelserver will only calculate the errors and not handle the graphical elements. This leads to another advantage, since the whole framework is more flexible and extensible, because the graphical representation is decoupled from the model validation. The usage of the modelserver will alter the proposed architecture as Figure 4.5 illustrates. Next to the

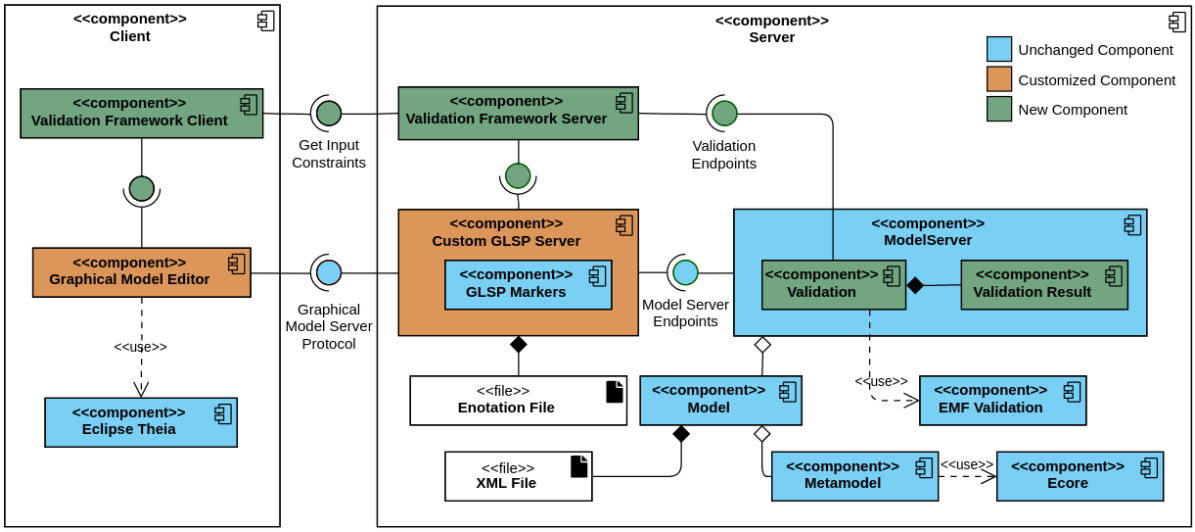

Figure 4.5.: Proposed Architecture with Modelserver

graphical model server the modelserver is introduced, which is used by the graphical model server to subscribe to any client-side changes on the model, and also to send the editing commands, if there is something changed in the editor. The model is now stored on the modelserver and the metamodel is also available on the modelserver, as it is needed for storing the model as well as the validation. The modelserver is extended with validation functionalities. These functionalities might include, but are not limited to, triggering the

validation, notify subscribed clients about changes on the validation result, triggering the validation automatically when the model changes or retrieving a list of constraints defined in the metamodel. To ease the usage of those functionalities, which are offered via endpoints, the model validation framework should offer an interface that can trigger the endpoints on the modelserver as well as retrieve the information. This interface can then be used by the validation implementation of the graphical model server. This implementation should integrate the functionalities of the interface to map the results of the validation to a graphical representation of the detected errors, which is then displayed on the client.

4.1.2. Constraint Validation Stategy

Another goal of this thesis is the coverage of different kinds of constraints. The general distinction here is between language specific and user-defined constraints.

Language-specific

Since language-specific constraints are part of the language itself, and not depended on the user, they can just be defined in the metamodel beforehand. Ecore offers a lot of tools, to add constraints to the metamodel, therefore it should be possible to add all of the constraints a language to the metamodel. If the constraint can not be enforced using the default mechanics, Ecore still allows for Java-coded constraints, which should be sufficient for any constraint that shall be coded.

User-defined

Handling user-defined constraint is more complicated. These constraints are defined at runtime and therefore need to be added during runtime to the metamodel. While it is certainly possible to alter the metamodel during runtime, this is out of the scope for this thesis, and will be discussed in Chapter 6. Another way of adding custom constraints, would be to offer a option to toggle specific constraints on or off. This way the user can decide, which of the rules they want to be enforced. using this approach the set of constraints might be customizable, but need to be added to the metamodel before runtime. Therefore, not giving the user the option to add new constraints, that are not part of the metamodel.

Filtering Constraints

To enable this the framework needs to be able to filter different constraints. Regardless of the usage for custom constraints, this is a feature that should be part of the validation. There are several use cases, where turning of specific constraints makes sense, for example turning of warnings for better visibility.

4.1.3. Persistence of Error Feedback

Another important design decision is if and, if yes, how error feedback should be persisted. In general, it would be advantageous to persist the feedback, in order to simply generate them, when the model is loaded, without having to validate the model again. However, this can lead to problems, if for example, the user alters a model in another editor, or changes the text in the model file. Then the persisted validation result might not be relevant anymore, or even faulty, for example by pointing at a none existing element. To avoid this the model should be validated, when it is loaded, before displaying any error feedback. But since the model is stored on the modelserver the user cannot alter the diagram without triggering a change on the modelserver. Therefore, the modelserver can persist the error feedback (not the graphical representation) and send it to the GLSP server, when the model is loaded up. If the modelserver detects any changes it will validate again and send the new result instead. Therefore, the result can be persisted and the graphical representation created accordingly, which should give the user the feeling of persistence.

4.1.4. Error Feedback

Like described before, there are different categories of constraints. Those constraints are different in the way they should be handled and how the feedback is presented to the user. This section will describe how the feedback for specific constraints is chosen and implemented. For the input validation the user will receive error messages, if the input is not valid. However, the user will not be restricted what he can enter into the input field. The input will be validated live, however the user cannot leave the input field, if the input is erroneous. The input validation is therefore pre-confirm, as the user can enter everything, but cannot confirm erroneous inputs. On top of that, the user will always be notified, which constraint they are violating. This will make sure, that the model stays valid, while not restricting the user and giving the user a learning effect, about what is possible and what not [28].
For the live validation, the feedback shall be presented using error markers, that can be hovered to show the error message. Furthermore, the error messages shall also be displayed in the problems view of the editor, to give a good overview over the state of the model. Both the problems view and the diagram, with the error markers, should be synchronized at all times. Additionally, clicking on an error in the problems view should open the same error in the diagram view.

4.2. Implementation of the Validation Framework

After discussing the architecture of the proposed solution, the following section will focus on implementing the proposed solution. Therefore, the generic part of the model validation will be implemented first, before building a example editor, as the proof of concept.

4.2.1. Modelserver Validation

Since the model validation will run on the modelserver, the first step is to validate a model on the modelserver. For that an *INSTANCE* of the class *Diagnostician* [35] can be used.

```
public BasicDiagnostic validate(final String modeluri) {
  if (!getModel(modeluri).isEmpty()) {
    BasicDiagnostic diagnostics = Diagnostician.INSTANCE
        .createDefaultDiagnostic(getModel(modeluri).get());
    Diagnostician.INSTANCE.validate( getModel(modeluri).get(), diagnostics,
        Diagnostician.INSTANCE.createDefaultContext());
    this.validationResult = diagnostics;
    return diagnostics;
  }
  return null;
```

The method *validate()* gets the modeluri, of the model that should be validated, and returns a *BasicDiagnostic* [36], if the modeluri is existing on the modelserver, which is checked by calling *getModel(modeluri).isEmpty()*.

BasicDiagnostic

A *BasicDiagnostic* contains a *message*, a *severity*, a *source*, a *code*, a *data* field and a list of *children*. The *message* is the error message, that the validator created for the *BasicDiagnostic*, and the *severity* contains an integer that categorises the feedback as an error (4), a warning (2), info(1), ok(0) or cancelled (8) [37]. *Source* and *Code* describe the constraint that is being validated. *Source* defines the metamodel, that the constraint is a part of, while *code* defines which specific constraint of a metamodel the current constraint is. The *data* field contains information about the objects in the model, that are relevant to the current *BasicDiagnostic*. Usually the first element in the field describes the element that has the error, while the second element describes the feature, that this *BasicDiagnostic* is referring to. So if the *BasicDiagnostic* is in regard to the feature rooms missing in a house, then the data field would reference the house element at the zero index of the *data* field, and the rooms feature at the second index. This will then let the GLSP server, retrieve the id of the element, that the error is referring to. Therefore, error feedback can be created for that element. Lastly, the *children* list of a *BasicDiagnostic* contains all of the child-*BasicDiagnostics* a *BasicDiagnostic* might have. Since the code above validates the whole model the *BasicDiagnostic* that is returned also applies to the whole model.
EMF validation always validates all the children of the *EObject* that is validated as well, since the children might contain errors, while the root *EObject* does not. However, this should still return an error for the root *EObject*, since an error in its features is also and error in itself. Therefore, the *BasicDiagnostic* of an *EObject* also contains all of the feedback the children have in the *children* list. Therefore, it is sufficient to validate the root of the model as this will result in the correct feedback for all of the elements. Additionally, EMF validation adjusts

the severity of a *BasicDiagnostic* to be the maximum of its own severity and the severity of its *children* [36]. Therefore, the *BasicDiagnostic* contains everything that is needed, to display the results of the validation.

Diagnostician

To get the *BasicDiagnostic* the model first needs to be validated. For this *Diagnostician* [35] can be used. *Diagnostician* offers the method *validate()*, which can be used to validate a specific *EObject* in this case the *EPackage*, or root, of the model with the given modeluri. Next to the *EObject* a context for caching the information as well as *DiagnosticChain* to accumulate the created *Diagnostics* [35]. For both of them the default ones were chosen, as they were sufficient enough. If the model exists, the above code then creates the corresponding *BasicDiagnostic*. Therefore, the validation does not need to be reimplemented, while still offering flexibility by adjusting the way the diagnostics behave.

