DTU Computer Science and Engineering
Department of Computer Science and Mathematics

Creation and Customization of Graphical Notations without Programming: A Universal Diagram Editor

Master's Thesis by Mikail Cengiz

Creation and Customization of Graphical Notations without Programming: A Universal Diagram Editor

Master's Thesis
January, 2025

By
Mikail Cengiz

Cover photo: Vibeke Hempler, 2012

Published by: DTU, Department of Computer Science and Mathematics, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark

www.compute.dtu.dk

Approval

This thesis has been prepared over five months at DTU compute, at the Technical University of Denmark, DTU, in partial fulfilment for the degree M.Sc. in Engineering (Computer Science and Engineering).

Mikail Cengiz - s230362

Mikail Cengiz

Signature

29/12-2024

Date

Abstract

Graphical modeling languages are essential in software engineering, supporting tasks such as system design, stakeholder communication, and documentation. However, configuring current diagram editors to accommodate new or modified graphical notations is complex and requires programming knowledge. These editors rely on code generation, making the process inaccessible to graphical notation developers and costly in terms of time and resources.

This thesis addresses these limitations by developing a universal diagram editor that enables graphical notation developers to define and customize graphical notations independently, without programming. The proposed solution uses Model-Based Systems Engineering (MBSE) principles, where graphical notations are defined by using meta models, mapping models, and graphical representation models. Using a static codebase for the diagram editor with dynamic model-driven configurations, the editor eliminates the need for code generation and compilation.

The solution was evaluated through user testing with both technical and non-technical users. Participants validated the tool's usability and flexibility, highlighting its intuitive interface and the ease of defining and using graphical notations in a diagram editor. Performance tests were also performed, and they showed responsive interactions with the tool.

This thesis highlights key challenges in how diagram editors are designed, including making notation configuration accessible to non-technical users and ensuring real-time diagram editor updates without code generation. The universal diagram editor serves as a user-friendly diagram editor for creating modeling languages, eliminating the dependency on software developers.

The findings contribute to diagram editor development by demonstrating a low-code approach together with MBSE principles to create easily configurable editors. Future research could include improving support for graphical modeling elements' appearances, refining the meta modeling interface for non-technical users, and expanding the constraint validation capabilities. This work establishes a foundation for more accessible and flexible diagram editors.

Acknowledgements

I would like to thank my supervisor Ekkart Kindler for his guidance throughout the whole project.

Contents

1 Introduction

Software engineering and its graphical modeling languages are in a state of continuous evolution. New languages are introduced, while existing ones are updated to address new requirements. These languages improve the clarity of system architecture, design, implementation, and documentation, making them essential for software developers and stakeholders.

Through an analysis of existing tools, it becomes clear that current diagram editors require software developers to have their graphical notations implemented when they are modified, or new notations need to be added. This dependency makes the modification process time-consuming and costly and limits the flexibility of organizations that need to frequently modify or introduce new graphical notations. Assuming that graphical notation developers are more knowledgeable in the business domain when they define a graphical notation on paper, the software developer might not interpret it as the graphical notation developer does. Therefore, the software developer would try to implement the graphical notation based on this paper, but the end result is usually not exactly what the graphical notation developer intended. This leads to another consequence: it is difficult to communicate the details of a graphical notation to a software developer.

We can avoid these challenges by using low-code practices in diagram editors, enabling graphical notation developers to realize graphical notations themselves without programming. But how can we accomplish this? In some current diagram editor tools, Model-Based Systems Engineering (MBSE) has shown that the configuration of graphical notations can be handled conceptually. They do so by allowing users to define meta models of the graphical notations and link each meta model concept to its graphical representation. However, these tools are too technical for graphical notation developers to use. By transforming this concept into a user-friendly application and implementing a low-code approach for configuring graphical notations, we could create greater accessibility for anyone interested in defining a graphical modeling language. This would lead to a universal diagram editor that serves the needs of all users, including developers of graphical notations.

An existing tool that uses MBSE principles to achieve a configurable editor is Eclipse, which utilizes its Eclipse Modeling Framework (EMF). EMF has shown that meta models can define a modeling language and can then generate code from this modeling language. The code it generates can be for Java classes, or, more interestingly, for this thesis, it can also generate code for an editor. However, EMF's editor only displays the tree structure of a model in a text-based way, and it does not allow editors to use graphical elements to move model elements around, thereby changing the structure of a model in a diagram-editor manner. [1] For this, Eclipse has built another layer on top of EMF called the Graphical Modeling Framework (GMF). GMF is a framework that provides generative components that enable the development of diagram-based graphical editors. GMF works by requiring additional information about the defined modeling language, which formulates its graphical features to generate the diagram-based graphical editor. This information is provided manually through multiple models. [2]

EMF and GMF become quite complex to use, not only for graphical notation developers but even for programmers, as there are many different models that one has to keep track of when developing graphical notations. Many things can go wrong; compile errors can occur when configuring these models, and if the graphical modeling language has been configured many times, old code from earlier configurations can cause errors in the newer generated code. The process in which graphical notations are implemented is also very time-consuming and static because when one, for example, has to modify an existing graphical notation element, one first has to edit the meta model, then manually create all the related graphical models, then generate the Java code, and finally run the code, and wait for the editor to compile and load.

Looking into state-of-the-art theories and knowledge in the MBSE field, including the concepts that Eclipse has demonstrated, with their use of models to achieve configurable graphical editors, and studying a variety of graphical modeling languages and diagram editors, this thesis aims to build a solution centered around graphical notation developers, improving their workflow in customizing and creating graphical notations. The thesis has resulted in a diagram editor capable of using configurable meta models to create diagrams. The diagram editor does not need to be programmed at all when a graphical notation is modified or added, nor does it need to have its code regenerated. Its codebase is completely static, and the only underlying structures that are dynamic are its models. The web application is twofold, consisting of a diagram editor (for diagram creators) and a model configurator (for graphical notation developers).

This thesis is divided into several chapters that describe the process of creating the universal diagram editor. In Chapter 2, the conceptual foundations of the thesis are introduced, presenting the key concepts and terminologies used.

In Chapter 3, we explore the core challenges of current diagram editors and formulate the problem statement on which this research is based. The methodology that has guided each phase of this research will also be introduced.

In Chapter 4, we dive into a conceptual analysis, examining the key principles and theories that enable the configurability of graphical notations in diagram editors. We explore relevant concepts within MBSE to inform the solution, establishing the core foundation needed to design the system.

With the theoretical foundation established, Chapter 5 moves into defining specific requirements, informed by the concepts analyzed in the previous chapter. This chapter outlines the functionalities of the solution and highlights the necessary ones to achieve the universal diagram editor.

Building on these requirements, Chapter 6 presents the conceptual design of the solution, detailing how each component will interact within the system. Chapter 7 then translates this conceptual plan into a software design, specifying the architecture for the implementation of the solution.

Chapter 8, the implementation chapter, provides a detailed explanation of the technical

aspects of building the system, discussing the challenges encountered and the solutions applied to ensure that the system meets its goals. The source code for the universal diagram editor developed is publicly available in the following Github repository: https://github.com/mikailcengizz/universal-diagram-editor.

Finally, in Chapter 9, we evaluate the system, assessing both its technical performance and its usability in enabling graphical notation developers to realize graphical notations independently. This chapter reviews to what extent the editor fulfills its purpose and discusses potential future improvements.

Chapter 10 concludes the thesis by reflecting on the contributions made and summarizing the knowledge gained.

2 Conceptual Foundations

This chapter will briefly present the key concepts and terminologies used throughout this thesis. These concepts were derived from existing diagram tools, including Eclipse, MetaEdit+, and Microsoft Visio, which will be analyzed in the next chapter.

2.1 Defining a Graphical Notation

In this paper, the term *graphical notation* is used interchangeably with *graphical modeling language*, as they refer to the same concept. A graphical notation consists of two parts:

- **Meta Model:** Defines the concepts of a modeling language.
- **Graphical Representation:** Defines the visual appearance of the concepts defined in the meta model.

The process of defining a meta model is referred to as *meta modeling*.

2.2 Instance Models

A meta model can be used to create another model called the *instance model*. Instance models can be represented in two types of diagrams: object diagrams (abstract syntax) and graphical diagrams (concrete syntax). In diagram editors, these representations are used in combination to ensure that the structure in a *graphical diagram* is correct based on its underlying *object diagram*.

2.3 Mapping Model

The connection between *abstract syntax* and *concrete syntax* is established through a *mapping model*.

- **Purpose:** A *mapping model* defines how each concept in the *abstract syntax* is mapped to a graphical appearance in the *concrete syntax*.
- **Example:** Consider a *meta model* that defines the UML class diagram language. A ”Class” concept in the meta model might map to a rectangle in the *graphical representation*, while an ”Association” maps to a line.

A single concept and its appearance together are referred to as a *graphical modeling element*.

2.4 Palette and Stencils

In diagram editors, *stencils* represent graphical modeling elements available in the editor’s *palette*. A stencil is linked to a concept in the meta model. A palette is a built-in window of the diagram editor. A palette allows diagram creators to drag stencils (such as classes) onto a canvas.

These concepts provide the foundation for implementing graphical diagram editors. Such editors enable end-users to drag *graphical modeling elements* from the palette onto a canvas, creating *graphical instance models*, where the underlying *object instance model* aligns with the *graphical notation* being used.

3 Problem Statement

Software engineering heavily relies on agile practices, prioritizing iterative updates. However, when examining the process of implementing graphical notation modifications in diagram editors with current tools, it is a complex and inefficient process that can make iterative updates take longer. [3] The process often involves a dependency between graphical notation developers and software developers. Graphical notation developers often cannot realize graphical notation changes themselves. Instead, they must communicate their modifications to a software developer, who then implements the changes in the diagram editor.

This dependency is problematic because software developers often lack deep domain knowledge, making it difficult for graphical notation developers to communicate the modifications to them. The communication can introduce misinterpretations, which can lead to incorrect implementations that deviate from the intended behavior, potentially leading to additional iterations to correct the implementation. This inefficiency negatively affects the flexibility of improving graphical notations and limits the accessibility to software developers. Furthermore, the reliance on software developers to implement graphical notations in diagram editors increases costs and development resources. All in all, it is an unnecessary hassle if we could utilize theory that has already been demonstrated to work.

Existing tools such as Eclipse and their GMF use MBSE principles, including meta models, to configure graphical notations in graphical modeling languages, but their way of doing so is too technical, complicated, and only suitable for software developers. The tools are very static in their usability because end-users go through many generative and time-consuming steps to configure graphical notations. When all steps are completed, a codebase for the diagram editor is created, and this must be loaded and run every time a new graphical notation modification is made. Existing tools are also heavyweight and need to be downloaded onto the computer to be used. Altogether, current tools are not suited for graphical notation developers who are not programmers, and they can even be a hassle for software developers to use. This thesis explores how MBSE principles demonstrated by existing tools can be better used to create a lightweight diagram editor that still utilizes meta models; however, where these meta models can be easily configured by graphical notation developers without requiring high technical knowledge.

The goal of this thesis is to use existing theory and tools to implement a universal diagram editor. We claim that a universal diagram editor is a diagram editor whose graphical notations can easily be defined and customized by anyone if they have already developed a meta model in concrete syntax for the language they want to realize. To contribute to the ease of defining graphical notations, the customizations should be dynamically reflected in the diagram editor, without graphical notation developers having to statically generate the codebase for the editor. Such a universal diagram editor would eliminate the dependency on software developers. The thesis aims to answer the following research questions:

- What are the key challenges in developing a universal diagram editor and how can they be addressed?

- How can we allow graphical notation developers to easily define and customize graphical notations without programming?
- How can a diagram editor be implemented in a dynamic way such that when its graphical notations are customized, the customizations are automatically reflected in the editor without having to generate new code for the editor?

Developing a universal diagram editor presents several challenges. First, existing tools demonstrate the ability to use meta models to define graphical notations but remain inaccessible due to their technical complexity. Simplifying this process while maintaining flexibility is difficult. Second, ensuring that customizations to graphical notations dynamically reflect in the diagram editor requires designing a system architecture that supports real-time updates without code generation processes. Finally, creating an intuitive user interface (UI) for non-programmers while maintaining technical capabilities is an additional challenge.

3.1 Related Work

This section dives deeper into existing approaches for developing graphical notations and their implementation in diagram editors. The aim is to gain insights into the current knowledge base and identify ideas that can inform the design of the universal diagram editor.

To identify relevant approaches, a systematic review was conducted, focusing on tools and techniques that enable the creation and customization of graphical notations and their application in diagram editors. The following criteria guided the selection of related work:

- Tools must support customizing graphical notations and their use in a diagram editor.
- Techniques must align with the principles of meta modeling.

The sources of information included academic publications, technical documentation, and hands-on empirical evaluations of the tools. The tools chosen are Eclipse, MetaEdit+, and Microsoft Visio because they directly meet the criteria. In addition, it is reviewed how meta modeling platforms are generally structured.

The selected tools generally adopt the following approaches:

1. **Meta Modeling:** Used to define modeling languages.
2. **Graphical Representations:** Used to define the appearance of meta model elements, including their shape and style.
3. **Mapping Models:** Used to create links between meta model elements and their graphical representations.
4. **Generative Mechanisms:** Used to let the system generate required models or to generate the codebase for a diagram editor.

3.1.1 Eclipse

The Eclipse Foundation provides a variety of open-source software. A popular tool of theirs is the Eclipse IDE. This tool works as a Rich Client Platform (RCP), meaning that the IDE provides the basic functionalities, and then the client can install plug-ins to extend the tool with the desired functionality. [4] One interesting plug-in is the GMF, which enables clients to build a diagram editor with a custom graphical notation using meta models. [2] In this section, we will look at how the GMF works in-depth and what it consists of.

Overview of GMF Components

The GMF is an Eclipse plug-in that builds on multiple layers of technologies to enable the creation of diagram editors with custom-defined graphical notations. For GMF to function, it relies on several foundational components:

1. **Eclipse Modeling Framework (EMF):** Provides the foundation for defining meta models of the graphical notations. These meta models are created using the Ecore model, which acts as the core model, also known as the meta meta model.
2. **Graphical Definition:** This layer enables users to define the graphical representations (concrete syntax) of meta model elements, specifying shapes and styles.
3. **Tooling Definition:** Defines which meta model elements should appear as stencils in the editor's palette.
4. **Mapping Definition:** Establishes the links between meta model elements and their graphical representations.
5. **Code Generation:** GMF combines the data from these layers to generate Java code, which implements a diagram-based editor where the custom-defined graphical notation can be used.

These components work together to allow users to define, map, and visualize custom diagram notations in a diagram editor. Now, we will dive deeper into EMF, the foundation on which GMF is built.

Eclipse Modeling Framework (EMF)

The EMF is a modeling and code generation framework to define meta models and generate Java code for a textual tree-structured editor, which will be presented. EMF uses the Ecore model as its core meta meta model. Ecore allows for defining classes, attributes, and relationships in a structured, object-oriented manner while maintaining compatibility with standards like the Meta Object Facility (MOF). [5, 6]

Figure 3.1 shows the Ecore model and its elements, essential to create structured models within the EMF.