Endpoints

Now that the modelserver is able to validate models, it needs to offer some endpoints to trigger the validation and to retrieve the results. Since the modelserver should be able to support live validation and batch validation, there needs to be functionalities for both of the use cases. For the batch validation it is sufficient to offer a single endpoint, that triggers the validation and returns the result of this validation.
For the live validation the validation needs to be automatically triggered to then notify the application about changes in the validation result. The modelserver has a update method that is called every time a specific model is updated. This method was altered, so that an update to the model also triggers the validation for the updated model. After the model was updated and re validated a *modelChanged()* method is called, that is sending the model update to all of the subscribed applications. The model validation framework extended this to also broadcast the validation result to all of the applications that subscribed to the validation result. Therefore, an endpoint to subscribe to these changes is necessary as well. To avoid unnecessary notifications, the validation result is only broadcasted if it is different from the last send validation result. If the application does not want to receive updates anymore, for example when closed, or when validating manually, it can unsubscribe from the subscription listener via another endpoint. Another feature that is supported is input validation. Since the inputs should be validated before being entered into the model, the modelserver cannot handle this task, as it only knows about the model and not the information that is about to be entered into the system. Therefore, an endpoint needs to be added, that lets the application retrieve all constraints that are relevant to the input validation, to then enforce them during the input. This list can be retrieved at startup and just be stored in the application to avoid calling the modelserver every time some input needs to be validated.

```
public void getConstraints(final Context ctx, final String modeluri) {
   Map<Integer, Map<Integer, Map<String, Object>>> jsonResult = new
       HashMap<>();
```

```
ObjectMapper mapper = EMFModule.setupDefaultMapper();
Optional<EObject> eObject = this.modelRepository.getModel(modeluri);
if (eObject.isPresent()) {
   EPackage ePackage = eObject.get().eClass().getEPackage();
   for (EClassifier e : ePackage.getEClassifiers()) {
      if (e instanceof EClass) {
         // Map Feature -> ExtendedMetaData
         Map<Integer, Map<String, Object>> featureMap = new HashMap();
         for (EStructuralFeature esf : ((EClass)
             e).getEStructuralFeatures()) {
            if (esf instanceof EAttribute) {
               EDataType dataType = ((EAttribute) esf).getEAttributeType();
               // Map facet -> Value
               Map<String, Object> facetMap = new HashMap();

               int whiteSpace =
                   ExtendedMetaData.INSTANCE.getWhiteSpaceFacet(dataType);
               facetMap.put("whiteSpace", whiteSpace);
               List<String> enumeration =
                   ExtendedMetaData.INSTANCE.getEnumerationFacet(dataType);
               facetMap.put("enumeration", enumeration);
               List<String> pattern =
                   ExtendedMetaData.INSTANCE.getPatternFacet(dataType);
               facetMap.put("pattern", pattern);
               int totalDigits =
                   ExtendedMetaData.INSTANCE.getTotalDigitsFacet(dataType);
               facetMap.put("totalDigits", totalDigits);
               int fractionDigits =
                   ExtendedMetaData.INSTANCE.getFractionDigitsFacet(dataType);
               facetMap.put("fractionDigits", fractionDigits);
               int length =
                   ExtendedMetaData.INSTANCE.getLengthFacet(dataType);
               facetMap.put("length", length);
               int minLength =
                   ExtendedMetaData.INSTANCE.getMinLengthFacet(dataType);
               facetMap.put("minLength", minLength);
               int maxLength =
                   ExtendedMetaData.INSTANCE.getMaxLengthFacet(dataType);
               facetMap.put("maxLength", maxLength);
               String minExclusive =
                   ExtendedMetaData.INSTANCE.getMinExclusiveFacet(dataType);
               facetMap.put("minExclusive", minExclusive);
```

```
                String maxExclusive =
                    ExtendedMetaData.INSTANCE.getMaxExclusiveFacet(dataType);
                facetMap.put("maxExclusive", maxExclusive);
                String minInclusive =
                    ExtendedMetaData.INSTANCE.getMinInclusiveFacet(dataType);
                facetMap.put("minInclusive", minInclusive);
                String maxInclusive =
                    ExtendedMetaData.INSTANCE.getMaxInclusiveFacet(dataType);
                facetMap.put("maxInclusive", maxInclusive);

                featureMap.put(esf.getFeatureID(), facetMap);
             }
          }
          // Map Class -> Features
          if (!featureMap.isEmpty()) {
             jsonResult.put(e.getClassifierID(), featureMap);
          }
       }
    }
 }
 ctx.json(JsonResponse.success(mapper.valueToTree(jsonResult)));
}
```

To retrieve the list of input constraints, the facets [24] of the data type of an object are used. A mapping is created, that maps the id of a *classifier* plus the id of the *feature* to the list of the values for all of the facets in EMF. If a facet was not set it is pointing to the default value. This list can then be used in the application to lookup all the facets for a specific object and apply the rules to the input field.

4.2.2. Validation Framework

To ease the integration of the validation functionality of the modelserver a framework should be build, that helps to integrate the model validation. The framework will offer ways to trigger the enpoints of the modelserver and read out the responses. The custom GLSP server will then be able to use the results of the validation framework server to create the error feedback accordingly.

Initialization

To initialize the framework a *ModelServerClient* [38] needs to be specified. This can either be done directly, with the same *ModelServerClient* that is used in the application to update the model on the modelserver or by creating a new *ModelServerClient* with the URL that is specified in the framework, which is the default address for the modelserver.

Reading out Responses

In order to remove the need of handling the JavaScript Object Notation (JSON) responses from the modelserver in the application itself. The responses are read out by the model validation framework and persisted in variables. When application triggers the *validate()* method the validation endpoint on the modelserver is triggered and the JSON result is parsed into a List of *ValidationResult*s. A *ValidationResult* is a defined type, that contains the id of an object as well as the *BasicDiagnostic* for that specific object.

```
package org.eclipse.emfcloud.validation;

import org.eclipse.emf.common.util.BasicDiagnostic;

public class ValidationResult {

    private String identifier;
    private BasicDiagnostic diagnostic;

    public ValidationResult(String identifier, BasicDiagnostic diagnostic){
        this.identifier = identifier;
        this.diagnostic = diagnostic;
    }

    public String getIdentifier(){
        return identifier;
    }

    public BasicDiagnostic getDiagnostic(){
        return diagnostic;
    }

}
```

The same thing is happening when the *getConstraintList()* Method is triggered, however in that case the response for each Object is stored in a *EMFFacetConstaint*, which is then mapped to a *classifier ID* and *feature ID*. A *EMFFacetConstaint* contains the values for all the emf facets, that are set and also contains a method for each facet, to check if the value is the default value. This helps the application to check whether or not a specific value should be validated or not. For clarity only the methods for the *pattern* facet were included, the methods also exists for all of the other facets.

```
package org.eclipse.emfcloud.validation;

import java.util.List;
```

```
public class EMFFacetConstraints {

    private int whiteSpace;
    private List<String> enumeration;
    private List<String> pattern;
    private int totalDigets;
    private int fractionDigets;
    private int length;
    private int minLength;
    private int maxLength;
    private String minExclusive;
    private String maxExclusive;
    private String minInclusive;
    private String maxInclusive;

    public List<String> getPattern() {
        return pattern;
    }

    public void setPattern(List<String> pattern) {
        this.pattern = pattern;
    }

    public boolean isPatternDefault() {
        return pattern.isEmpty();
    }

    public EMFFacetConstraints(int whiteSpace, List<String> enumeration,
        List<String> pattern, int totalDigets,
            int fractionDigets, int length, int minLength, int maxLength,
                String minExclusive, String maxExclusive,
            String minInclusive, String maxInclusive) {
        this.whiteSpace = whiteSpace;
        this.enumeration = enumeration;
        this.pattern = pattern;
        this.totalDigets = totalDigets;
        this.fractionDigets = fractionDigets;
        this.length = length;
        this.minLength = minLength;
        this.maxLength = maxLength;
        this.minExclusive = minExclusive;
```

```
        this.maxExclusive = maxExclusive;
        this.minInclusive = minInclusive;
        this.maxInclusive = maxInclusive;
    }


}
```

Validation Result Subscription Listener

To retrieve the latest validation results and then trigger the creation of the graphical respresentation of the errors accordingly. The validation framework offers a subscription listener on the *recentValidationResult* list. The application can implement this subscription listener and is then always notified when the validation results are changing. The application can then trigger the creation of the error markers based ion the new validation result.

```
package org.eclipse.emfcloud.validation;

import java.util.List;

public abstract class ValidationResultChangeListener {

    public abstract void changed(List<ValidationResult> newResult);

}
```

Filtering Constraints

In order to filter some of the rules from being displayed by the error feedback, the validation framework contains a list of *ValidationFilter*s.

```
package org.eclipse.emfcloud.validation;

public class ValidationFilter {

    private int code;
    private String source;

    public ValidationFilter(int code, String source){
        this.code = code;
        this.source = source;
    }
```

```
    public int getCode(){
        return code;
    }

    public String getSource(){
        return source;
    }

    @Override
    public boolean equals(Object obj) {
        ValidationFilter other = (ValidationFilter) obj;
        if (code == other.code && source.equals(other.source)){
            return true;
        }
        return false;
    }

}
```

ValidationFilter contains a integer for the code and a String for the source of the constraint that should be filtered. With Code being the number, that is assigned to a specific constraint and source being the identifier of the metamodel. Before adding a *BasicDiagnostic* returned from the modelserver to the *recentValidationList*, the diagnostics code and source is compared to the validationFilter. If the code and the source are matching, then the *BasicDiagnostic* is disregarded, as it is not intended to be displayed.

```
ValidationResult validationResult = jsonToValidationResult(
      mapper, responseData);
ValidationFilter filter = new ValidationFilter(
      validationResult.getDiagnostic().getCode(),
      validationResult.getDiagnostic().getSource());
if(!validationFilterList.contains(filter)){
       result.add(validationResult);
}
```

The decision to filter the results on the validation framework was chosen, as this allows for different clients to have different rule-sets enabled, which would not be possible if the filtering was done on the modelserver itself.

Get Constraint List

Last but not least the model validation framework is able to retrieve the list of input constraints from the modelserver and store it in a *Map<Integer, Map<Integer, EMFFacetConstraints»*. This *Map* maps from the metamodel id of the element to another *Map* that maps from the metamodel id of the feature the constraint is used for, to the *EMFFacetConstraint*s of the specific feature of a specific element. The application should trigger the method *getConstraintList()* when the model editor is launched. as the *Map* is then initialized. The *Map* can then be used to lookup the facets of a feature of a element, by using the elemtent id and feature id.

4.2.3. Relevance of Validation Client

Originally the architecture was designed to split the validation of constraints between the client and the server, based on the characteristics of the different constraints. It was intended, to move the input validation to the client, by retrieving the input constraint list from the model validation server. However GLSP offers a *LabelEditValidator* [39]. to validate inputs on the GLSP server, with the possibility of returning a *ValidationStatus* [40]. This would then, based on the type of the status, automatically trigger user feedback. A *ValidationStatus* can either be ok, a warning or an error. All of the different severitys can display a message to the user above the input field. However, the *LabelEditValidator* also prevents the user from adding an input with an error by not letting the user exit the input field until the error is fixed. Therefore, the application itself can override the default *LabelEditValidator* and decide which inputs are valid and which are not, based on the retrieved constraint list. Since the premises for validating the inputs on the client was that a additional server round trip time, every time a character changes, the above offered solution is only acceptable if the server round trip does not introduce any kind of noticeable lag. If this is the case, the constraint list should be downloaded to the client and be validated there. If there is no additional lag introduced, the client is obsolete for the architecture. This architecture is pictured in Figure 4.6.
To check, whether or not additional latency is introduced the input validation was tested using two different bandwidths and pings. The first test was conducted with a bandwidth restricted to *50 mbit/s* and zero added latency. Measured was the idle time of the client between the keyboard input and the rendering of the error message. The test was conducted for the rendering of a new error label, a change in the label and the removing of the label for each of the different connections, and the average was taken for the specific connection, as the coefficient of variation (CV) of the averages for all connections was smaller than *0.14*, indicating a good representation. The results of these tests can be seen in Figure 4.7. For *50 mbit/s* and zero ping the response took 288 milliseconds (ms) on average. If this is compared to the *0.5 mbit/s* and zero ping connection the difference is smaller than half a ms, while the slower bandwidth connection is actually a little bit faster, which is probably just coincidence and both of the connection deliver the same result, if the test is run enough times. This indicates, that the bandwidth of the connection is not introducing latency to the editor. Both of the connections were also tested with a 1000 ms ping. As expected this increases the idle

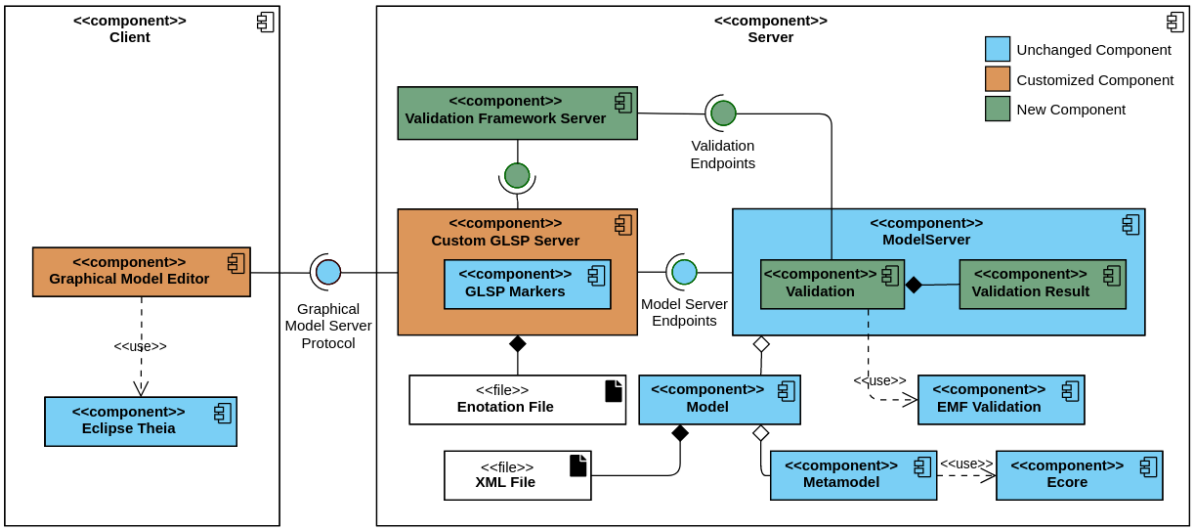

Figure 4.6.: Proposed Architecture without Client

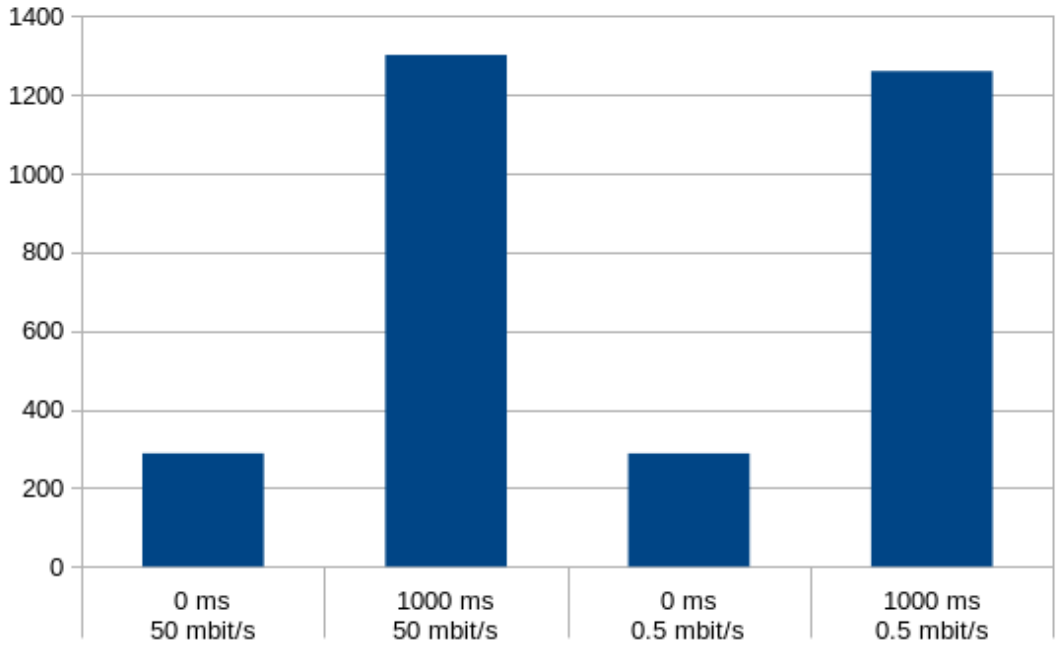

Figure 4.7.: Idle Time for GLSP Calls

time close to 1000 ms. Again, the slower bandwidth, was a little bit faster than the *50 mbit/s* connection, which further solidifies the fact, that the bandwidth does not really matter at reasonable connection speeds (0.5 mbit/s should be achievable in 2020). From this it can be derived, that the bandwidth does not matter to the editor, however the latency to the server does. Since the latency will also introduce this latency to all the other operations in the editor, this is not a problem specific to the input validation, but rather of the whole cloud-based approach. If there are problems with high latency, the technical foundation for cloud-based applications, might not be given.

5. Evaluation

As the model validation framework is an extension to modelling editors, it has two different user groups. For one, the users of the editors that are using the model validation, and on the other hand the developers of such editors. They are also using the model validation framework to improve their editors. Therefore, there are two use cases for the framework. One is the general usage of the features of the model validation, the other is the integration of the framework into a modelling tool. Because of that, this evaluation should focus on the applicability and efficiency of those use cases. To prove those an example editor using the validation framework will be implemented and the advantages and limitations of the integration of the framework will be discussed. Then, a case study will be conducted, where the usability and the accuracy of the model validator in the context of the implemented editor will be evaluated.

5.1. CMMN editor

As a proof of concept for the framework a CMMN [9] editor was build, that uses to model validation framework. Before focusing on the implementation of the editor a metamodel will be created for CMMN, so that all the important features can be derived, before discussing the constraints that will be evaluated in this case study and adding them to the metamodel.

5.1.1. CMMN Constraints

Since the goal of the proof of concept was to ensure that the model validator is working, some constraints that are validated need to be defined. For that some language-specific constraints as well as user-defined constraints were chosen to be validated in the proof of concept.

Language-specific Constraints

First the standards of the CMMN language need to be enforced. For that four errors and three warnings were selected to validate core CMMN functionalities (language-specific constraints). First it should be validated, that all elements are nested, where they are allowed to be created. For example, a *Case* cannot be nested within another *Case* and a *Stage* cannot be nested inside another *Stage*. These rules are categorised as Invalid Nesting constraints. The next category of constraints is Invalid Connection. This contains all the constraints, that describe to which elements sentrys and criterions can be connected to. Furthermore, there are some constraints for the number of sub-elements a specific elements needs to have. For example, a *Case* needs to have exactly one *Casefile* and at least one *Stage*. Last but not least, there are some input

rules for the names of the elements. For example, the name of a *Task* may not exceed 256 characters and the input for the if-part of a *Sentry* needs to be a criterion.
Additionally, three warnings should be validated. The first warning are unused objects. When a element has no edges connected to it it should be marked as unused. Second, any Cyclic Processes in the model, should also be flagged with a warning. Lastly, all objects that reference to a non-existing object should show a warning.