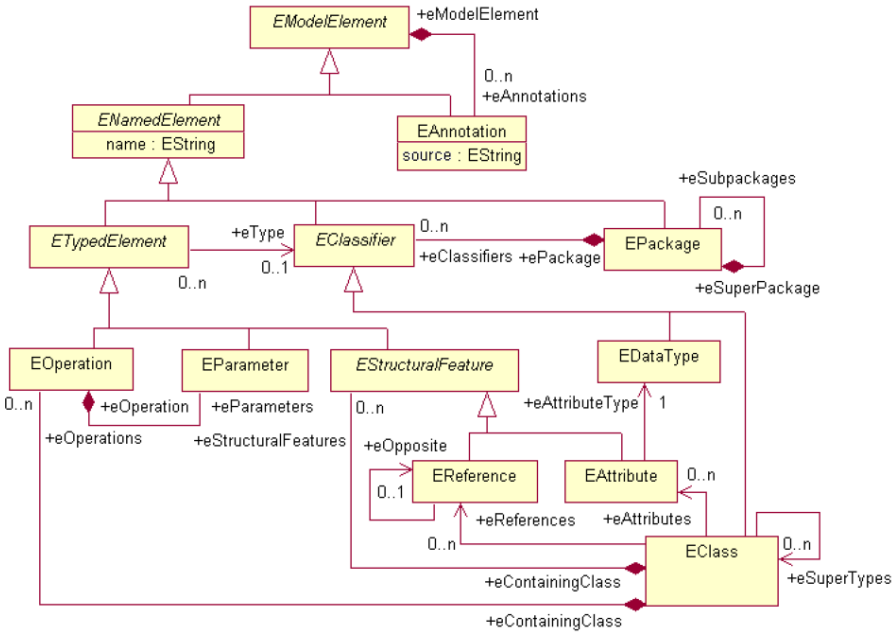


Figure 3.1: Kernel of Ecore Model

The Ecore model contains elements such as:

- **EClass:** Represents a class in the model.
- **EAttribute:** Defines the attributes of a class.
- **EReference:** Specifies relationships between classes.
- **EDataType:** Provides data types for attributes (e.g. *String*, *Integer*).

Furthermore, *Eclasses* can be grouped into *EPackages*, which can be further grouped into subpackages. Each model element in EMF can also be annotated using *EAnnotation*. Abstract classes, such as *ENamedElement* and *ETypedElement*, provide structure and reuse for common properties (e.g., names or types).

To understand these concepts, consider the construction of a simple model using the Ecore model in the EMF 3.2.

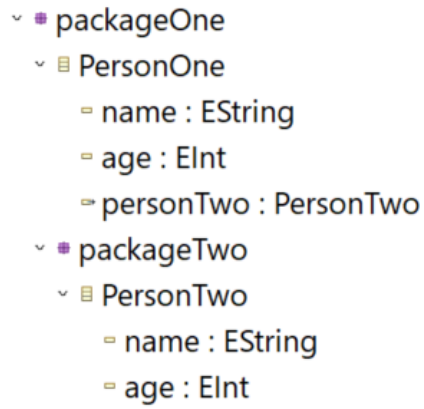


Figure 3.2: Instance Model Example Using The Ecore Model

The model is shown in a tree-structure view. In the model:

- *packageOne* and *packageTwo* are instances of *EPackages*. Each *EPackage* serves as a container grouping related *EClasses*.
- Inside *packageOne*, the *PersonOne* class is defined as an instance of *EClass*. It contains:
  - *EAttributes name* and *age*, which define properties of the class.
    - The *name* attribute is of type *EString*, representing a string *EDataType*.
    - The *age* attribute is of type *EInt*, representing an integer *EDataType*.
  - An *EReference personTwo*, which creates a relationship between the *PersonOne* class in *packageOne* and the *PersonTwo* class in *packageTwo*.
- *packageTwo* contains the *PersonTwo* class, also defined as an *EClass*, with similar attributes as the *PersonOne* class.

The model presented in Figure 3.2 only explained the structure of Ecore. It is not a meta model and does not describe a modeling language. Let us try to create a meta model that can be used to create other models. In Figure 3.3, the concrete syntax of a simple meta model is shown. This meta model defines a language that can model boxes and their relationships.

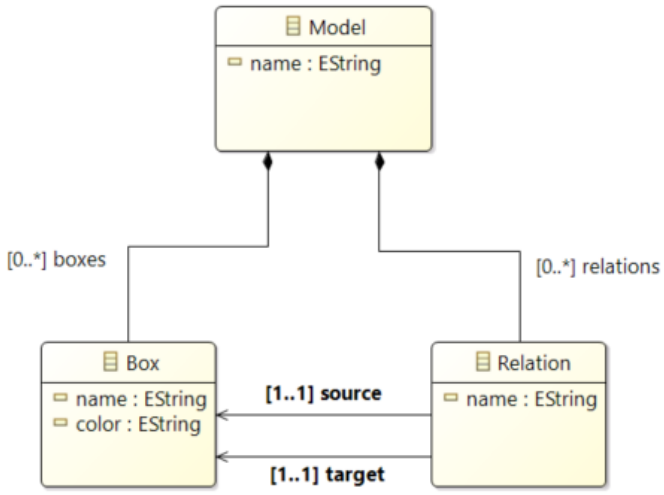


Figure 3.3: Simple Meta Model for a Box Modeling Language

An instance model of the language would have a name, and the model would consist of boxes and relations. Boxes and relations also have names. In addition, the boxes have colors. Relations will always reference one source box and one target box. The tree structure view of our box modeling language can be seen in Figure 3.4.

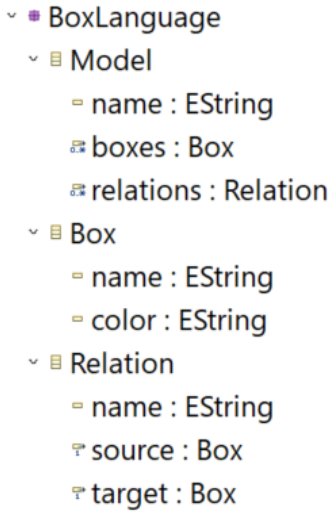


Figure 3.4: Tree-Structure View for the Box Modeling Language

Once a meta model has been defined using the Ecore model, Java code can be generated from it for domain classes or an editor application. Most interesting to this thesis is the editor application; however, EMF's editor application is a tree-structure-based editor like the one seen in Figure 3.4 and not a diagram-based one. This is where the GMF comes in.

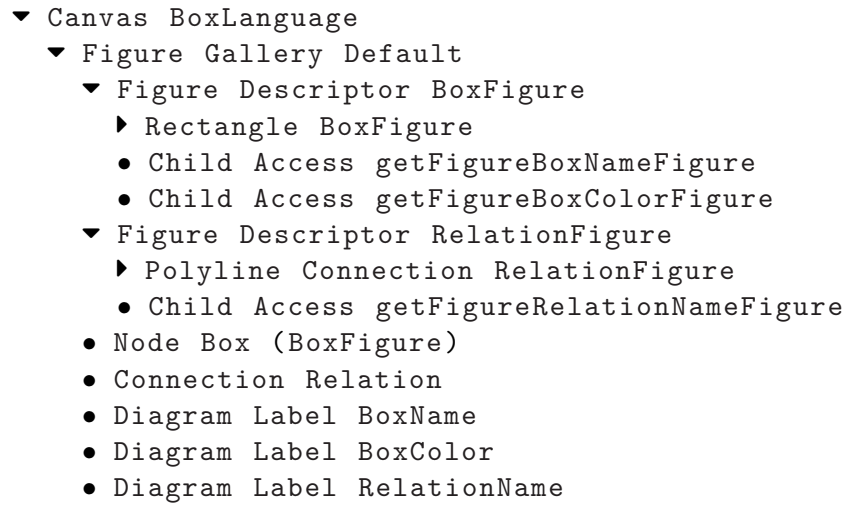
Graphical Modeling Language (GMF)
The GMF is an Eclipse plug-in that builds on the EMF to enable the creation of diagram-based graphical editors. The GMF provides a set of generative tools that allow users to define the appearance of custom modeling languages by extending the meta model with a graphical dimension.

The GMF requires defining three core models to generate code for a graphical diagram-based editor. These models are: [2, 7]

1. **GMF Graph:** This model defines the graphical representation of a meta model concept. It specifies how an element should be visually represented, e.g., as a shape, line, or label.
2. **GMF Tool:** The tool model defines the tools, also known as the meta model concepts, that should be available in the diagram editor palette. These tools are then selectable for the user to drag onto the diagram canvas.
3. **GMF Map:** The mapping model connects the graphical representations defined in the GMF Graph Model and the tools defined in the GMF Tool Model to the meta model elements. This mapping is used to give each concept of the modeling language an appearance. This mapping bridges the abstract syntax (defined in EMF) and the concrete syntax (defined in GMF).

After defining these models, GMF generates a **GMF Gen Model**, which contains all the necessary information to produce Java code that implements the diagram-based editor. The Java code for the diagram editor can finally be run, and the defined graphical modeling language can be used inside the diagram editor.

To better understand how these GMF concepts work and are applied in practice, consider the creation of a diagram-based editor with the box meta model defined in Figure 3.3. In the Eclipse IDE, we begin by defining the GMF graph model for the box modeling language to tell what appearance our meta model concepts should have. The defined model can be seen in Listing 3.1.



Listing 3.1: GMF Graph Model for the Box Modeling Language

The model shows that graphical representations are defined for both the *Box* and *Relation* classes. The *Box* class is a node represented as a rectangle, and its attributes *name* and *color* are represented as labels. The *Relation* class is a connection represented as a polyline, and its attribute *name* is represented as a label.

Next, we define the GMF tool model for the language to specify which meta model concepts should be available in the palette of our diagram editor. The defined model can be seen in Listing 3.2.

```
▼ Tool Registry
  ▼ Palette BoxLanguagePalette
    ▼ Tool Group BoxLanguage
      ▶ Creation Tool Box
      ▶ Creation Tool Relation
```

Listing 3.2: GMF Tool Model for the Box Modeling Language

Our GMF tool model shows that our palette consists of the *Box* and *Relation* concepts from our meta model.

Now, we define the mapping from our GMF graph model and GMF tool model to our meta model. The defined mapping model is presented in Listing 3.3.

```
▼ Mapping
  ▼ Top Node Reference <boxes:Box/Box>
    ▼ Node Mapping <Box/Box>
      • Feature Label Mapping false
  • Link Mapping <Relation{
                   Relation.source:Box->
                   Relation.target:Box
                  }/Relation>
```

Listing 3.3: GMF Map Model for the Box Modeling Language

The elements of the mapping model are mainly formatted as *"(Diagram Concept) Mapping <(Meta Model Concept)/Graph Model Concept)>*. For example, the *Box* meta model element and the *Box* graph model element are mapped to a *Node* in the diagram editor. Similarly, the *Relation* meta model concept and the *Relation* graph model element are mapped to a *Link* in the diagram editor.

The link mapping requires two underlying properties that specify which references from the *Relation* meta model concept are mapped to the source and target of the *Link*. These are called *source features* and *target features*. In our case, the mapping specifies the source feature as *Relation.source:Box*, indicating that the *source* reference from the *Relation* concept, which points to the *Box* concept, should be mapped as the *source* of the *Link*. Likewise, the *target* feature is specified to map the *target* reference from the *Relation* concept to the *target* of the *Link*.

Finally, the GMF Gen model can be generated and it can be used to generate the code-base for the diagram editor. However, the Eclipse GMF Tooling plug-in, which enables this process, is no longer maintained and has not been updated for over eight years. Additionally, the repository hosting the GMF Tooling package is no longer available. As a result, it was not possible to generate the code for the editor, and therefore, this step is not demonstrated in this thesis.

Object Constraint Language (OCL) in Eclipse
The Object Constraint Language (OCL) is a formal specification language standardized by the Object Management Group (OMG). It is often used in MBSE and meta modeling to define constraints or querying models. In Eclipse, OCL is integrated with EMF to force rules on diagram instance models which are derived from meta models. [8]

A simple example of an OCL constraint in EMF could be for arcs in a Petri Net. The arcs in a Petri Net connect either a *Place* to a *Transition* or a *Transition* to a *Place*. This rule can be expressed as shown in Listing 3.4.

```
context Arc
inv: (self.source.oclIsKindOf(Place) and
      self.target.oclIsKindOf(Transition)) or
     (self.source.oclIsKindOf(Transition) and
      self.target.oclIsKindOf(Place))
```

Listing 3.4: Example OCL Constraint

This constraint ensures that an arc's *source* and *target* properties adhere to the rules of a valid Petri Net connection.

To apply OCL constraints to meta model concepts in Ecore, the OCLinEcore editor can be used to define constraints directly in the meta model. For the Petri Net example, we would construct a rule like seen in Listing 3.5.

```
class Arc {
    property source : Node[1];
    property target : Node[1];

    invariant validConnection :
       (source.oclIsKindOf(Place) and
        target.oclIsKindOf(Transition)) or
       (source.oclIsKindOf(Transition) and
        target.oclIsKindOf(Place));
}
```

Listing 3.5: Example OCL Constraint in Ecore

When OCL constraints are added to a concept in the meta model, that concept becomes the OCL context. All constraints written for the concept are evaluated against its instances, which ensures that instance models adhere to defined rules.

3.1.2 MetaEdit+

MetaEdit+ is another tool for defining graphical modeling languages and using them in a diagram editor. Unlike EMF, which bases its meta modeling on the Ecore model, MetaEdit+ utilizes the GOPRR model. GOPRR stands for graph, object, property, role, and relationship, and represents the primary elements used to define the structure, relationships, and appearance of graphical modeling languages within MetaEdit+. A snippet of the GOPRR model can be seen in Appendix A.2.

The GOPRR concepts are: [9]

- **Graph:** The graph is the highest level of abstraction, and it serves as the main container for models. Each graph type defines a graphical modeling language.
- **Object:** For each graph type, there are multiple objects, which are the individual model elements, representing the concepts of the graphical modeling language.
- **Property:** Properties describe the attributes of objects, roles, or relationships and can be in various data types such as strings or numbers.
- **Role:** Roles define how objects take part in a relationship type.
- **Relationship:** A relationship type links objects within the graph.

To better understand these concepts, we can utilize the meta model of our simple box modeling language presented in Figure 3.3. Originally, this meta model was structured using the Ecore model. Figure 3.5 demonstrates how the meta model can be converted into a meta model based on the GOPRR model.

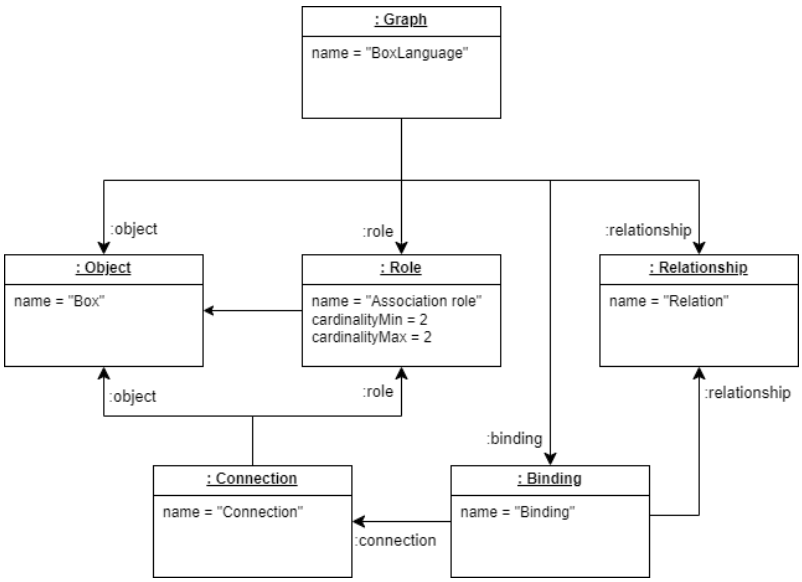


Figure 3.5: Meta Model for a Box Modeling Language with GOPRR

The graph type for our modeling language is named BoxLanguage. This language consists of a *Box* object, an *Association* role, and a *Relationship* named Relation. In GOPRR, every relationship must be defined with two roles, which in Ecore terms can be understood as source and target. The relation in our meta model connects two Box objects, with one serving as the source and the other as the target. GOPRR includes the concepts of bindings and connections to specify the objects that roles can connect

to and the relationships they belong to. Binding specifies the relationship and its roles. Connection specifies which objects can participate in the relationship.