User-defined Constraints

For user-defined constraints, there were two different constraints defined. On one hand, the user should be able to toggle off the object is unused warning, as some element may not need sentrys connected to it and the warning is not wanted. On the other hand, the user should be able to activate a predefined rule. In this case the rule was, that "every case must have at least one exit criterion".

5.1.2. CMMN Metamodel

Since the validation framework was developed to work with the EMF stack, Ecore [23] was chosen as the metamodelling language [26]. Following the CMMN-Documentation [9] and the above described constraints, the following metamodel was created, which pictured in A.1. The metamodel contains all the elements that are part of the PoC. The connections between the elements contain, multiplicities, that represent the minimum and maximum occurrences of that features. So for example, a *Case* contains exactly one *Casefile*. These occurances are validated by the validator. Since Ecore allows for inheritance, the features and attributes of parents are also part of the child. Therefore, a *Stage* contains all of the features and attributes of a *PlanItemDefinition*. The metamodel also contains all of the Java-coded constraints. They are denoted as *Operations* in the metamodel. For example, a *Case* validates the *Operation duplicateStageName()*, which checks whethr or not two *Stages* have the same name. Note, that these constraints need to be coded, after the code is generated from the metamodel. By default, these constraints will not return a result. This metamodel can now be used to implement the editor, and integrating the model validation.

5.1.3. Editor Implementation

The CMMN editor was implemented following the architecture described in Figure 5.1. The architecture was adjusted, so that it now uses a CMMN model and metamodel, for its custom CMMN editor and GLSP server.

Connection to Modelserver and Framework

When a specific model is being loaded the modelserver is the connection to the modelserver as well as the model validation framework server need to be initialized. Therefore, a *ModelServerClient*, for the modelserver connection, and a *CMMNValidationResultChangeListener*, for the model validation framework server, need to be created. The *ModelServerClient* [38] can

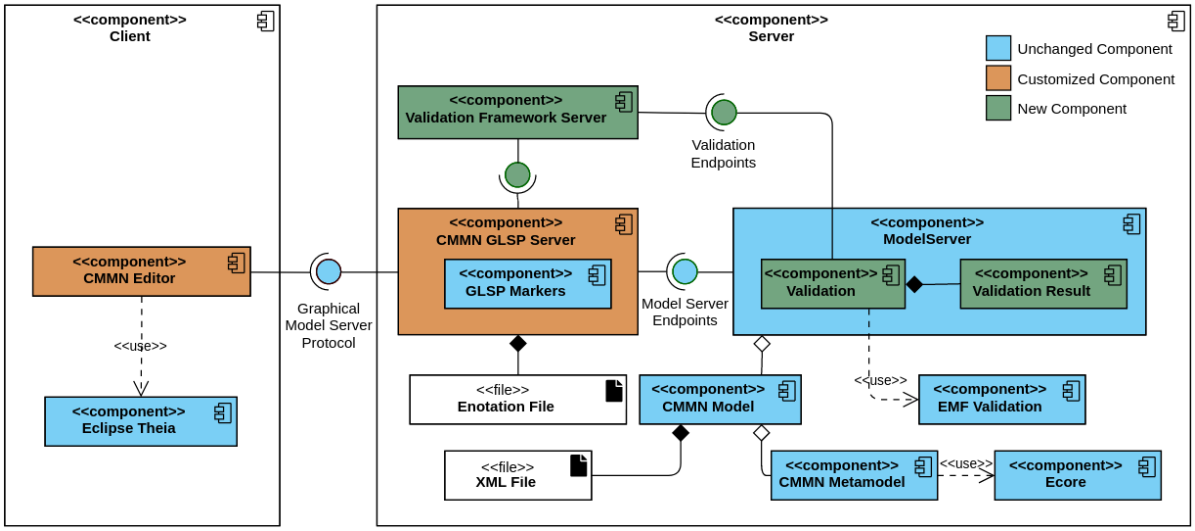

Figure 5.1.: Proposed Architecture with CMMN

be used to trigger the endpoints of the modelserver. The *CMMNValidationResultChangeListener* is used to get updates on the validation results from the model validation framework. To ease the access to both components a *CMMNModelServerAccess* was created. It is initialized with a *ModelServerClient*, *CMMNValidationResultChangeListener* and the uri of the model it is referring to. It contains methods to trigger all of the endpoints for the modelserver as well as the model validation framework. As an example the following are two of those methods, one for receiving the validation result and the other one for initializing the constraint list on the model validation framework.

```
public List<ValidationResult> getValidationResult() throws IOException,
    InterruptedException, ExecutionException {
   return this.validationFramework.recentValidationResult;
}

public void initConstraintList() {
   this.validationFramework.getConstraintList();
}
```

Usage of the Modelserver

```
public CompletableFuture<Response<EObject>> update() {
      EObject root = cmmnFacade.getSemanticResource().getContents().get(0);
      return modelServerClient.update(sourceURI, root, FORMAT_XMI);
}
```

This will trigger the validation every time something is changing on the client, if the model has changed.

ChangeListener

The results of this validation are available via the *CMMNValidationResultChangeListener* that the *CMMNModelServerAccess* was initialized with, as it is set as the ChangeListener in the model validation framework. For this to work the *CMMNValidationResultChangeListener* needs to extend the *ValidationResultChangeListener* defined in the framework and override its *changed()* method.

```
@Inject
private ActionDispatcher actionDispatcher;

@Override
public void changed(List<ValidationResult> newResult) {
   actionDispatcher.dispatch(new ActionMessage(modelState.getClientId(),
      new SetMarkersAction(createMarkers(newResult, modelState))));
}
```

In this case it overrides the *changed()* method to create new markers based on the new result and sending a *SetMarkersAction* [41], which will add the markers, created by the following method to the diagram.

```
public List<Marker> createMarkers(List<ValidationResult> validationResult,
    CMMNModelState cmmnModelState){
   CMMNModelIndex index = cmmnModelState.getIndex();
   List<Marker> markers = new ArrayList();
   for(ValidationResult r: validationResult){
      Optional<GModelElement> gElement =
          index.getGModelElement(r.getIdentifier());
      BasicDiagnostic diagnostic = r.getDiagnostic();
      String message = diagnostic.getMessage();
      if(diagnostic.getSource().equals("org.eclipse.emf.ecore")) {
         message = adjustMessage(diagnostic);
      }
      if(gElement.isPresent()){
         markers.add(new Marker(message, message, gElement.get().getId(),
             getMarkerKind(r.getDiagnostic().getSeverity())));
      }else{
         markers.add(new Marker(message, message,
             cmmnModelState.getRoot().getId(),
             getMarkerKind(r.getDiagnostic().getSeverity())));
      }
```

```
    }
    return markers;
}
```

The method iterates over the list of *ValidationResults*, after it created a new *Marker* list and retrieved the index of the model. Then the graphical model element is retrieved from the index of the model, using the identifier of the *ValidationResult*. The message from the *BasicDiagnostic* is retrieved from the *ValidationResult*. If the *BasicDiagnostic* is reffering to a standard Ecore error (indicated by the source *org.eclipse.emf.ecore*), the message is adjusted in the method *adjustMessage()* to better fit the application. After that, the *Marker* is added to the return list. It will contain the *Id* of the element it is referring to, if the element is available in the index. If not there is most likely a problem with the model state. To avoid any exception, the *Marker* is then created on the root of the diagram. In Figure 5.2 a error marker is pictured on a stage, that does not contain any tasks.

Figure 5.2.: GLSP Makers on an empty Stage

Input Validation

For the input validation the *LabelEditValidator* [39] is extended by a *CMMNLabelEditValidator*, which overrides its *validate()* method. This method is called, when the text of an input field is altered and returns a *ValidationStatus* that can returns a severity, which defines the correctness of the input and an optional message, that is displayed to the user. At the beginning, the type of the element needs to defined, before checking which feature of the element is being altered.

```
if(eObject instanceof Case) {
    elementId = CMMNPackage.CASE;
    if(type.equals(Types.LABEL_NAME)) {
        featureId = CMMNPackage.CASE__NAME;
    }
}
```

In this case the *elementId* is set to the id of a *Case* inside the metamodel if the element is

indeed a *Case*. When it is a case the feature that is altered needs to be defined. If the label is the name label, the *featureId* is set to the id of the feature name in the metamodel. This needs to be done for all the elements and label that are available for them. After all possible elements are traversed the *elementId* and *featureId* need to be checked, wheter or not they are differnt from the stadnard value (-1). If they are not the feature or the element, this field wants to alter is not supported by the input validation. If the ids are not the standard value the *EMFFacetConstraints* of the feature can be loaded from the model validation framework via the *CMMNModelServerAccess*. After that all of the facets may be applied to the input field, if they are not the default value of the specific facet.

```
if(!constraints.isMinLengthDefault()) {
   if(label.length()<constraints.getMinLength()) {
      return ValidationStatus.error("Name must be longer than "+
          constraints.getMinLength()+" digits");
   }
}
```

This would basically be enough to show a generic error message, like the minimum number of characters needed for a specific field. However, if this is done, for example, for the pattern facet. The generic message can only show the pattern itself as a correct solution, which might not be readable to the user. Because of that the type of the element can once again be checked to give a more detailed explanaition of the error, basically making the error message more readable.

```
if(!constraints.isPatternDefault()) {
   if(type.equals(Types.LABEL_VALUE)) {
      for(String pattern: constraints.getPattern()) {
         if(!Pattern.matches(pattern, label)) {
            return ValidationStatus.error("Must be Valid HTTP
                (http://www.example.org)");
         }
      }
   }else {
      for(String pattern: constraints.getPattern()) {
         if(!Pattern.matches(pattern, label)) {
            if(eObjectOptional.get() instanceof Sentry)
                  return ValidationStatus.error("Must be a Condition
                     (a <,=,> b (+,-,*,/ c)");
            if(eObjectOptional.get() instanceof Case)
                  return ValidationStatus.error("Must consist only of
                     letters, numbers, - and /");
            if(eObjectOptional.get() instanceof Stage)
                  return ValidationStatus.error("Must consist only of
```

```
                letters, numbers, - and /");
            if(eObjectOptional.get() instanceof Task)
                return ValidationStatus.error("Must consist only of
                    letters, numbers, - and /");
            else return ValidationStatus.error("Must fit the following
                expression: "+pattern);
        }
    }
  }
}
```

Apart from that, the type of the label can also be used to distinguish between different errror feedbacks. In the example the *HTTPHookDecorator* has a different label type (Types.LABEL_VALUE) than the other labels, therefore a custom error message can be created, without looking up the type of the element. If not errors were found a ok status is returned at the end. To summarize the developer has many freedoms in how he wants to apply the rules to the input field, without having to crete the feedback himself. He can customize the order in which the constraints are being validated and he can create custom messages for different use cases. He could also extend this behaviour to check which rule of the regular expression the input is violating and giving even more specific feedback about the erroneous input. Additionally, constraints that are not part of the metamodel, for example empty checking, can be added as well. The graphical representation of the input validation is picture in Figure 5.3.

Figure 5.3.: Graphical Feedback of Validation Status

5.2. User Study

To evaluate the usability and the accuracy of the model validator, when integrated in an editor, the main challenge is to ensure that the participants are evaluating the model validator and not the editor itself. While it might be possible to evaluate the impact of the model validator by letting the users evaluate the editor with and without the model validator, this can lead to some problems, with the participants building up knowledge about the editor and then having another level of knowledge once they evaluate the second time.

5.2.1. Test Setup

Participants

The case study was conducted with ten participants, from three different occupations, namely students, research assistants and software engineers, each with different prior knowledge about model validation. Since none of the participants had prior knowledge in CMMN they received a short introduction into the most important features and constraints. The research

Occupation	Participants	Experience with any Modelling Editor	Experience with EMF Validation
Research Assistants	2	0%	0%
Software Engineers	5	80%	80%
Students	3	100%	0%

Figure 5.4.: Experience of Participants

assistants did not have any prior knowledge with modelling editors or EMF validation, which sets up an interesting use case, as they can evaluate the model validator without having any bias about specific missing features. Therefore, their evaluation on usability would ideally reflect most users, that are new to the field of modelling. The students on the other hand have had prior knowledge in modelling editors, due to their studies and therefore have certain expectations about modelling editors, and therefore also for validation in a modelling editor, which can be used to see whether or not the experience of other editors plays a role in the usability of the PoC [28]. Last but not least, most of the software engineers had prior experience with EMF validation. This is useful, since they might expect the editor to work in a similar way and can therefore evaluate the similarities between the model validator and the EMF validation. The biggest problem with the participants that had experience with modelling editors, was to ensure that they evaluate the constraint validator and not the editor. However, during the evaluation and in their answers it seems like they did not have any confusion with this distinction, as it was clearly stated multiple times in the introduction to the evaluation.

Modelling Task

The participants were then asked to model the example diagram, pictured in Figure 5.5 after a given script to evaluate the usability: The script contained intentional errors, to ensure that errors are happening during the evaluation. The participants were told, that there would be errors in the script, but not what was an error and how they should solve it. Therefore, the participants, needed to recognize errors that were made and solve them by reading the error message and interpreting it correctly. This ensures, that the error feedback is sufficient to ensure a valid model at all time. The participants were also confronted with actions that were blocked by the editor, for example the creation of a Stage inside another Stage. This way it can be tested whether or not the participants realized, that the action is not possible. The script

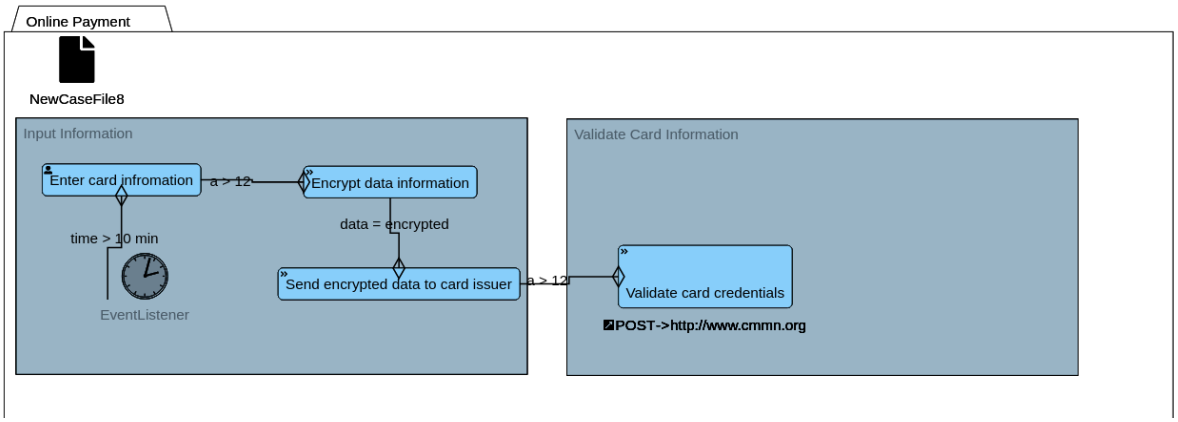

Figure 5.5.: Result of the Evaluation Script

was divided in three different parts after each the participants were asked statements, that regarded the features that were tested in the most recent part. Furthermore, the participants got the chance to give general feedback at the end of each part and were asked if they felt like they could have created an erroneous state. Since the participants are not able to test the accuracy of the validator, the modelling task was observed and all misbehavior's of the validator were noted down.

System Usability Scale

After the modelling task was finished, the participants also were asked some general usability statements, to calculate the usability of the model validator with a standardized scale, to easily compare it with other systems. To evaluate the model validator the System Usability Scale (SUS) was chosen [42]. The System Usability Scale (SUS) offers a free and quick way of measuring the usability of a tool [43]. It also returns a score, which can be compared with other tools, to compare the usability [43].
The standard SUS describes the following ten statements [42]:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.

9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The odd-numbered statements, or statements, are worded positively, while the even-numbered statements are worded negatively [42]. A 5-point scale is used to receive the participants opinion, while 1 means strongly disagree and 5 strongly agree. The answers can then be calculated to a score from 0 to 100, which will be further described in Section 5.2.2. This score allows for an easy comparison between different systems. At first, the standard SUS might seem like it is not a good fit for testing a model validation framework, since it was originally designed for the usability of user interfaces [42]. This may be problematic, as the participants might not evaluate the model validator, but the editor itself, which is not intended. To avoid this confusion the statements were altered in the following way:

1. I think the model validation would improve other editors (e.g. preventing errors, more consistent models).
2. I found the model validation unnecessary complex (e.g. error feedback in strange places).
3. I thought the model validation was easy to use (e.g. errors were visible, quick fixes).
4. I think i would need the support of a technical person to be able to use the model validation (e.g. to read/find the messages).
5. I found that various functions in the model validation were well integrated (live validation, quick fixes, custom constraints).
6. I thought there was too much inconsistency in the model validation. (e.g. contradicting error messages).
7. I would imagine that most people would learn to use the model validation very quickly.
8. I found the model validation very unintuitive to use.
9. I felt very confident using the model validation.
10. I needed to learn a lot of things before I could get going with the model validation.

In general the word *system* was replaced with *model validator* to ensure the participant recognize correct system, that is being tested. Furthermore, examples were added to most of the statements to further describe which aspects should be part of the evaluation. Additionally, two statements were altered to better reflect the model validator. First the statement *I think that I would like to use this system frequently.* was changed to *I think the model validation would improve other editors.*, since the choice of using the model validator is depended on its support in different editors. Therefore, the participant is asked, whether or not he thinks the model validator would improve other editors, since he might have experience with other editors that do not have model validation integrated. Secondly, *I found the system very cumbersome to use.*

was changed to *I found the model validation very unintuitive to use.*, since a main criticism of the SUS is the usage of the word cumbersome. Some non-native english speaker might have problems understanding this word [44]. Therefore, the word unintuitive was chosen, since the goal was to create a intuitive interface with the model validation framework and the fact, that unintuitive closely resembles the german translation, which is advantageous, since most of the participants first language is german.

5.2.2. Results

The results of the described tests will now be evaluated.

Accuracy of the Model Validator

First, the accuracy of the model validator will be analyzed, for this the modeling tasks of all participants were observed. For the tests the live validation was enabled, therefore the state of the validation could be observed at any time. Only two problems were encountered during all evaluations. First there was a problem with the circle detection. If the user created a circle and another edge was added, which was connected to, but not part of the circle, the circle was not detected by the model validator. This happened to all but one of the participants, because they all followed the same script and the script contained this specific use case. Only one participant could get a circle detection warning, since he made a slight mistake during the script and did not add one edge. After investigating the implementation, a fault is discovered in the circle detection, where the circle detection would not be triggered, if another edge, not relevant to the circle was added to a node within the circle. Therefore, the model validator in general is still able to detect syntax errors and violations of CMMN rules, yet the implementation of circle detection warning is not correct.
The other problem that occurred only happened for one user. When trying to come up with a name for the *Event Listener* the participant choose an empty string with a space. This input was accepted by the input validation, but then broke the live validation and the participant had to manually validate after each step or part. This happened because the input validation was not configured to check for this specific case. Normally, the model validation framework should be able to handle the resulting error, however, the error message cannot be parsed on the client side due to a missing handler in the JSON mapper. Once this is fixed the live validation should be able to handle bad inputs, when they are not blocked by the input validation.
To summarize, the model validator always, with the exception of the two mentioned problems, showed the correct state of the model. Therefore, the framework offers a decently accurate validation, provided that all constraints are coded properly.