The graphical appearance can be applied to objects, roles, and relationships. This is done through MetaEdit+'s built-in symbol editor, where the appearance can be drawn. In Figure 3.6, it is shown how an appearance can be drawn for the *Box* object.

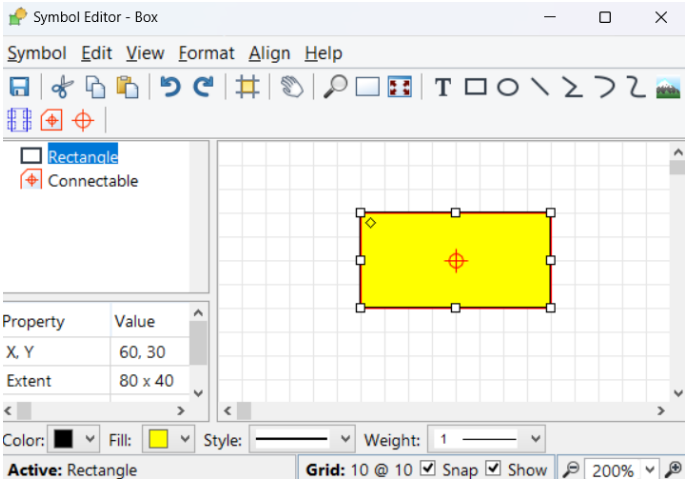


Figure 3.6: MetaEdit+'s Symbol Editor

The top panel shows the predefined shapes that can be used to draw the appearance of the meta model concept. A rectangle shape was used for the *Box* concept. A connectable shape was attached to the rectangle to allow relationships to connect to it. The bottom left tooltip shows the x and y positions of the top left point of the shape placed on the canvas. The tooltip also shows how much the shape extends to the right and downward.

The meta model editor and the symbol editor function dynamically, meaning that when a concept or its graphical representation is changed, it is reflected directly in the diagram editor. Unlike GMF, you do not need to generate code and re-run the diagram editor application. In MetaEdit+, code generation is only necessary if you want the domain classes of your instance model, which is also possible in the GMF. [9]

3.1.3 Microsoft Visio

Microsoft Visio is a diagram editor for creating diagrams of many types, ranging from flow diagrams and organization diagrams to brainstorming diagrams. It can integrate with other applications of the Microsoft Office suite. Mircosoft Visio uses the following terms for its diagram editor: [10]

- **Master:** An object in a stencil. It occurs in two forms: a simple shape or a connection shape. In graph terms, a simple shape and a connection shape would be considered as a node and an edge, respectively.
- **Stencil:** A collection of masters.
- **Shape:** An object on a drawing page.

- **Template:** A document that includes one or more drawing pages. A template can also include multiple stencils and background page designs.
- **Workspace:** A collection of windows. At a minimum, the workspace consists of a drawing window and zoom settings for pages. It can also include a Shapes window that contains one or more stencils. The workspace can also include the Shape Data window.

Visio supports the creation of meta models through its stencil-based approach, which enables the definition of the abstract and concrete syntax of a modeling language. *Stencils* group *Masters* and *Masters* represent the modeling elements of a language. Each *Master* has an associated graphical representation (Shape) for the concrete syntax and data properties for the abstract syntax. The *Stencil* serves as the meta model for a language, while the *Masters* define the model elements of the language. The core meta modeling structure of Visio is illustrated in Figure 3.7.

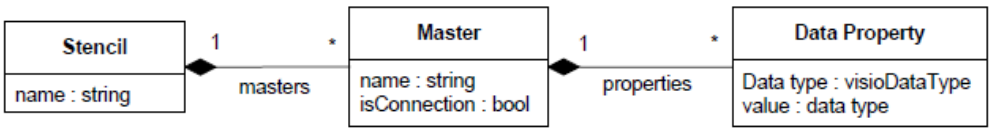


Figure 3.7: Visio's Core Meta Model. Source: [11]

As a practical example, we use the meta model of the box modeling language, previously introduced in Figure 3.3, and reframe it within Visio's core meta modeling structure. The resulting meta model is shown in Figure 3.8.

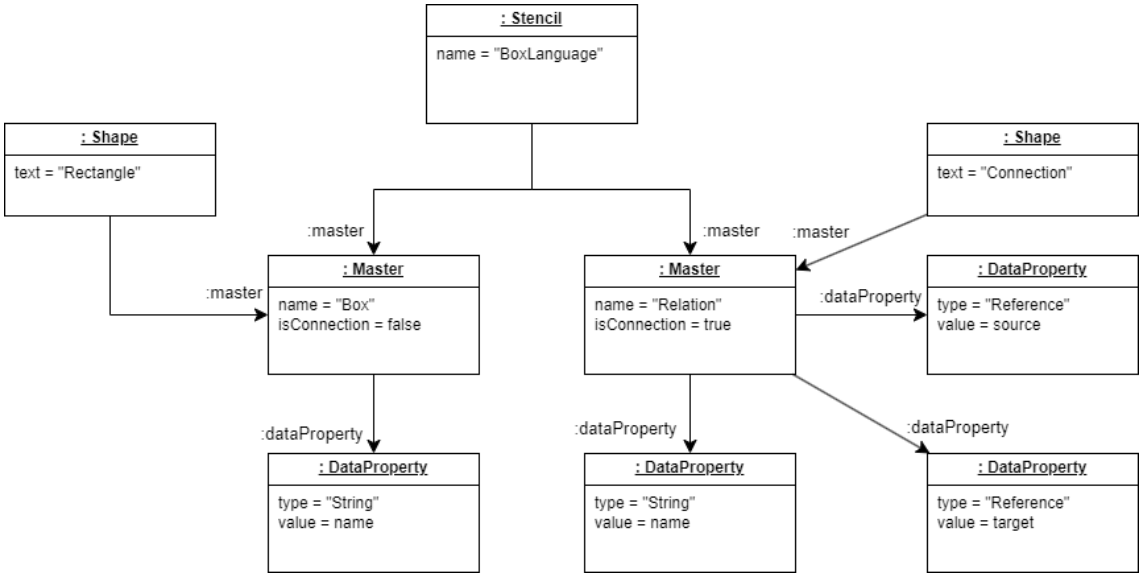


Figure 3.8: Meta Model for the Box Modeling Language with Visio's Core Meta Model

In the model:

- The *Stencil* contains the core model elements of the box modeling language: the masters *Box* and *Relation*.

- Each *Master* defines both:
  - The *concrete syntax:* the graphical appearance (e.g., rectangle for Box, connection for Relation).
  - The *abstract syntax:* the data properties (e.g., name and color for Box, and source and target for Relation).

When these meta models are used to create instance models, they are saved within the *Visio Object Model*, which manages the run-time structure of diagrams. The Visio Object Model can be seen in Appendix A.3.

The Visio Object Model organizes diagrams hierarchically: [10]

- **Document:** Stores the entire diagram.
- **Page:** Represents a canvas within the document that holds diagram elements.
- **Shape:** Represents an instantiated object derived from a master in the stencil.
  - Each shape references a Master from the stencil, inheriting its graphical and abstract syntax definitions.
  - Shapes can contain other shapes or be connected to other shapes through relationships.
- **Data Properties:** Stores run-time attribute values for shapes and pages, such as names, colors, or references to other shapes. Supported data types include text, numbers, and dates.

For example, using our meta model for the box modeling language to instantiate a diagram, its underlying vision object model would look like the model seen in Figure 3.9.

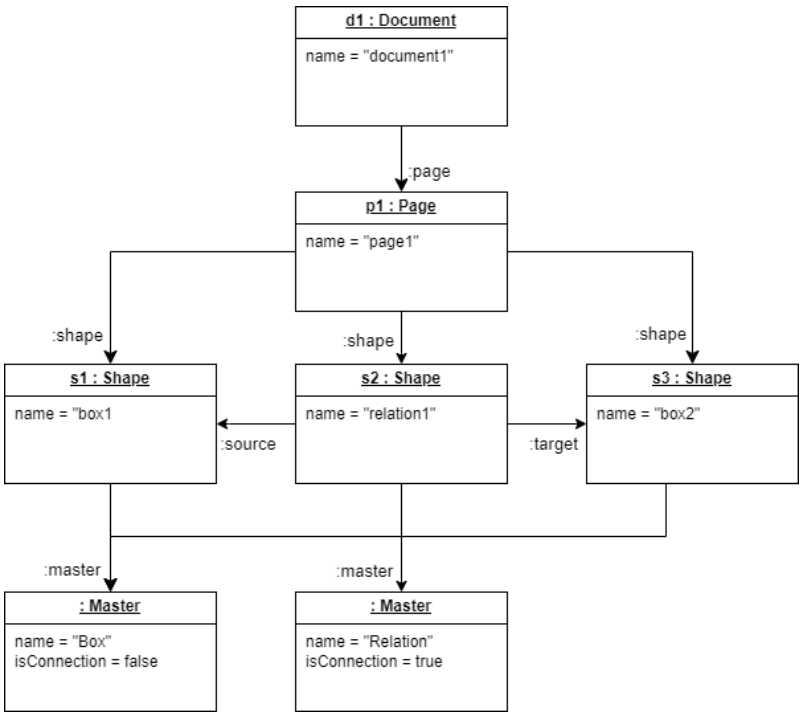


Figure 3.9: Visio Object Model of the Box Modeling Language

The page would contain two shapes instantiated from the *Box* master and connected by a shape instantiated from the *Relation* master. Each shape would store run-time values for its data properties (e.g., the name of a box or the source and target of a relation). This distinction between the stencil model (modeling language meta model) and the object model (instance model of a modeling language) shows how Microsoft Visio separates the definition of a language from its usage in diagrams.

3.1.4 Comparison of Approaches

To systematically assess the expressive power of the reviewed approaches, a comparison is provided in Table 3.1. Each approach is assessed across a structured list of categories relevant to defining and using graphical notations in diagram editors. These categories are as follows:

1. **Configurability:** Whether the tool supports the configuration of the abstract and concrete syntax of graphical notations and their use in a diagram editor.
2. **Flexibility:** The extent to which users can define and customize the abstract and concrete syntax of graphical notations.
3. **Ease of Use:** The scale of how easy it is to use the tool for non-programmers.
4. **Support for Constraints:** Whether the tool supports applying constraints to the custom-defined abstract syntax to ensure model validity in its diagram editor.

The comparison results are summarized in Table 3.1.

Tool/Framework	Configurability	Flexibility	Ease of Use	Support for Constraints
Eclipse GMF	Yes	High	Low	Yes
MetaEdit+	Yes	High	Low	Yes
Microsoft Visio	Yes	Low	High	No
Thesis's Universal Diagram Editor	Yes	High	High	Yes

Table 3.1: Comparison of Approaches

Generally, tools that enable the configuration of graphical notations and have high flexibility in doing so are difficult to use and not suitable for non-programmers. In contrast, tools that have low flexibility in the configuration of graphical notations are easier for non-programmers to use. The universal diagram editor aims to rank highly in both flexibility and ease of use while enabling the configuration of graphical notations.

GMF and MetaEdit+ enable both the configuration of graphical notations and their use in diagram editors. They do so with a high level of flexibility; however, their tools are difficult for non-programmers to use. Both tools are desktop applications that must be installed on one's computer. GMF's method of configuring graphical notations is very static, as one must procedurally define many different models to enable the final generation of a diagram editor where the graphical notations can be used. The generation creates a diagram editor codebase and is, therefore, also vulnerable to code errors. MetaEdit+

offers an intuitive way to configure the graphical appearance of graphical notations. However, its GOPRR meta model makes it difficult to define meta models of modeling languages. Because of its three-layered structure of objects, relationships, and roles, more meta model elements have to be defined compared to the Ecore model, where only classes and references are defined, which can be understood as objects and relationships in MetaEdit+. Both tools support the ability to apply constraints to their meta models.

Microsoft Visio can be used as a simpler tool for defining and using graphical notations in diagram editors. However, its ability to customize graphical notations is limited and does not support advanced graphical representations, such as dynamically updating a graphical appearance based on changes in the underlying models. For example, if a *Box* element in a model has a *color* attribute, that changes to *green*, Microsoft Visio cannot automatically change the graphical appearance of the box to green.

3.1.5 Meta modeling platforms

Karagiannis et al. [12] highlight the changing environment of organizations transitioning to flexible meta modeling instead of fixed ones. They mention that it is due to the rapid change of business requirements in today's world. Flexible meta modeling helps manage this complexity because the meta model can be freely defined and adapted to the problem. This section will provide a brief overview of their work, where they introduce meta model concepts, graphical representation concepts, and a generic architecture for developing meta modeling platforms.

Meta Models

A meta model is created using a modeling language. This modeling language is referred to as the meta modeling language, while the model defining the meta modeling language is the *meta meta model*. This hierarchical concept is essential to many meta modeling frameworks, including the Meta-Object Facility (MOF) standard by the Object Management Group (OMG). The MOF hierarchy uses a four-layer architecture similar to the Karagiannis et al. hierarchy illustrated in Figure 3.10.

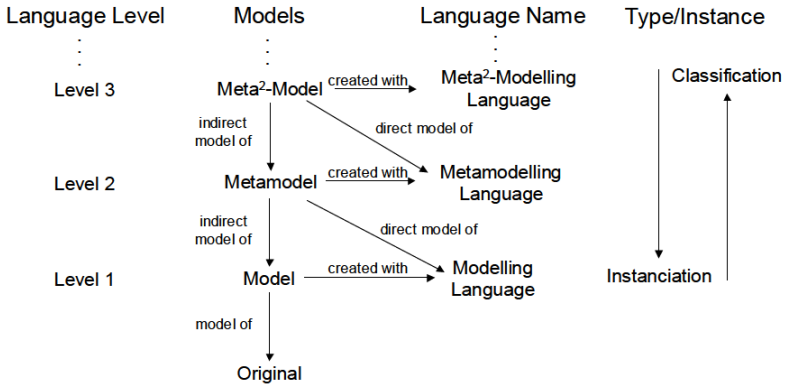


Figure 3.10: Metamodel Language Levels. Source: [12]

Both frameworks have a level 3 layer that consists of a meta meta model, which defines the foundation for creating meta models. The level 2 layer defines the modeling language, which in MOF is defined by the use of UML class diagrams, but here, general meta modeling languages are used to define custom modeling languages. At the level 1 layer,

models are created based on defined meta models. This corresponds to the same layer in MOF, where UML diagrams are instantiated. The level 0 layer corresponds to the actual run-time instance derived from the instance models at level 1.