Usability Feedback

Next, the feedback from the participants regarding usability will be evaluated. First of all, the clarity of error messages will be discussed. There where four statements regarding the

error messages in the evaluation. Two were based on input validation messages and the other two on basic, live validation, error messages. The statements were *It is clear which elements are missing to resolve an error.* (Statement 1), *It is clear what characters are allowed in a specific input field.* (Statement 2), *It is clear how inputs have to be formed.* (Statement 3) and *The error messages were always concise enough to recognize the error.* (Statement 4). The first two statements were answered after the first part of the modelling task, where the focus was basic node creation. The third statement was answered after the second part, which was focused on sentry-creation and the last statement was answered after the modelling task was finished to ensure that all error messages were seen first. Overall, the participants

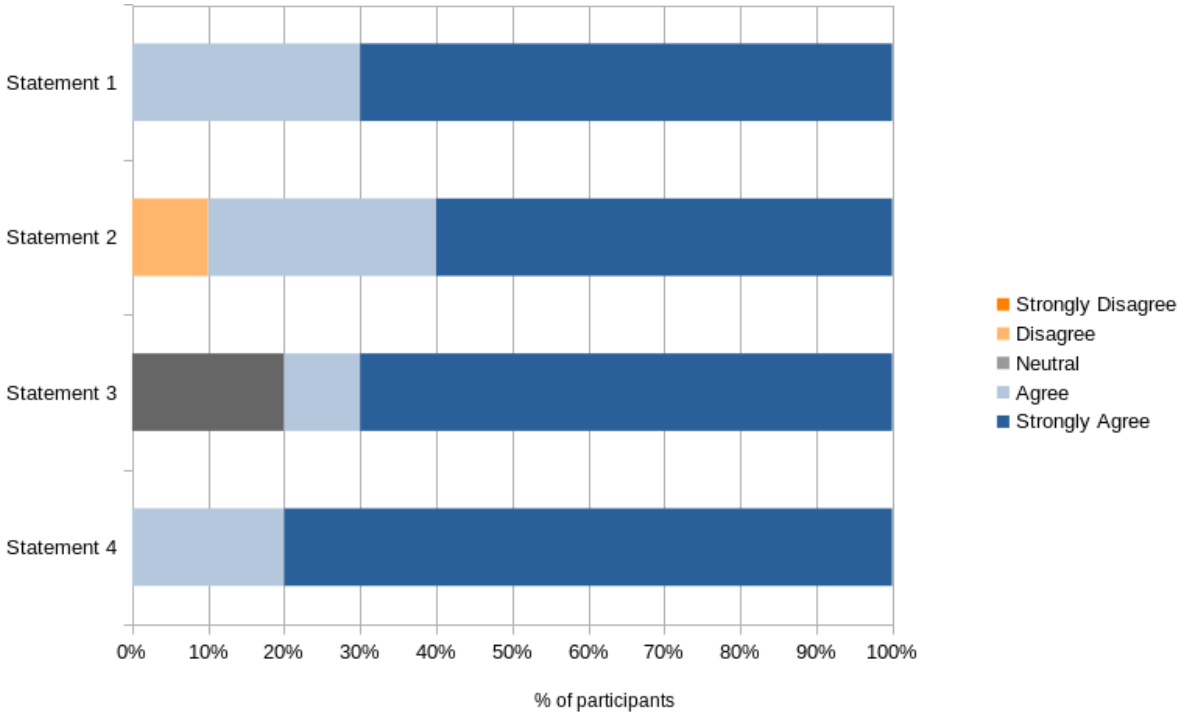

Figure 5.6.: Results of Error Message Statements

did not have any problems solving the errors that occurred during the modelling task, with the given error messages. The average of all statements is greater than 4.0, which means, that the participants are agreeing with the statement of the statement. Also, only 3 answers where neutral or disagreed with the statement, which further strengthens the argument, that the error message were understandable for the participants. Since the editor also restricts some actions, like the invalid nesting of cases within other cases, the users were also asked about the clarity of those restrictions. There were two statements regarding those restrictions. One for the creation of nodes after the first part: *It is clear which elements can be nested within each other.* (Statement 5). And another for the creation of sentrys: *It is clear where sentrys can be created.* (Statement 6). This is arguably the part, where the participants had the most problems. The graphical representation of the restrictions was just not clear enough. Most of

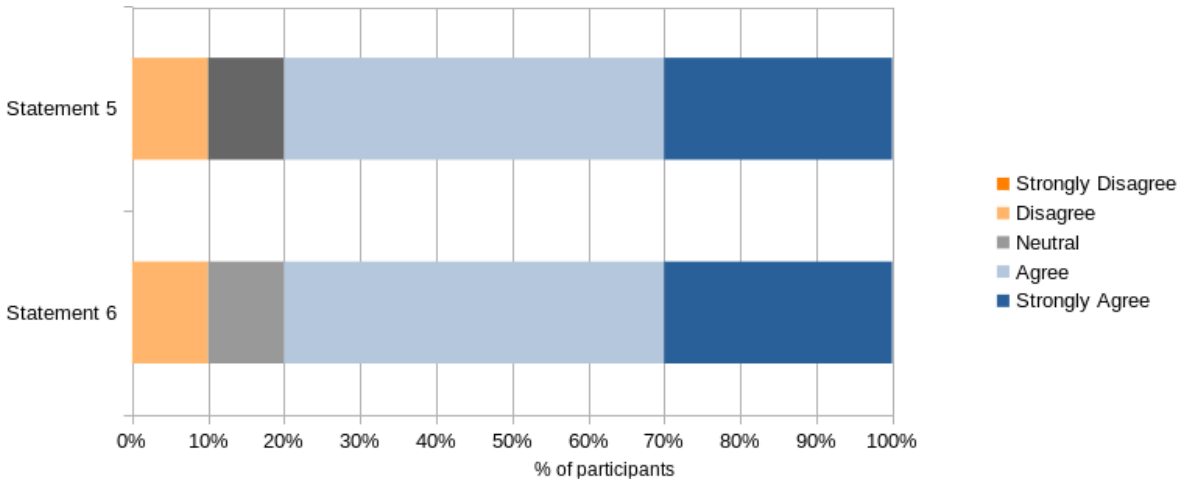

Figure 5.7.: Results of Restriction Statements

the time they had to look twice to spot the graphical feedback. However the answers are still positive, with an average rating of 4, but some improvements may be needed. One participant mentioned, that the object that can receive the selected element as a child, should be marked instead of only receiving the feedback, if the user hovers over the element. Since, the model validation framework does not define the graphical representation, this can easily changed in the editor, or other editors for that matter. Also, adding this functionality to GLSP may be a nice contribution. After the last part, the participants were asked the following statement: *Did the toggling of validation constraints work as expected?*. None of the participants had any problems with this and all answered with yes. Lastly, the users where asked after each part whether or not they felt like they could have created an erroneous state without getting an error message. After the basic node creation part two of the ten participants answered yes. One participant was the instance, where he broke the live validation by entering a space as an empty name like described in Section 5.2.2. The other participant stated, that he could have created a tree structure with the different nodes, with the root being the case. The user noted, that the elements could have been removed from the object they were originally added to, without receiving an error. Additionally, one participant stated that he would try to break the input validation, with "copy&paste, dragging, duplicating and undo/redo", since these features were not part of the CMMN editor. These features certainly need to be tested, when they are integrated, but since the input validation is pre-enter it should be able to pick up these attempts of breaking it, as it doesn't matter were the input is coming from. On top of that, the participants had the chance to give general feedback at the end of every part. One participant stated, that it was hard to realize the severity of an error during the input validation. He stated, that the message could be seen as a warning if the node already has a warning before starting the input validation, because the input validation feedback does not have an additional indication of the severity. An example for this can be seen in Figure 5.8. It was also noted, that an error would be expected, if for example a stage is moved outside of a case, since this looks wrong on the diagram. This can be fixed two ways,

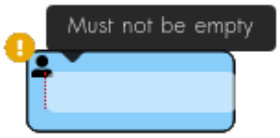

Figure 5.8.: Display of Error Marker and Input Validation

either the editor prevents the movement outside of an allowed element or the editor uses the graphical representation to check if the object is nested in a correct element. Furthermore, it was notes, that the mouse pointers are not consistent for different restrictions and that the input validation should maybe only be triggered once enter is pressed instead of live validation during typing. Again, due to the generic approach of the proposed solution, all of these features can easily be adapted, and in this specific case can be contributions to GLSP to enable this functionality for all editor that use GLSP.