Meta model language levels are not limited to a certain level; however, to finish the modeling hierarchy it is necessary to find a useful level of abstraction. Usually concepts such as ”thing”, ”property” and ”relation” are used to construct the top-most level model, but its semantics should be kept consistent as it is the top-most level model that provides the foundation for implementing the lower levels.

Graphical Modeling Language
A *graphical* modeling language is defined by its syntax, semantics, and notation. These elements collectively define how the language is structured and visualized:

- **Syntax:** Specifies the elements and rules for model creation, described using meta-models or graph grammars. UML class diagrams are often used to define the meta model of the syntax, while additional constraints can be expressed using languages such as OCL.
- **Semantics:** Assigns meaning to the syntactical constructs through a semantic domain and mapping. The semantic domain provides a foundation, such as mathematical models, while the semantic mapping connects syntactical constructs to their meaning.
- **Notation:** Defines how models are visualized.
  - Static notations use fixed symbols, such as rectangles for classes or arrows for relationships.
  - Dynamic notations extend static features by including a *representation part* (static symbols) and a *control part* (rules for adapting visualization based on the model state).

To bridge the *syntax* and *notation*, meta modeling platforms incorporate *mappings* in its *semantics*, which defines how elements of the meta model are visualized. For example:

- **Classes:** Mapped to rectangles, with attributes displayed as labels within compartments.
- **Relationships:** Represented as lines with arrowheads and labels that specify their type or cardinality.

Mechanisms, as described by Karagiannis et al., provide the functionality to use and evaluate models. These mechanisms operate on level 2 and 3.

Generic Architecture
The generic architecture that Karagiannis et al. present to build meta modeling platforms can be seen in Figure 3.11.

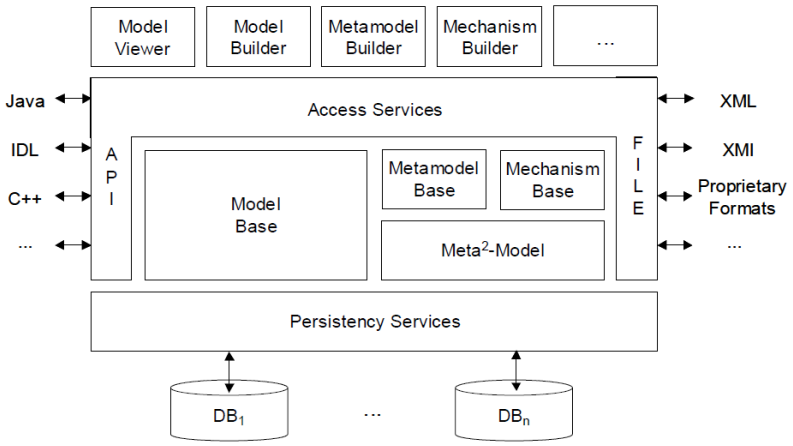


Figure 3.11: Generic Architecture of Meta Modeling Platforms. Source: [12]

The storage model information is managed by *persistency services*. The *meta meta model* provides the concepts to create meta models and mechanisms. It is the central part, as it provides the conceptual foundation and is connected to all other components. The *meta model base* contains information on all meta models currently managed by the platform. Changes in the meta model base are reflected in the model base to keep them consistent. The *mechanism base* contains information on the functionalities to be applied to the models. The *model base* contains all the models based on the meta models. *Access services* provide online interfaces to the different types of bases. In addition to access services, different *viewer* and *builder* components support the use and development of the meta modeling platform, such as *model builder*, *meta model builder*, and *mechanism builder*.

3.2 Methodology

The previous sections provided insight into existing tools that enable graphical notation configuration and their use in diagram editors. We looked into their approaches and identified their strengths and weaknesses. Now that we have gotten a general idea of how these tools are structured and what concepts they are using. We introduce Design Science Research (DSR) as the methodological framework for systematically developing and assessing the artifact, the universal diagram editor.

3.2.1 Design Science Research (DSR)

DSR focuses on creating artifacts in the form of models, methods, or systems that help to solve practical problems, especially within information systems. These artifacts address specific problems that can generally be categorized into two types: [13]

1. Problems where the current state is viewed as truly unsatisfying and the desirable state is viewed as neutral.
2. Problems where the current state is viewed as neutral, and the desirable state is viewed as a potentially huge and surprising improvement.

For a research project to be classified as DSR, it must meet the following criteria: [13]

1. **Research Strategy**: The project must select an appropriate research strategy that includes methods for collecting and analyzing data. Additionally, it must evaluate the artifact produced using adequate methodologies.
2. **Knowledge Inclusion**: The research must relate the produced results to existing theories, models, and artifacts, ensuring that it builds upon and extends the knowledge base in the field.
3. **Dissemination**: The project results must be shared with both researchers and professionals within the field.

DSR focuses not only on the creation of new artifacts but also their systematic evaluation, ensuring that they address the identified problem.

DSR Framework

A DSR framework provides a structured approach to ensure systematic development and evaluation. A widely used DSR framework is from Hevner et al. [14], which outlines three essential DSR cycles: the relevance cycle, the rigor cycle, and the design cycle, which guide the research. The cycles are shown in Figure 3.12.

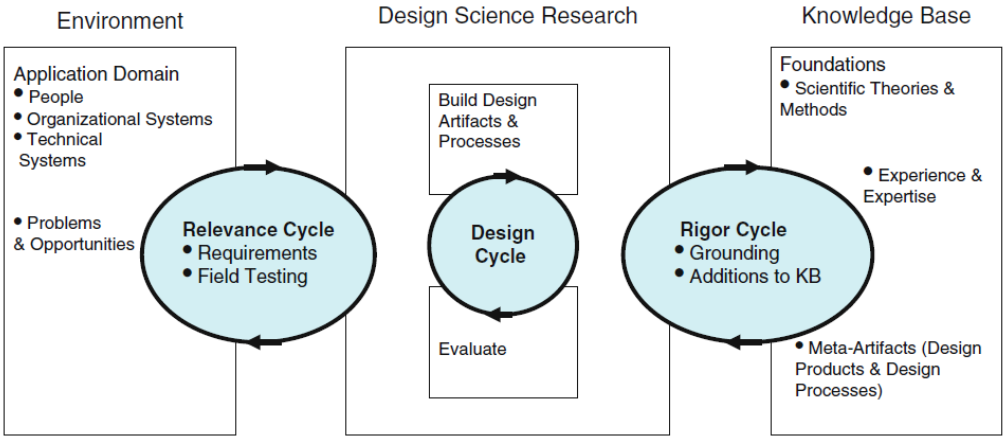


Figure 3.12: DSR cycles

Relevance Cycle

The relevance cycle connects the research project to its contextual environment. It defines the problem and identifies the requirements for the artifact. This cycle ensures that the artifact developed addresses real-world needs. The cycle also defines the acceptance criteria for evaluating the artifact in which the evaluation results decide whether or not more iterations of the relevance cycle are necessary. [14]

Rigor Cycle

The rigor cycle ensures that the research is informed by a knowledge base consisting of scientific theories, engineering methods, and artifacts. This cycle ensures that the artifact is developed based on established best practices while also contributing new knowledge to the field. [14]

Design Cycle
At the center of the DSR framework is the design cycle, which iterates between building and evaluating the artifact. The design cycle is where the artifact is created, tested, and improved. Although the design cycle is a fundamental concept in DSR, different researchers may outline its steps with slight variations. Kuechler et al., for example, have discussed the central design cycle and the steps and results that it should have and have come up with the following iterative steps, which align well with the goals of this research: [14]

1. Awareness of the problem, outputs a proposal.
2. Suggestion, outputs a tentative design.
3. Development, outputs an artifact.
4. Evaluation, outputs performance measures.
5. Conclusion, outputs results.

By combining these cycles, DSR provides a structural process to ensure that the artifact created is both practically relevant and grounded in research. The next section describes how DSR has been applied to the development of the universal diagram editor in this thesis.

3.2.2 Application of DSR in This Thesis

In this thesis, DSR is applied to create a universal diagram editor that addresses a clear and practical need in software engineering. By following the DSR framework, the thesis ensures that the developed artifact is grounded in theory and addresses a real-world need, contributing to the field of software engineering. Figure 3.13 illustrates the application of DSR cycles in this thesis.

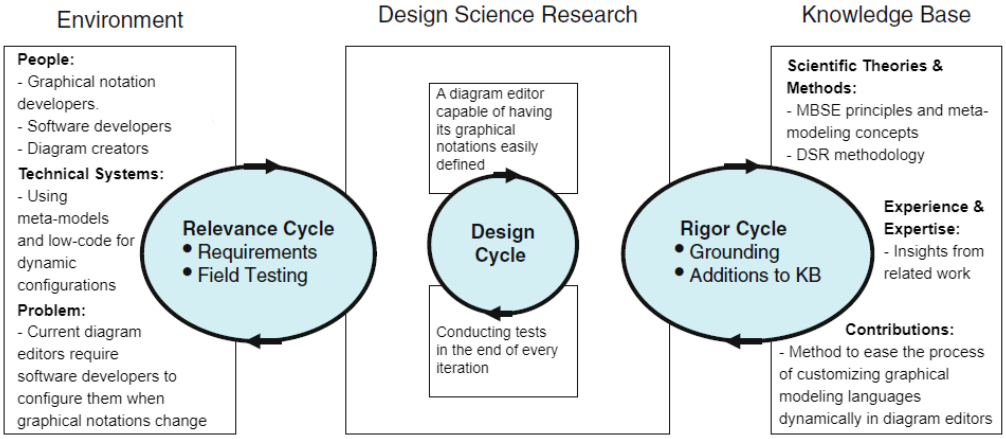


Figure 3.13: Application of DSR cycles in this thesis

Relevance Cycle
The relevance cycle identifies the requirements for the universal diagram editor. The application environment involves three main groups of people. First, graphical notation developers who need a flexible diagram editor to customize their graphical modeling languages. Second, software developers who are normally tasked with implementing

these customizations. Lastly, diagram creators who want to create diagrams with some graphical modeling language.

The technical systems used in this thesis include meta models and low-code principles to ease the process of customizing graphical notations. The primary problem addressed in this thesis is the dependency of current diagram editors on software developers for implementing graphical notations whenever changes are required. This dependency leads to delays, higher costs, and a lack of flexibility to adapt to newer requirements.

The requirements for the universal diagram editor were derived by an analysis of existing tools as presented in Section 3.1, where their strengths and weaknesses were evaluated in Section 3.1.4.

While a full-scale field test is beyond the scope of this thesis, the artifact is evaluated through user tests and interviews. Feedback obtained from these tests and interviews will serve as a measure of the artifact's effectiveness and guide to improve it.

Rigor Cycle
The rigor cycle ensures that the research is informed by a knowledge base while also contributing to the field of software engineering. This thesis is grounded in established theories including MBSE principles, meta modeling concepts, and insights from existing tools.

Building on this knowledge, this thesis contributes to the knowledge base by introducing:

1. **A Dynamic Approach to Customize Graphical Notations:** Unlike existing tools, which require code generation when modifying graphical notations, this thesis proposes a method that enables dynamic customization. This is discussed in Section 7.3.
2. **A Dynamic Mapping Mechanism:** A mapping model that dynamically links meta model elements to their graphical representations without requiring user input. This is analyzed in Section 4.5.
3. **Integration of Low-Code Principles:** The dynamic customization approach, together with a non-technical and user-friendly UI, makes the thesis's universal diagram editor accessible for non-programmers. This was evaluated in Chapter 9.

The findings and contributions of this thesis are described in Chapter 9.

Design Cycle
The universal diagram editor is developed through iterative cycles of design, testing, and refinement. Initial iterations focus on core functionalities, particularly the customization of simple graphical notations with minimal visual complexity.

At the end of each cycle, usability tests will be performed to evaluate whether the tool meets the requirements identified in the relevance cycle. User feedback gathered from these tests informs later cycles, ensuring continuous relevant improvements.

4 Conceptual Analysis

This chapter presents the key concepts required to build a universal diagram editor. Graphical notations used in diagram editors can often be linked to *diagram concepts*, such as nodes, edges, labels, and compartments. These diagram concepts are characterized by their *graphical features*, which define their static appearance and dynamic visual behaviors. For example, a static graphical feature in UML class diagrams could be a class's rectangle shape and its class name text. A dynamic graphical feature could be a class's attributes, which extends/shrinks the class's rectangle shape whenever attributes are added/removed.

The analysis begins by examining selected modeling languages, including UML Class Diagrams, BPMN, and Petri Nets, to identify their graphical features. These languages are chosen because they represent diverse application domains, from software engineering to business process modeling, ensuring a broad understanding of graphical features.

Afterward, the identified graphical features are mapped to their generic diagram concepts, bridging the gap between modeling language's graphical features to generic diagram concepts.

After analyzing the graphical part of modeling languages, we dive deeper into their underlying structure, looking into how meta models are used to define modeling languages, how instance models are created to transition to lower-level models, and lastly, how a custom constraint language can be used to enforce specific modeling rules.

4.1 Graphical Features in Graphical Modeling Languages

This section identifies the various graphical features in different graphical modeling languages to understand the static and dynamic features that the universal diagram editor should be able to support.

The languages selected for analysis are:

- **Unified Modeling Language (UML) Class Diagram:** UML is a popular modeling language in software engineering, offering a range of diagram types to capture various perspectives of the system. Some often used UML diagram types are class diagrams, activity diagrams, use case diagrams, sequence diagrams, and state machines. [15]
- **Business Process Model and Notation (BPMN):** BPMN is designed to model business processes. Some graphical notations it uses are events, activities, and gateways. Its simplicity improves communication between business stakeholders and technical developers. [16]
- **Petri Nets:** A Petri Net is used to model control flow and synchronization. It has powerful methods to do so, particularly in systems that have asynchronous and concurrent activities. A Petri Net consists of circles called places and squares/bars called transitions. Places and transitions are connected by directed arcs. [17]

These languages were chosen because they serve to model different business purposes, which can help to identify more unique graphical features. Additionally, their underlying structure is common, as they are all graph-related, meaning that they consist of both nodes and edges. Non-graph-related languages do not use edges but mainly use nodes, often where nodes are embedded and relate to other nodes indirectly. For example, flow diagrams only consist of pointy rectangle nodes, where one node points to another by having them side by side, without including any edges in between. The languages chosen do, however, still have some non-graph-related features, which will be presented. However, complete non-graph-related languages such as flowcharts or UML Sequence Diagrams are excluded due to the scope of this thesis.

4.1.1 Analysis of Graphical Features

Now, as the graphical modeling languages have been chosen, we will proceed with the analysis of their graphical features, starting with the core graphical features of UML class diagrams, shown in Figure 4.1.