System Usability Scale

After evaluating the answers to the model validator specific statements the answers to the SUS statements shall be evaluated. To do that the SUS offers a uniform way to calculate a score from 0 to 100 from the answers of the users [42]. To calculate the score the statements must first be neutralized since some of them are worded positively and others negatively. To do that the result from positive-worded statements (odd-numbered) needs to be reduced by 1 and the result from negative-worded statements (even-numbered) need to be subtracted from 5 [42]. These new values will then be summed up for all of the statements and the result is multiplied by 2.5 to receive a value from 0 to 100. Figure 5.9 depicts the list of all calculated SUS scores and the number of their occurrence. The mean μ of all the scores is

Score	67.5	85	87.5	90	92.5	97.5	100
N	1	1	1	1	3	2	1

Figure 5.9.: Distribution of SUS Scores

90.25, which would score an A on the SUS scale. To ensure that the mean is representative and the actual scores are close to the mean the standard deviation can be calculated with the following equation:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}} \tag{5.1}$$

With this formula the standard deviation of all the values is *8.76* (rounded to two digits). To analyze, whether or not this standard deviation denotes a high variation in the SUS scores or not the *CV* can be calculated.

$$CV = \frac{\sigma}{\mu} = \frac{8,76}{90.25} \approx 0.1 \tag{5.2}$$

A CV of *0.1* denotes very little variation in collected data, meaning the mean μ of *90.25* is very representable of the overall opinion of the participants.

5.2.3. Benefits

For the User

According to the feedback of the participants the following benefits can be derived for a user, using an editor with the model validation enabled. First of all, it gives the user confidence to create valid models, as almost all of the participants stated, that they did not think they could have created an erroneous state without being notified about the error. Furthermore, the evaluation showed very good usability of the model validator, which can be enhanced by the fact, that the application can further improve the validation techniques to the specification of the editor language. Combining those two advantages gives the user a usable model validation, which eases the model creation, by preventing and detecting errors right away. To further increase the usability quick fixes are offered, which help the user to fix occurring problems. Finally, the framework supports the filtering of errors, which allows the user, if supported by the application, to select the rules they want to validate. This decision can then also be changed during runtime, by toggling specific rules.

For Editor Developers

Apart from the advantages the user gets from the model validation framework, the developer of an editor using the model validation also benefits from the framework. Overall the framework allows for easier integration of model validation into an editor. If the metamodel is created in EMF/Ecore, the main work is to customize the generic GLSP server to use the modelserver, and the model validation framework, which saves a lot of implementation work. Since the constraints and the features of the editor are stored in the metamodel, the developer can just follow the metamodel to derive the needed features, without having to worry about the representation to work with the model validation. Additionally, the framework allows for easy customization of the error messages to improve the usability, since the default messages do not need to be used. Also error messages are easily filterable to create different set of rules for the editor, that can be enabled in different use cases.

5.2.4. Limitations

After discussing the benefits that were discovered during the evaluation, the limitations that were found shall be discussed. Of course, one limitation is the faulty implementation of the circle detection that was discussed before. Additionally, the missing handler, causing the live validation to break, for parsing the input information should be fixed as well. Apart from these technical limitations, the participants of the user study had some useful feedback, which could be adapted to improve the feel of the model validation. First the feedback for the input validation, needs better indication of the severity. Currently, a warning or an information message cannot be distinguished from a error. Furthermore, the fact that no severity is shown

in the feedback, the users often mistakenly assumed that the severity of the node is also the severity of the input validation, which lead to them believing the error was actually a warning, therefore ignoring it. This could be a good contribution to GLSP. Another problem the users encountered was with the quick-fixes. The users stated, that they are hard to find, if someone does not know where to find them and that there is no indication of whether or not a quick-fix is available until the quick-fix menu is opened. This could also be improved in the future and also be a contribution to GLSP.

6. Conclusion and Future Work

In conclusion, the, in this thesis, proposed solution offers a model validation framework, that is usable for the user as well as the developers that may use the model validation in their editors. The usability for the developers is derived from the flexible and extensible way the framework was created. The usage of the Graphical Language Server Protocol offers the flexibility to use any client and server that may already exist. Furthermore, the framework is adaptable to any modelling language without much effort, since neither the validation framework nor the modelserver validation, require any kind of knowledge about the modelling language except for the metamodel. Therefore, designing the metamodel is the main work that needs to be done, in order to use the model validation framework. Despite its easy integration into other editors, the framework supports a variety of features. These include the live and batch validation of models, automatic validation on model changes, the filtering of specific rules and a constraint list, that can be used for implementing input validation for the application. Provided a correct implementation of the validation framework and the constraints in the metamodel, the correctness of these functionalities were proven in this thesis. However, some of these may be improved in the future.
The decision whether or not the model needs to be re-validated could be improved in the future. Currently it is re-validated every time something is changing on the model. Since it is hard to distinguish between actions that should trigger a re-validation and actions that should not trigger a re-validation, this decision could be handled by the application itself, by patching the modelserver with a list of actions that should trigger a re-validation. Furthermore, the way the default Ecore messages are overridden can be improved as well. Currently those messages are adjusted in the application itself, however once the modelserver is moved from a Java application to an eclipse application a fragment can be created, that patches the default messages with a custom file, negating the need to implement the changes in the application.
Apart from improvements for existing features, some new features may also be added to the model validation framework in the future. One example for such a feature would be the possibility of adding Java-coded constraints to the metamodel at runtime. This would allow the application to offer a code editor, which lets the user upload Java constraints to the metamodel, that are then also validated. This would give the user complete control over adding his own custom constraints to the editor, based on the existing metamodel. This approach could also be extended by offering a possibility of customizing the metamodel itself during runtime. Then a application could offer ways to completely change the feature set of a specific language during runtime and therefore letting the user add Ecore specific constraints during runtime without the need of coding them himself.

A. General Addenda

A.1. Model Validation Framework Code

The source code of the model validation framework is available as a Digital Object Identifier (DOI) at [45]

A.2. Modelserver Code

The source code of the modelserver, and the added model validation functionality, is available on the modelserver Github repository [46].

A.3. CMMN Editor Code

The source code for the CMMN editor is available as a DOI at [47].

A.4. Evaluation Results

The results of the evaluation are available as a DOI at [48]

A.5. CMMN Metamodel

A simplification of the CMMN Metamodel is available on the next page.

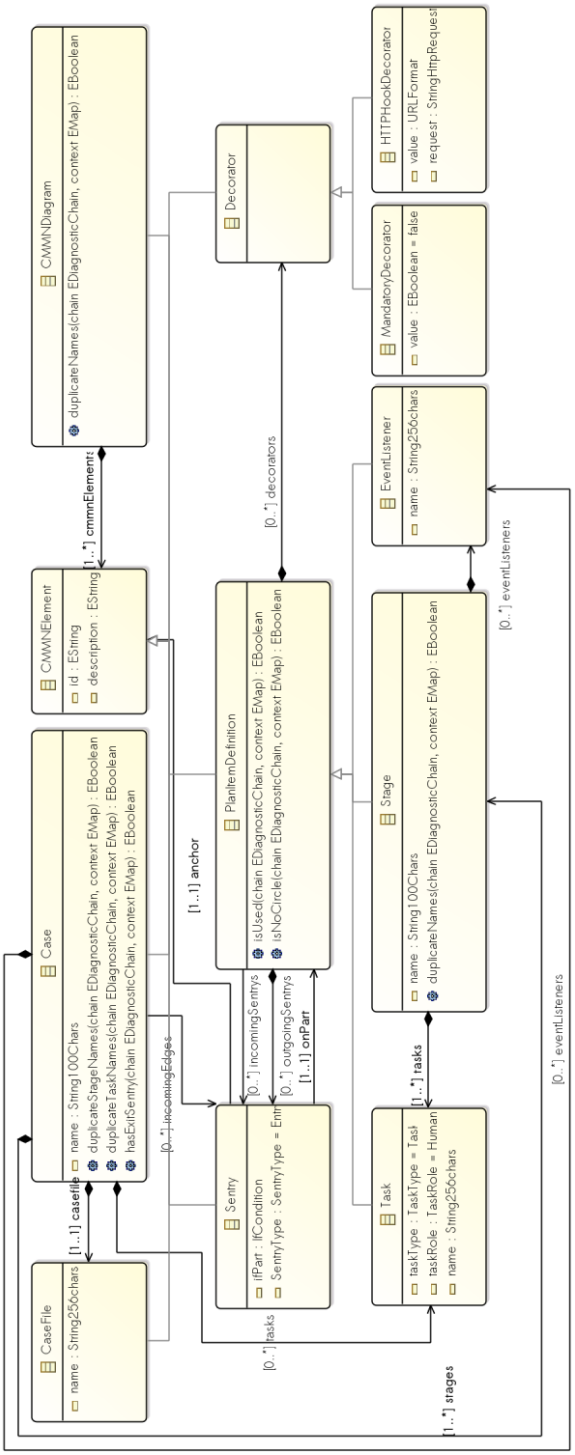

Figure A.1.: CMMN Metamodel for the CMMN Editor

List of Figures

2.1. Simplified Ecore Metamodel [22] 14
2.2. Custom EDataType 15
2.3. Assigning an EEnum as a Type 15
2.4. Vehicle Model Example in Ecore 16

4.1. Proposed Architecture 25
4.2. Overview GLSP [30] 26
4.3. Proposed Architecture with EMF 28
4.4. Synchronizing different Views with the Modelserver [32] 29
4.5. Proposed Architecture with Modelserver 29
4.6. Proposed Architecture without Client 41
4.7. Idle Time for GLSP Calls 41

5.1. Proposed Architecture with CMMN 45
5.2. GLSP Makers on an empty Stage 47
5.3. Graphical Feedback of Validation Status 49
5.4. Experience of Participants 50
5.5. Result of the Evaluation Script 51
5.6. Results of Error Message Statements 54
5.7. Results of Restriction Statements 55
5.8. Display of Error Marker and Input Validation 56
5.9. Distribution of SUS Scores 56

A.1. CMMN Metamodel for the CMMN Editor 61

Acronyms

API application programming interface. 27

BPMN Business Process Model and Notation. 4

CMMN Case Management Model and Notation. iv, 2–4, 43, 44, 50, 55, 60

CV coefficient of variation. 40, 56, 57

DOI Digital Object Identifier. 60

EMF Eclipse Modelling Framework. 4–6, 14, 16, 24, 25, 27, 28, 32, 35, 44, 50, 57

EMOF Essential MOF. 14

GLSP Graphical Language Server Platform. 22, 26–28, 31, 32, 35, 40, 44, 47, 55–58, 62

GUI Graphical User Interface. 27

JSON JavaScript Object Notation. 36, 53

JSR Java specification request. 6, 7

LSP Language Server Protocol. 18, 22, 26

MDE model-driven engineering. 1

MOF Meta Object Facility. 14

ms milliseconds. 40, 42

OCL Object Constraint Language. 5, 6, 25

PoC Proof of Concept. iv, 1, 4, 44, 50

SUS System Usability Scale. 51–53, 56, 62

SVG Scalable Vector Graphics. 26

UI User Interface. 27

UML Unified Modelling Language. 5, 6, 15

UX User Experience. 24

XML eXtensible Markup Language. 11, 14, 16, 28

XSD XML Schema Definition. 11–14

Bibliography

[1] M. Gogolla, F. Hilken, and K.-H. Doan. "Achieving model quality through model validation, verification and exploration". In: *Computer Languages, Systems & Structures* 54 (Dec. 2018), pp. 474–511. ISSN: 14778424. DOI: 10.1016/j.cl.2017.10.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S1477842417300118 (visited on 11/11/2020).

[2] D. Byrne, C. Corrado, and D. E. Sichel. *The rise of cloud computing: minding your P's, Q's and K's*. Tech. rep. National Bureau of Economic Research, 2018.

[3] Microsoft. *Visual Studio Code [Website]*. Apr. 2016. URL: https://code.visualstudio.com/ (visited on 11/11/2020).

[4] TypeFox. *Eclipse Theia [Website]*. URL: https://theia-ide.org/ (visited on 11/11/2020).

[5] M. Miller. "Cloud computing pros and cons for end users". In: *microsoftpartnercommunity.co. uk* (2009).

[6] R. Picek and V. Strahonja. "Model Driven Development-future or failure of software development". In: *IIS*. Vol. 7. Citeseer. 2007, pp. 407–413.

[7] S. Beydeda, M. Book, V. Gruhn, et al. *Model-driven software development*. Vol. 15. Springer, 2005.

[8] E. Seidewitz. "What models mean". In: *IEEE software* 20.5 (2003), pp. 26–32.

[9] Object Management Group. *CMMN Specification [Website]*. URL: https://www.omg.org/spec/CMMN/1.1 (visited on 11/11/2020).

[10] S. A. White. "Introduction to BPMN". In: *Ibm Cooperation* 2.0 (2004).

[11] M. Kurz, W. Schmidt, A. Fleischmann, and M. Lederer. "Leveraging CMMN for ACM: examining the applicability of a new OMG standard for adaptive case management". In: *Proceedings of the 7th International Conference on Subject-Oriented Business Process Management*. 2015, pp. 1–9.

[12] Eclipse Foundation. *EMF.cloud [Website]*. URL: https://www.eclipse.org/emfcloud/ (visited on 11/11/2020).

[13] Object Management Group. *OCL Specification [Website]*. URL: https://www.omg.org/spec/OCL/About-OCL/ (visited on 11/11/2020).

[14] M. Richters. *A precise approach to validating UML models and OCL constraints*. 2001.

[15] A. J. Shidqie, R. Möller, D. Gollmann, and M. S. M. Garcia. "Compilation of OCL into Java for the Eclipse OCL Implementation". PhD thesis. Citeseer, 2007.

[16] E. Bernard and S. Peterson. "JSR 303: Bean validation". In: *Bean Validation Expert Group, March* (2009).

[17] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. "TranStrL: An automatic need-to-translate string locator for software internationalization". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009 IEEE 31st International Conference on Software Engineering. ISSN: 1558-1225. May 2009, pp. 555–558. DOI: 10.1109/ICSE.2009.5070554.

[18] World Wide Web Consortium (W3C. *XML Schema Definition Language (XSD)*. URL: https://www.w3.org/TR/xmlschema11-1/ (visited on 11/11/2020).

[19] D. Lee and W. W. Chu. "Comparative analysis of six XML schema languages". In: *ACM SIGMOD Record* 29.3 (Sept. 1, 2000), pp. 76–87. ISSN: 0163-5808. DOI: 10.1145/362084.362140. URL: http://doi.org/10.1145/362084.362140 (visited on 10/16/2020).

[20] W3schools. *XSD Restrictions/Facets*. URL: https://www.w3schools.com/xml/schema_facets.asp (visited on 11/11/2020).

[21] E. Merks and D. Steinberg. "From Models to Code with the Eclipse Modeling Framework". In: *EclipseCon. Eclipse Foundation* (2005).

[22] M. Elaasar, L. C. Briand, and Y. Labiche. *A Metamodeling Approach to Pattern Specification and Detection*. 2006.

[23] Eclipse Foundation. *Package org.eclipse.emf.ecore [API]*. 2015. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html (visited on 11/11/2020).

[24] Eclipse Foundation. *ExtendedMetaData [Documentation]*. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/util/ExtendedMetaData.html (visited on 11/11/2020).

[25] L. Haibing, Z. Ning, L. Yonglin, L. Xiaobo, and Z. Yifan. "EMF Based Validation Methods of the Static Semantics of Models". In: *2015 2nd International Conference on Information Science and Control Engineering*. 2015 2nd International Conference on Information Science and Control Engineering. Apr. 2015, pp. 207–211. DOI: 10.1109/ICISCE.2015.53.

[26] E. Merks and J. Sugrue. *Essential EMF*. 2009.

[27] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot. "Towards a language server protocol infrastructure for graphical modeling". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2018, pp. 370–380.

[28] J. Nielsen. *Ten usability heuristics*. 2005.

[29] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. "The interactive performance of SLIM: a stateless, thin-client architecture". In: *ACM SIGOPS Operating Systems Review* 33.5 (1999), pp. 32–47.

[30] Eclipse Foundation. *GLSP [Website]*. URL: https://www.eclipse.org/glsp/ (visited on 11/11/2020).

[31] Eclipse Foundation. *Sprotty [Website]*. URL: https://projects.eclipse.org/projects/ecd.sprotty (visited on 11/11/2020).

[32] P. Langer and M. Fleck. "Lifting the greatness of EMF into the cloud with EMF.cloud". Eclipsecon Europe. 2019. URL: https://www.eclipsecon.org/europe2019/sessions/lifting-greatness-emf-cloud-emfcloud (visited on 11/11/2020).

[33] Eclipse Foundation. *glsp-theia-integration [Source Code]*. URL: https://github.com/eclipse-glsp/glsp-theia-integration (visited on 11/11/2020).

[34] J. Helming. "Eclipse Theia vs Che vs VS Code". Eclipsecon. 2020. URL: https://www.eclipsecon.org/2020/sessions/eclipse-theia-vs-che-vs-vs-code (visited on 11/11/2020).

[35] Eclipse Foundation. *Diagnostician [Documentation]*. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/ecore/util/Diagnostician.html (visited on 11/11/2020).

[36] Eclipse Foundation. *BasicDiagnostic [Documentation]*. June 2011. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/common/util/BasicDiagnostic.html (visited on 11/11/2020).

[37] Eclipse Foundation. *Diagnostic Error Severity [Documentation]*. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/constant-values.html (visited on 11/11/2020).

[38] M. Github. *ModelServerClient Github*. URL: https://github.com/eclipse-emfcloud/emfcloud-modelserver/blob/master/bundles/org.eclipse.emfcloud.modelserver.client/src/org/eclipse/emfcloud/modelserver/client/ModelServerClient.java (visited on 11/11/2020).

[39] Eclipse Foundation. *LabelEditValidator [Source Code]*. URL: https://github.com/eclipse-glsp/glsp-server/blob/master/plugins/org.eclipse.glsp.server/src/org/eclipse/glsp/server/features/directediting/LabelEditValidator.java (visited on 11/11/2020).

[40] Eclipse Foundation. *ValidationStatus [Source Code]*. URL: https://github.com/eclipse-glsp/glsp-server/blob/master/plugins/org.eclipse.glsp.server/src/org/eclipse/glsp/server/features/directediting/ValidationStatus.java (visited on 11/11/2020).

[41] Eclipse Foundation. *SetMarkersAction [Source Code]*. URL: https://github.com/eclipse-glsp/glsp-server/blob/master/plugins/org.eclipse.glsp.server/src/org/eclipse/glsp/server/features/validation/SetMarkersAction.java (visited on 11/11/2020).

[42] J. R. Lewis and J. Sauro. "The factor structure of the system usability scale". In: *International conference on human centered design*. Springer. 2009, pp. 94–103.

[43] J. Brooke. "SUS: a "quick and dirty'usability". In: *Usability evaluation in industry* (1996).

[44] K. Finstad. "The system usability scale and non-native English speakers". In: *Journal of usability studies* 1.4 (2006), pp. 185–188.

[45] S. Graband and J. Helming. *Model Validation [Source Code]*. Nov. 2020. DOI: 10.5281/zenodo.4269343. URL: https://doi.org/10.5281/zenodo.4269343.

[46] Eclipse Foundation. *Modelserver [Source Code]*. URL: https://github.com/eclipse-emfcloud/emfcloud-modelserver (visited on 11/11/2020).

[47] S. Graband. *CMMN Editor [Source Code]*. Nov. 2020. DOI: 10.5281/zenodo.4269341. URL: https://doi.org/10.5281/zenodo.4269341.

[48] S. Graband. *Model Validation Framework Evaluation Result*. Zenodo, Nov. 2020. DOI: 10.5281/zenodo.4270214. URL: https://doi.org/10.5281/zenodo.4270214.