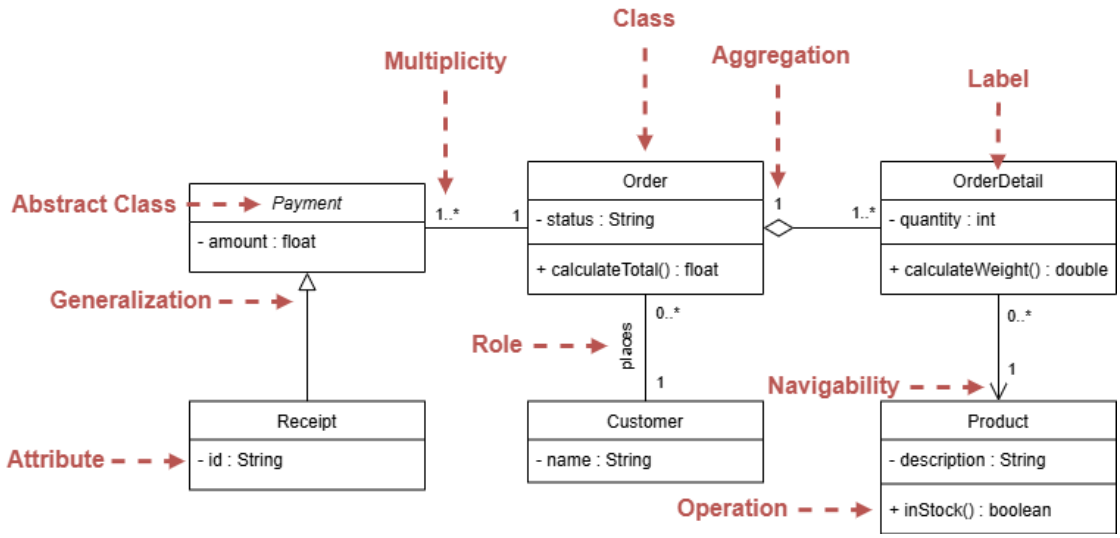


Figure 4.1: Graphical Features in a UML Class Diagram. Inspired by: [18]

From the diagram in Figure 4.1, the following graphical features were identified:

- Classes:
  - are represented as squares.
  - has one name, and it is represented as text within the topmost area of the square surrounded by a border.
- Attributes:
  - are represented as text within classes below the class name.
  - is dynamic, in which there can be 0 to infinite. When they are added or removed, the square of the class extends vertically to make them fit.
- Operations:
  - are represented as text within classes below the attributes.

  - is dynamic, in which there can be 0 to infinite. When they are added or removed, the square of the class extends vertically to make them fit.

- Associations:
  - are represented as solid lines.
  - can have text on their ends, representing the multiplicity between classes.
  - can have markers/icons on their ends, specifying the type of relationship, such as aggregation and generalization.

Now, fundamental graphical features from BPMN were identified, and they can be seen in Figure 4.2.

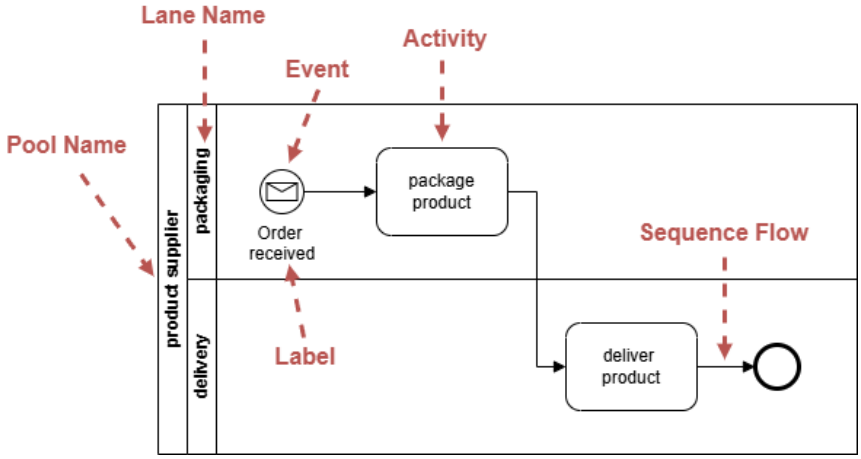


Figure 4.2: Graphical Features in a BPMN Diagram

The following features were found:

- Pools:
  - are represented as rectangles.
  - has one name, and it is represented as text within the leftmost area of the rectangle. The name text is aligned vertically, and it is surrounded by a border.
- Lanes:
  - are represented as rectangles within a pool.
  - has one name, and it is represented as text within the leftmost area of the lane's rectangle. The name text is aligned vertically, and it is surrounded by a border.
  - can contain events, tasks, and sequence flows.
- Events:
  - are represented as circles with some further detailing inside.
  - has one name, and it is represented as text outside of the circle.
- Activities:
  - are represented as rounded rectangles.

– has one description, and it is represented as text aligned in the center of the task's rectangle.

Lastly, the graphical features from Petri Net can be seen in Figure 4.3.

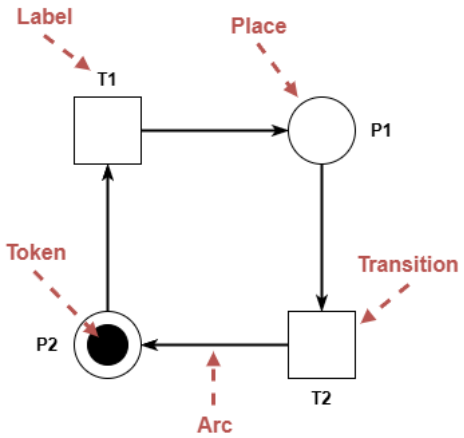


Figure 4.3: Graphical Features in a Petri Net

Here, the following features were found:

- Places are represented as white circles. Places can contain tokens.
- Transitions are represented as white squares.
- Arcs are represented as solid black lines with a closed arrowhead marker on one end.
- Tokens are represented as black-filled circles which are smaller than the circles of places.

Now that the graphical features have been found in our chosen modeling languages, we will map them to their respective diagram concept.

4.2 Mapping Graphical Features to Diagram Concepts

For diagram editors to interpret graphical features and show them on a diagram canvas, we need to map the features to generic diagram concepts. The generic diagram concepts are nodes, edges, and edge connection points. These concepts were derived based on their occurrence in existing diagram editor frameworks. Nodes are the core diagram concept as it is used in both graph and non-graph-related modeling languages. Nodes and edges can appear as anything graphically, and they can incorporate dynamic graphical features. Edges connect two nodes. Therefore, every edge requires a source and a target node. Connection points are used to tell where on a node an edge can connect to or from. The mapping between these concepts and the identified graphical features can be seen in Table 4.1.

Diagram Concept	Core Graphical Features Identified
Node	• Rectangles/Squares: Used to represent classes, activities, and transitions. • Circles: Used to represent places and events. • Annotations: Usually in the form of comments or notes, placed beside specific diagram elements to provide context or explanations. • Compartment: Represent areas that some nodes use to organize information – Classes may have compartments for attributes and operations. – Lanes for separating different actors or processes within the same diagram. – Places for visualizing tokens. • Label: Text elements describing node's or edge's meaning or specify additional information.
Edge	• Solid: The most common edge feature used to represent flow or control transfer. • Dotted/Dashed: Used to represent dependencies or less direct relationships. • Edge Head: Represents the direction of flow by showing a head marker at the ends of an edge. – Open arrow: Used for general direction indication. – Closed arrow: Indicates more definitive or stronger directional flow.
Connection Point	• All three languages had invincible connection points, where edges, by default, connected to the most centered point depending on the incoming side of an edge. If the incoming edge was from the left of a node, it would connect to the left center of the node. However, the edges could be manually dragged to point to another handle instead of the default one.

Table 4.1: Mapping between Identified Graphical Features to their Diagram Concept

The mapping in Table 4.1 categorizes identified graphical features into generic diagram concepts. However, some features require more advanced mappings. For example, in a Petri Net, *compartment* nodes (such as tokens) are visualized within *place* nodes based on some attribute value. Similarly, edges may or may not include heads at their ends. These advanced mappings require a deeper understanding of the attributes and references of meta model concepts.

These more complex mappings will be analyzed in Section 4.5 after presenting meta models. Advanced mappings often rely on the attributes and relationships defined in the underlying models to achieve the required appearance.

With the graphical features mapped to generic diagram concepts, the next step is to analyze the conceptual structure of the underlying models in the editor.

4.3 MBSE with Eclipse's Ecore Model

As identified in the work of Karagiannis et al. in Section 3.1.5, building meta modeling platforms requires a top-level meta model, also referred to as the meta meta model. For simplicity, throughout this thesis, the term *core model* will be used to refer to the meta meta model. The core model should be well structured, as it serves as the foundation for defining lower-level meta models that represent the modeling languages in our universal diagram editor.

As discussed in Section 3.1, existing tools adopt different core models to define their modeling languages. Eclipse uses the Ecore model, MetaEdit+, the GOPRR model, and Microsoft Visio, the Visio Object Model. This thesis uses the Ecore model developed by Eclipse as the core model because of its alignment with widely recognized UML. The Ecore model is based on the MOF, which was developed to support UML. Furthermore, its use in the EMF, often in combination with other frameworks such as GMF, demonstrates its scalability and robust structure. [9]

4.3.1 Overview of Meta Model Levels

To understand the models used in the universal diagram editor, a hierarchical structure is presented in Figure 4.4, inspired by [19]. The level 2 core model was introduced in Section 3.1.1, while the level 1 and level 0 models will be discussed further in this section.

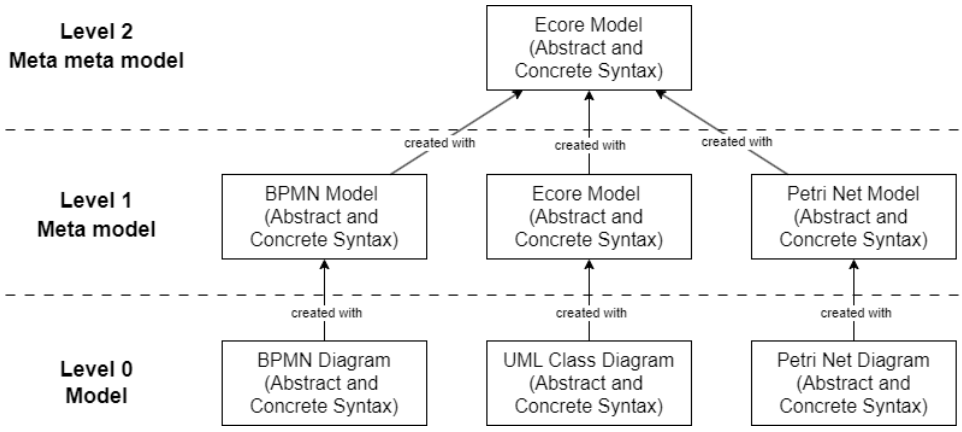


Figure 4.4: Meta Model Levels

First, all models at all levels can be represented in two forms: *concrete syntax*, which is the graphical representation of models typically used and understood by humans, and *abstract syntax*, which is the object-structured representation used by software systems. At the top-most level (level 2), the Ecore model serves as the meta modeling language that defines all modeling languages. At level 1, the prioritized modeling

languages that the solution aims to support are shown. These level 1 models are represented as UML class diagrams because their meta model, the Ecore model, serves as the meta model for UML class diagrams. This also allows the Ecore model to work as a one-to-one meta model for the UML class diagram modeling language at level 1. However, for languages such as BPMN and Petri Nets, their meta models must be individually defined, as the Ecore model cannot directly represent their language.

To illustrate how a non-UML modeling language can be defined, the meta model for Petri Nets is presented in Figure 4.5. This meta model demonstrates that Petri Nets can be modeled using UML notation i.e. the Ecore model.

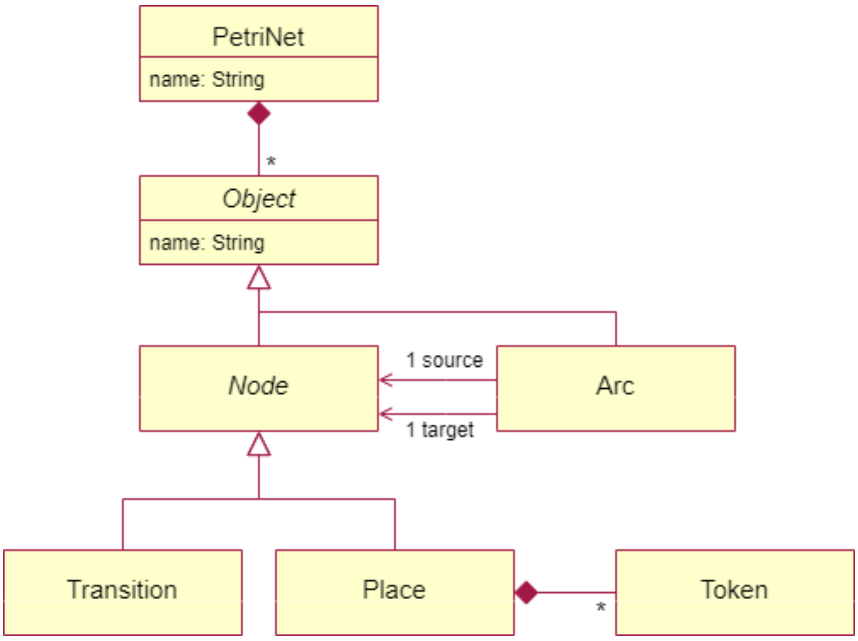


Figure 4.5: Petri Net Meta Model (Concrete Syntax) (level 1 of Figure 4.4). Source: [19]

The Ecore model is used correctly because everything within the Petri Net Meta Model can be seen as an *EModelElement*, and a model always has one core *EPackage*. All classes are represented as *EClasses*. Every *EClass* can have multiple *EAttributes*, *EOperations*, and *EReferences*. In this case, *PetriNet* and *Object* both have an *EAttribute* ”name” which is of *EDataType* ”String”. Lastly, we see *EReferences* appear in between all *EClasses*, as *EClasses* may include multiple *EReferences*. An *EReference* can point to another *eType* of type *EClassifier*. Since *EClasses* are also *EClassifiers*, this relationship is valid. Additionally, some *EReferences* are opposites to each other; these can be seen between the EClasses *PetriNet* and *Object* and between *Place* and *Token*.

From level 1 meta models, level 0 models, referred to in this thesis as *diagram models*, can be constructed. These level 0 models are what users typically interact with in diagram editors when they appear in their concrete syntax. However, level 0 also includes the abstract syntax model of the concrete syntax model, serving as the underlying structure to validate diagram instances.

Having established that the Ecore model will serve as the level 2 core model and

demonstrated its applicability to defining level 1 meta models like Petri Nets, the next step is to analyze the transition from level 1 to level 0. This analysis will use the UML class diagram language.

4.4 Diagram Model from Core Model

We need to understand how to use meta models to create instantiations of them that comply with their concepts. Instance models constructed at level 0 are our diagram models in abstract and concrete syntax. The abstract syntax would be the object model that our editor uses as the underlying structure to synchronize its nodes and edges and keep itself validly structured without taking into consideration the additional modeling rules that cannot be resembled purely by using models but instead through a custom constraint language. The additional application of rules will be looked into in the last part of this chapter. The instance model in concrete syntax form is the diagram model that has a graphical appearance. To illustrate these two instance model forms, consider the Ecore model shown in 4.6, which is used to create UML class diagrams. The Ecore model was already introduced in 3.1; however, for the simplicity of reading and understanding the transition to instance models, it is shown again.

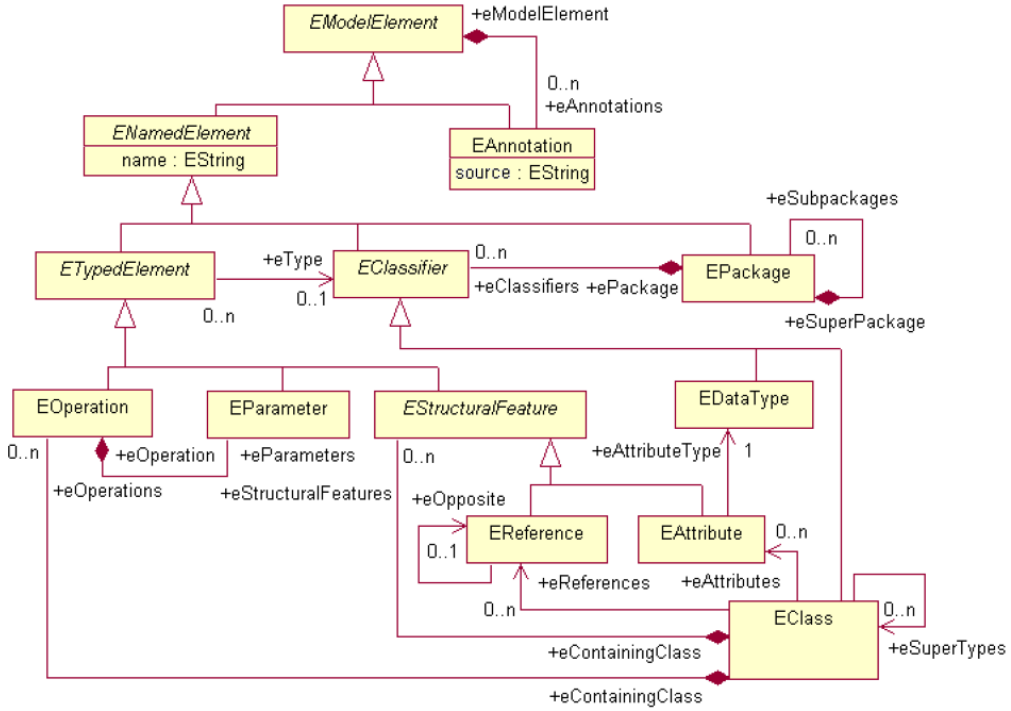


Figure 4.6: Kernel of Ecore Model (level 1 and 2 of Figure 4.4)

Now, we will use the Ecore model to construct one instance in two different representations: abstract and concrete syntax.

4.4.1 Diagram Model (Abstract Syntax)

To represent an instance model from a UML Class Diagram in abstract syntax, the UML object diagram language was used, as it fits its purpose of illustrating instances of classes in object format. Using the Ecore model, we construct the instance model seen in Figure 4.7. To understand the model, it is recommended to take a look at its concrete syntax

model first in the next Section 4.4.2, which shows the same instance model but in concrete syntax. This order of presenting the models was chosen because of the necessity to start from the core model, which is the top-most level, and then work chronologically down in levels.

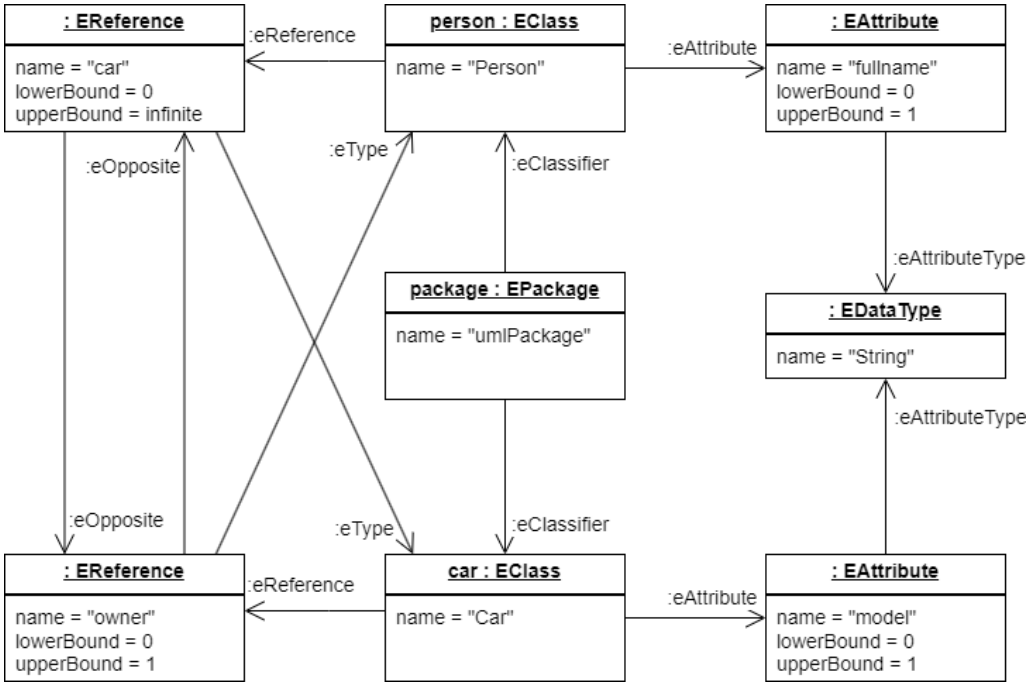


Figure 4.7: A Diagram Model in Abstract Syntax (level 0 of Figure 4.4)

When modeling the instance model, the process always starts from a base package. Here, the base package has an identifier ”package” and a name ”umlPackage”. The package consists of two classifiers: ”person” and ”car”, both of which are of type *EClass*. The person *EClass* has an *EAttribute* ”fullname” with boundaries, and it has an *EDataType* of ”String”. Similarly, the car *EClass* has an *EAttribute* ”model” with boundaries and an *EDataType* of ”String”. Furthermore, the person *EClass* has an association to an *EReference*, where the *EReference* has the name ”car”, boundaries, and points to another classifier, being the car *EClass*. Oppositely, the car *EClass* has an association to another *EReference*, where the *EReference* has the name ”owner”, boundaries, and it points to the person *EClass*. Therefore, these two *EReferences* are opposite to each other, and they reflect that, by having an association between them named ”eOpposite”.

The Ecore model itself can also be represented in abstract syntax. This is shown in Appendix A.2.

4.4.2 Diagram Model (Concrete Syntax)

Now that we know how the underlying instance model looks, we can construct the corresponding model in concrete syntax seen in Figure 4.8.

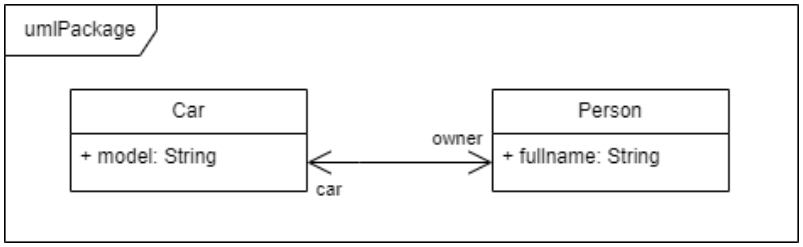


Figure 4.8: A Diagram Model in Concrete Syntax (level 0 of Figure 4.4)

The *EPackage* is mapped to a rectangle wrapping the *EClasses* where its upper left corner consists of a label that refers to the ”name” attribute of the *EPackage*. *EClasses* are mapped to rectangles with labels mapped to the *EClasses* ”name” attribute. Each of the *EClasses* ”eAttribute” associations is mapped to compartments inside of the class. The type of the *eAttributes* are gotten from the ”name” attribute of the object that they associate with through *eAttributeType*. The two *EClasses* have an association to each other through their individual *EReferences*. The two *EReferences* have an association to each other through ”eOpposite” which means that the two *EReferences* can be visualized as a merged association.

As shown, transitioning from abstract syntax to concrete syntax involves multiple mappings. We need to define a systematic mechanism to perform this mapping to make sure that in our diagram editor, graphical notations are represented correctly in their concrete syntax, and when they are drawn on a canvas, their underlying abstract syntax model remains intact. For this, a mapping model is defined at level 1, linking the meta model concepts in level 1 to their corresponding graphical apperances.

4.5 Mapping Graphical Features to Meta Model Concepts

In this section, we explore how graphical features can be mapped to meta model concepts. This is to provide the overall graphical representation of a model element. These mappings are categorized into two types: *direct mappings* and *advanced mappings*. Direct mappings are typically associated with static graphical features, whereas advanced mappings involve dynamic visual behaviors. Within our mapping model, graphical features are referred to as *shapes*, and the mapping model itself is part of the graphical representation model.

4.5.1 Direct Mapping

Direct mappings link static graphical features directly to meta model concepts. For example, consider the *EPackage* element of the Ecore meta model. We can provide the graphical features for the concept as illustrated in Figure 4.9.

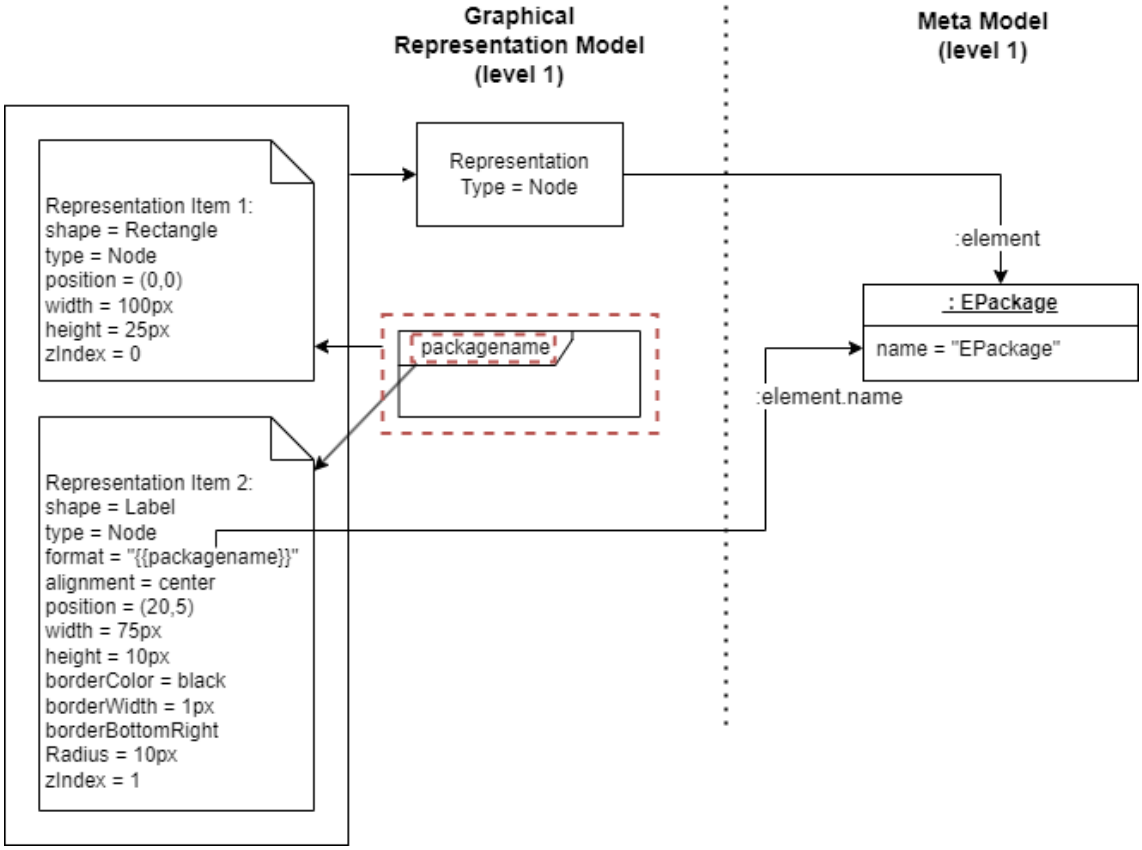


Figure 4.9: Mapping Graphical Features to EPackage Model Element

The graphical representation model provides a *Representation* consisting of two *RepresentationItems*: a rectangle shape and a label shape. The label shape is positioned in the upper-left corner of the rectangle shape, and it has a black border, with its bottom-right border cut off. A *format* style property defines the string value displayed within the label. Keywords enclosed in double-curly brackets within the format string reference attributes of the meta model element that the overall *Representation* refers to. In this case, *Representation* refers to the *EPackage* element.

At level 0, when creating diagram instances in concrete syntax in the diagram editor, every *Representation* will have its own position of where it is placed on the diagram canvas. All *RepresentationItems* also have their own positions. However, these are relative to the overall *Representation* and not relative to the diagram canvas.

4.5.2 Advanced Mapping

Advanced mappings handle dynamic visual behaviors. Consider the UML class diagram language as an example. For dynamic visualization of the attributes of a class, we can construct the mapping shown in Figure 4.10.

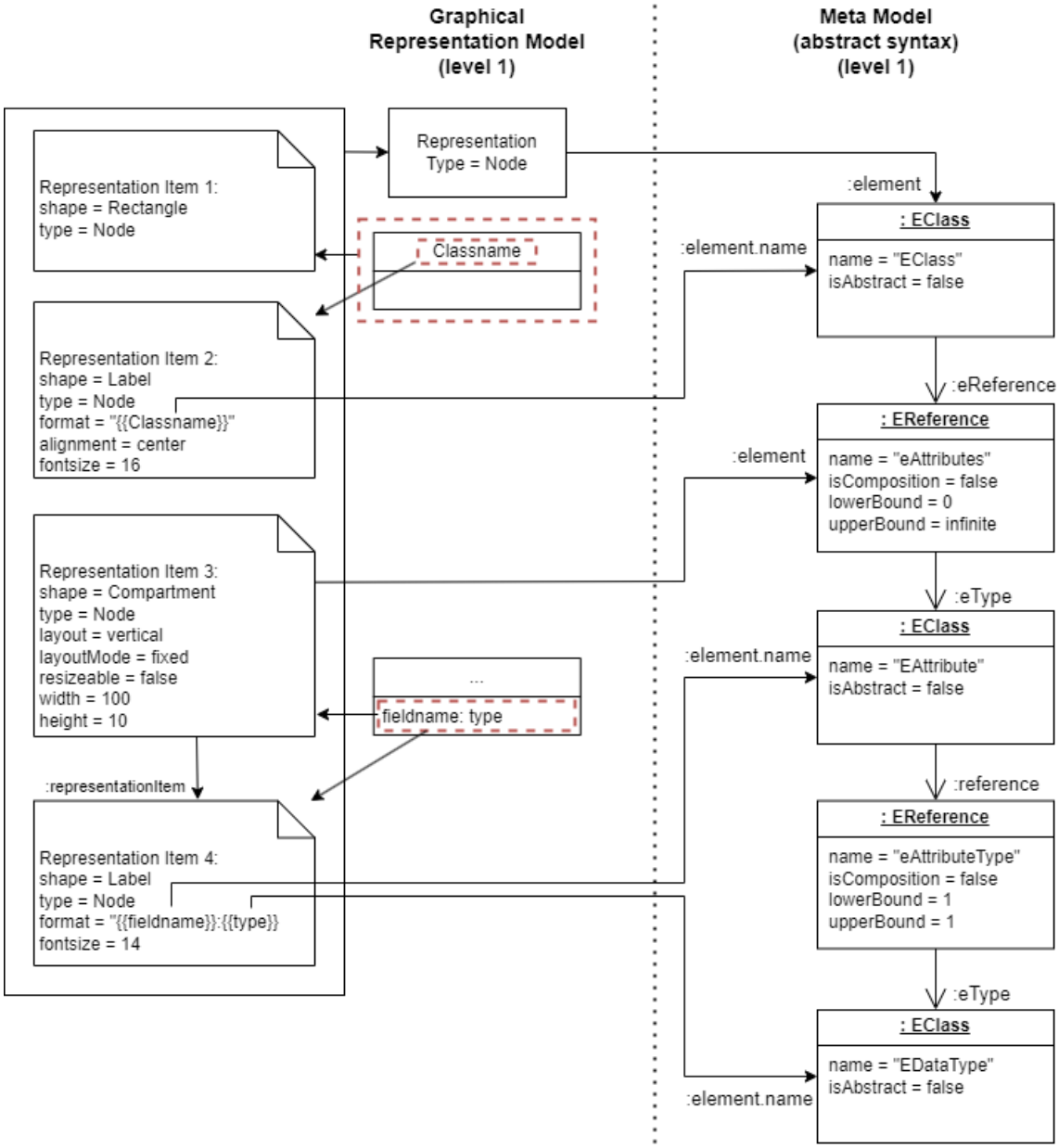


Figure 4.10: Mapping Graphical Features to Class

Representations items 1 and 2 provide the graphical features for the *EClass* concept similar to the direct mapping for *EPackage*. However, to visualize a class's attributes, a *Compartment* shape is used. The compartment is placed in the area that should display attributes. The compartment includes a *layout* style property, which specifies in which directions it can expand, either "vertical" or "horizontal". Additionally, the *layoutMode* "fixed" restricts the content of the compartment from being moved freely around.

The content of a compartment is determined by the *EReferences* and *Represen-*

tationItems that it references. In this case, the compartment references a label *RepresentationItem* that has a *format* of ”fieldname:type”. These placeholders correspond to the names of the two *EClasses* that represent *EAttributes* and their respective *EDataTypes*.

The compartment shape can, for example, also be used in BPMN diagrams for swim lanes, as illustrated in Figure 4.11.

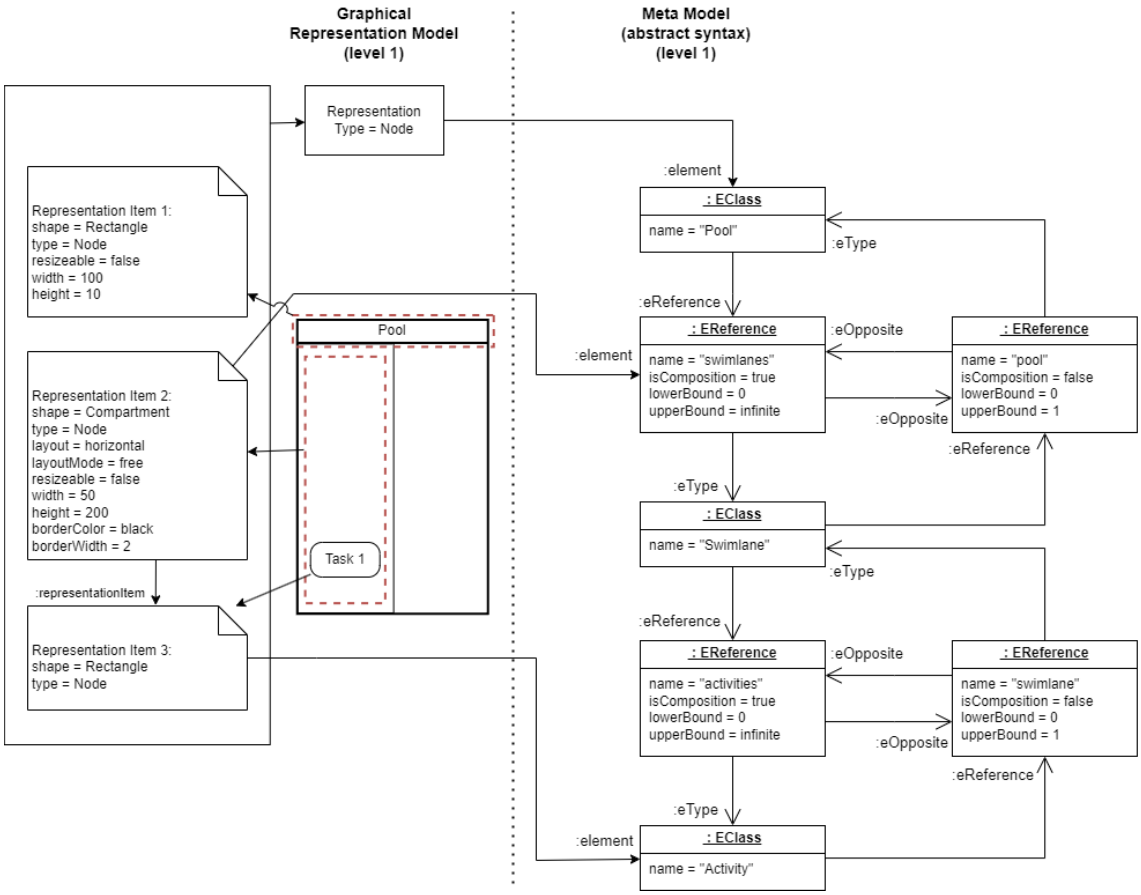


Figure 4.11: Mapping Graphical Features to Swimlane

In this example, the compartment’s *layout* is horizontal, and its *layoutMode* is ”free”, allowing representation items positioned inside the compartment to be freely moved without being restricted to a designated location.

For graphical representation of edges, such as associations in class diagrams, mappings can be created as shown in Figure 4.12.

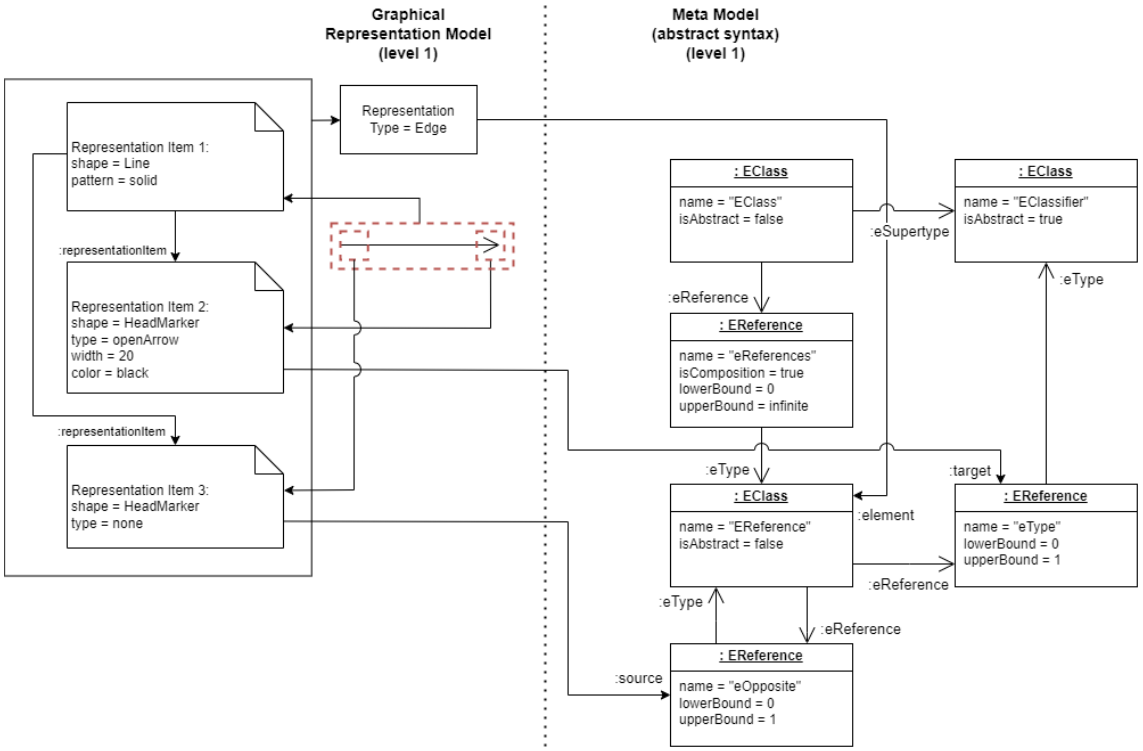


Figure 4.12: Mapping Graphical Features to Reference

The association representation consists of a *Line* shape with a *pattern* style property defining it as solid. The line includes two *HeadMarker* shapes, that represent arrowheads at the ends of the edge. The *source* and *target* property of each HeadMarker indicates whether the marker appears at the beginning or end of the edge. The appearance of the arrowhead is determined by the *type* property. For example, one end may have an ”open arrow” type while the other end has ”none”. Additional types, such as ”closed arrow”, ”composition” or ”aggregation” can also be specified.

Edges can also have other features like labels, multiplicities, line jumps, and paths, as shown in Figure 4.13.

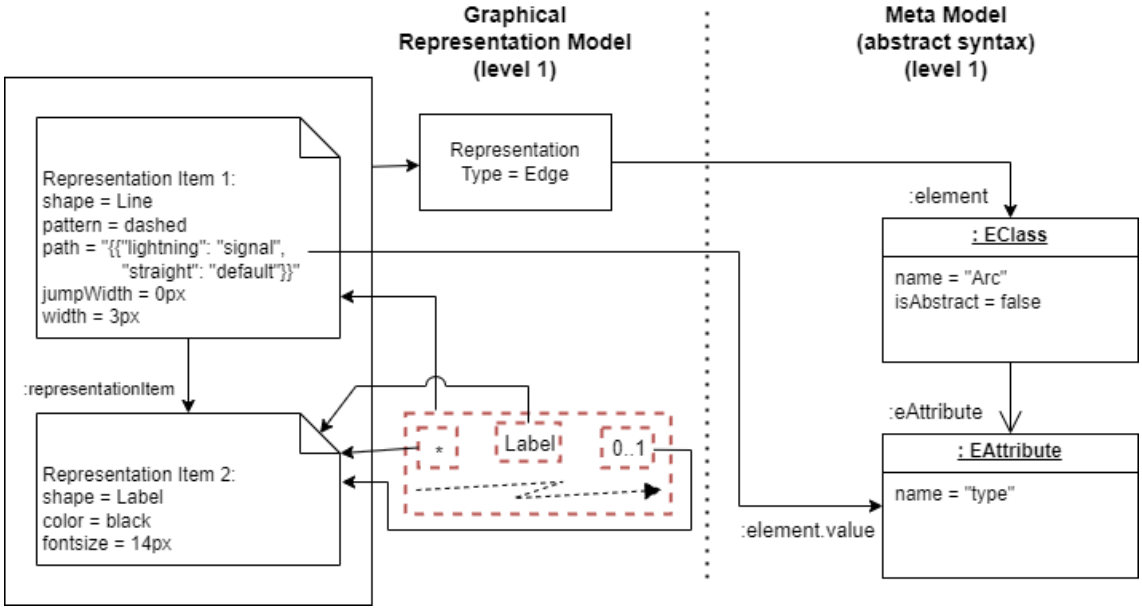


Figure 4.13: Mapping Additional Graphical Features to Reference

For example, style properties can depend on attribute values from the meta model concept. A mapping string such as ”{{”lightning”: ”signal”, ”straight”: ”default”}}” can specify that if the *type* attribute of an edge matches ”signal”, the line path will appear as ”lightning”. Otherwise, the default path (in this case, ”straight”) is applied. Other line paths may include orthogonal and curved edges. The line shape is linked to labels that define the default text displayed on an edge, such as its multiplicities or other attributes.

The main difference between direct mappings and advanced mappings is that direct mappings associate graphical features directly with meta model concepts, whereas advanced mappings involve references between meta model concepts to represent dynamic features.

Now that we understand how to map graphical features to meta model concepts, we can construct a graphical representation model to structure this mapping information into classes. By doing this, we can provide the information to our diagram editor in a structured way, allowing the diagram editor to have implemented mechanisms to render each *Representation* and its *RepresentationItems*.

4.6 Graphical Representation Model

The graphical representation model was created and is shown in Figure 4.14.

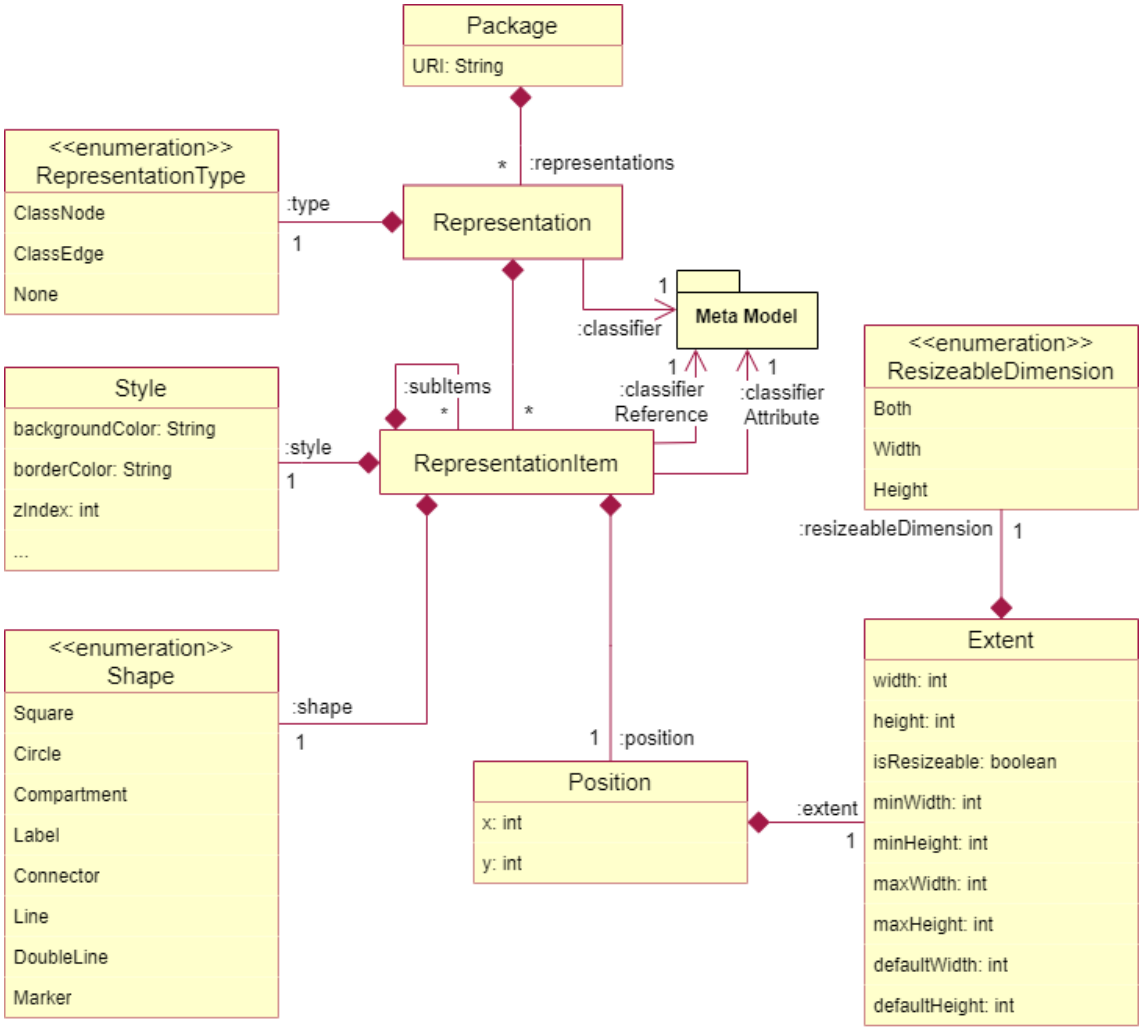


Figure 4.14: Graphical Representation Model

The topmost element in the representation model is the *Package*, which defines the URI (Uniformed Resource Identifier) specifying the location of the representation model. The *Package* contains multiple *Representation* objects. Each *Representation* points to the meta model element that it represents.

A *Representation* has multiple *RepresentationItem* objects, which define the various graphical layers of shapes that make up the *Representation*. For example, one representation could include a square shape and a label shape. The layering of shapes is controlled by the *zIndex* property within the *Style* object of each *RepresentationItem*. This property determines the stacking order of shapes. For example, to display a label shape on top of a square shape, the square shape would have a *zIndex* of 0, while the label shape would have a *zIndex* of 1.

The *Position* object specifies the x and y coordinates of the top left corner of a *RepresentatioItem*, while the *Extent* object defines the width and height of the shape relative to

this position.

To enable graphical representation of nested concepts, such as a Petri Net token inside a place, compartment shapes are used. This is achieved by allowing a *RepresentationItem* to reference other *RepresentationItems*, and applying the advanced mapping mechanisms presented in Section 4.5. The *classifierReference* and *classifierAttribute* are references that are used in these advanced mapping mechanisms. They point to an *EReference* and an *EAttribute*, respectively, of the *EClass* that the *Representation* points to.

4.7 Custom Constraint Language for Diagram Validation

To ensure that diagram instance models created within the universal diagram editor are valid, a custom constraint language is introduced. This language enables the definition of modeling rules that can be forced during the diagram creation process. The constraints are categorized into two types: life constraints and offline constraints.

4.7.1 Life Constraints

Life constraints are hard rules that prevent users from performing actions that would violate a constraint. For example, in Petri Nets, an arc can only connect a place to a transition or a transition to a place. The diagram editor would not allow users to draw an arc between two places or two transitions.

4.7.2 Offline Constraints

Offline constraints are softer rules that do not immediately restrict user actions. Instead, they allow the user to perform the action but mark the resulting diagram as invalid. For example, in Petri Nets, a place should not have more than a defined maximum number of tokens. The editor would allow users to add more tokens than allowed but display a warning indicating the violation.

4.7.3 Defining our Language

The custom constraint language allows graphical notation developers to define constraints by specifying:

1. **Concept:** The meta model element to which the constraint applies (e.g., Arc or Place).
2. **Condition:** The logical rule that defines validity.
3. **Type:** Whether the constraint is a life or an offline constraint.

The diagram editor evaluates constraints based on their type. Life constraints are forced during user interactions, giving immediate feedback, while offline constraints are evaluated on specific events, for example, after an action, on saving, or manually triggering the validation.

We can construct a language syntax similar to OCL with support for:

1. **Static Methods:** Methods like *kindOf* to evaluate type comparison.

2. **Attributes and Relationships:** Expressions like *self.source* and *self.target* to access properties or references in the level 0 abstract syntax model.
3. **Comparison Operators:** Standard operators such as *and*, *or*, ==, and >.
4. **Context:** Constraints are evaluated in the context of the *self* object.

Combining all the aspects, we can construct a constraint on the *Arc* concept with our language as seen in Listing 4.1.

```
condition: (self.source.kindOf('Place') and
            self.target.kindOf('Transition')) or
           (self.source.kindOf('Transition') and
            self.target.kindOf('Place'))
type: Life
```

Listing 4.1: Example OCL Constraint with Our Language

The diagram editor would access the type of constraint and evaluate its condition where appropriate. However, we need mechanisms for reading the condition string. Here, we can use an Abstract Syntax Tree (AST) approach. The AST represents the syntactic structure of the condition and enables modular evaluation.

4.7.4 Reading our Language

The AST for the condition presented in Listing 4.1 would be structured into the AST seen in Listing 4.2.

```
OR
  AND
     CALL kindOf(source, 'Place')
     CALL kindOf(target, 'Transition')
  AND
      CALL kindOf(source, 'Transition')
      CALL kindOf(target, 'Place')
```

Listing 4.2: AST of Condition String in the Example OCL Constraint

These elements of an AST are usually structured into nodes. The nodes that can be identified in our AST are:

- **BinaryExpression:** Represents logical (*and*, *or*) and comparison *(==, >)* operators.
- **CallExpression:** Represents method calls like *kindOf*.
- **Identifier:** Represents variables like *self.source* or *self.target*.
- **Literal:** Represents constant values like *Place* or *Transition*.

4.7.5 Evaluation Process

When performing the actual evaluation, we need to provide a context when evaluating identifiers. We define our context with a pointer to the *self*, *source*, and *target* diagram instance model elements. Each node in the AST is evaluated recursively. For static methods, we need to have their logic implemented in the diagram editor before being

able to use them. The *kindOf* method would, for example, check the *source* or *target* object against the meta model.

4.8 Conclusion

This analysis identified key elements necessary for designing the universal diagram editor. The key outcomes are summarized below:

- **Mapping Various Graphical Features to Diagram Concepts:** By analyzing modeling languages such as UML, BPMN, and Petri Nets, this chapter identified various graphical features. These features were mapped to generic diagram concepts, enabling their graphical representation in diagram editors.
- **Models in Meta Levels:** The Ecore model was chosen as the core meta model at level 2 due to its compatibility with UML and its scalable structure. The hierarchical structure of meta levels (levels 2, 1, and 0) was defined, showing how higher-level meta models can be instantiated into lower-level models.
- **Graphical Representation Model and its Mapping to Meta Model Concepts:** A graphical representation model was proposed to link meta model elements to their appearance. Direct mappings were used for static graphical features, while advanced mappings addressed dynamic behaviors such as compartments. This model ensures that the graphical representations of notations are well structured and mappable in the editor.
- **Integration of a Constraint Language:** A custom constraint language was introduced to validate diagram models. Life constraints provide immediate feedback during diagram creation to prevent invalid actions, while offline constraints allow actions but flag violations for correction.

5 Requirements

This chapter uses the findings from the conceptual analysis to determine and prioritize the requirements for the universal diagram editor. By identifying essential features of the system, this chapter sets the foundation for its design and implementation. Requirements are categorized into functional and non-functional requirements and are prioritized using the MoSCoW model (Must have, Should have, Could have, Won't have).

5.1 Scope

The scope of this thesis centers around creating a universal diagram editor that enables graphical notation developers to realize graphical notations easily and dynamically without manual code generation. The editor aims to support graph-related modeling languages such as Petri Nets, UML Class Diagrams, and BPMN.

As there are countless graphical features in these languages, the focus will not be on supporting them all. Instead, core graphical features are chosen from those identified in Section 4.1.

While lightweight validation and rule enforcement are implemented, the focus is mainly on demonstrating that meta models can be better applied than they are in current tools to achieve easily configurable diagram editors.

5.2 Functional requirements

Functional requirements focus on the features of a system. The identified functional requirements were derived from the conceptual analysis and are shown in Table 5.1.

ID	Description	Prioritization
R01	The system must allow graphical notation developers to define the graphical appearance of graphical notations	Must have
R02	The system must provide a way to define and modify the meta model of graphical notations using a user-friendly UI	Must have
R03	The system must provide a way to define the mappings between meta model elements and their graphical representation in a user-friendly UI	Must have
R04	The system must provide a diagram editor that is dynamic, meaning that whenever graphical notations are configured, they must be directly reflected in the diagram editor without performing any additional steps such as generating diagram editor code	Must have
R05	The system should support defining custom constraints on graphical notations using a constraint language	Should have
R06	The system should provide real-time validation of diagrams against defined constraints to ensure validity	Should have

ID	Description	Prioritization
R07	The system could allow exporting of the created diagrams in a picture format	Could have

Table 5.1: Functional requirements

The rationale behind the identified functional requirements are:

1. **Graphical Appearance Definition (R01):** As identified in Section 4.5, graphical notations consist of multiple graphical features. The system must enable graphical notation developers to define these features easily, including the definition of dynamic behaviors, such as the addition/removal of attributes in class diagrams.
2. **Meta Model Definition (R02):** As identified in Section 4.3, the meta model represents a modeling language. The Ecore model must be used as the core language for graphical notation developers to define meta models of modeling languages.
3. **Mapping Model (R03):** From Section 4.5, it was identified that a mapping model is a must to link concepts to their appearance. The system should, therefore, enable graphical notation developers to apply graphical features to meta model concepts.
4. **Dynamic Diagram Editor (R04):** Looking at Eclipse's EMF and GMF framework in Section 3.1.1, it was identified that whenever a graphical notation had its meta model or graphical representation changed, a new codebase for a diagram editor had to be generated, compiled, and run. As the universal diagram editor focuses on easing and optimizing the process of configuring graphical notations, this static procedure must be dynamic.
5. **Custom Constraint Language (R05):** Section 4.7 analyzed the need to force rules within diagrams to ensure valid modeling. While a constraint language is important, it is an additional layer on top of meta models, and therefore, it will not be the main focus of this thesis; hence, the requirement is prioritized as a ”Should have”.
6. **Real-Time Validation (R06):** As identified in 4.7, real-time validation against defined constraints ensures the validity of diagrams during their creation. However, as this thesis focuses on the application of meta models, it has been prioritized as a ”Should have”.
7. **Exporting of Diagram (R07):** Diagram creators could have the possibility of exporting the concrete syntax of created diagrams in a picture format such as JPEG.

5.3 Non-functional Requirements

Non-functional requirements address the performance, usability, security, and scalability of a system. The rationale behind the identified non-functional requirements was primarily by performing user tests on existing diagram editors presented in 3.1. Table 5.2 shows the identified non-functional requirements for the universal diagram editor.

ID	Description	Prioritization
R01	The diagram editor should be intuitive and user-friendly to use	Should have
R02	The system must be web-based without the need for installation	Must have
R03	The system should ensure performance efficiency, including smooth interaction with diagrams	Should have
R04	The graphical modeling languages that graphical notation developers define should be secure to use in the diagram editor	Should have
R05	The system could support collaboration features, such as sharing and concurrent creation of diagrams	Could have
RO6	The system should be able to have multiple graphical modeling languages defined	Should have

Table 5.2: Non-functional requirements

6 Conceptual Design

This chapter builds on the conceptual analysis and identified requirements to outline the design of the universal diagram editor. The chapter defines the conceptual architecture and key components that guide the software design.

6.1 Concept Overview

We start by identifying the different components of the concept that the universal diagram editor should contain. The components can be seen in Figure 6.1.

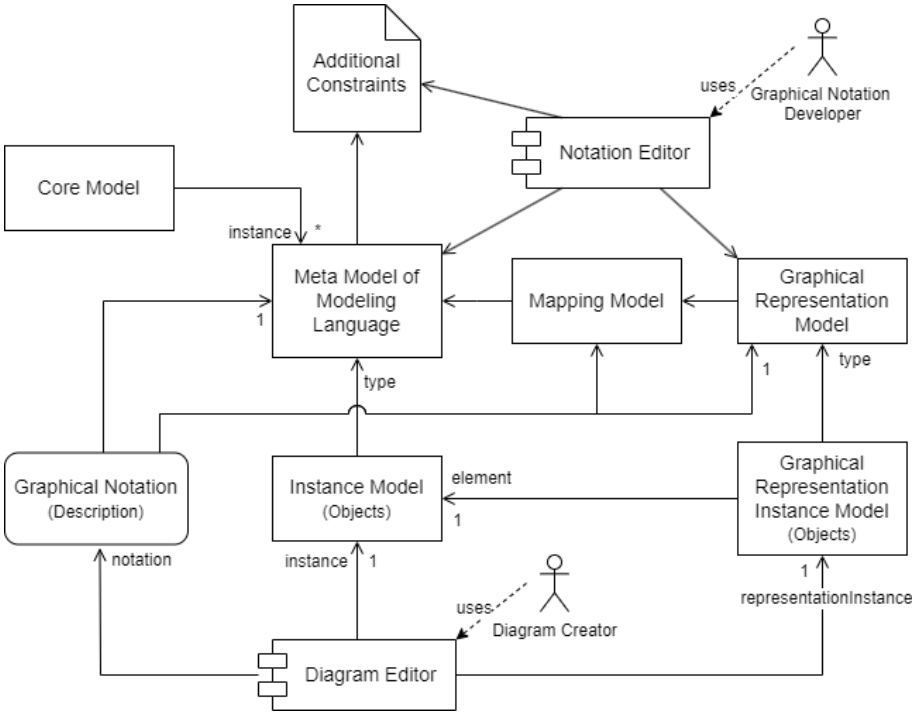


Figure 6.1: Concept Architecture

The core model is used to define the meta model of a graphical notation. A meta model can include additional constraints to enforce specific validation rules for diagrams. The graphical representation model defines how elements of the meta model appear. This is done by mapping each element of the graphical representation model to an element of the meta model through a mapping model.

A graphical notation consists of a meta model, a representation model, and a mapping model that defines how elements between these are linked. The diagram editor uses this graphical notation to initialize and manage instance models. In this context, a diagram is represented by:

- The *Instance Model* in abstract syntax, which defines the structure of the diagram, consisting of objects and their relationships, derived from the meta model of the modeling language.

- The *Graphical Representation Instance Model*, in abstract syntax, defines how these objects and relationships are graphically represented, including their shapes, styles, and positions, indicating where on the diagram canvas they appear.

Together, these two models constitute a diagram. The diagram editor validates diagrams by comparing the structure of the instance model to the additional constraints of the meta model, ensuring that the diagram adheres to the defined rules of the notation.

The key concepts in the universal diagram editor are:

- **Core Model:** The meta meta model is used to define meta models that represent modeling languages. A meta meta model was seen in Figure 4.6, where we demonstrated how to use the Ecore model to instantiate meta models.
- **Meta Model:** Defines the concepts, i.e., the abstract syntax of a modeling language, which was presented in Section 4.4.1.
- **Graphical Representation Model:** Manages the layers of graphical features of meta model concepts. The graphical representation model that will be used was presented in Section 4.6.
- **Mapping Model:** Maps graphical representations to meta model concepts. A proposed mapping model was introduced in Section 4.5, and it is part of the graphical representation model.
- **Dynamic Diagram Editor:** Reflects configurations of graphical notations directly in the diagram editor without it depending on any manual process, including, for example, code generation. This concept will be presented in the software design chapter, in Section 7.3.
- **Constraint Language:** Used to define additional constraints that the diagram editor can interpret to validate created diagrams. In Section 4.7, a custom constraint language was proposed.

We will now reiterate some of these individual key concepts and define how they should be designed to fit the purpose of our universal diagram editor.

6.2 Core Model

The Ecore model from Eclipse provides a language for meta modeling the modeling languages in the universal diagram editor. However, to align with the specific requirements and scope of this thesis, the Ecore model was simplified and scoped down.

Figure 6.2 presents the simplified Ecore model. This simplified model serves as the core model for the system.

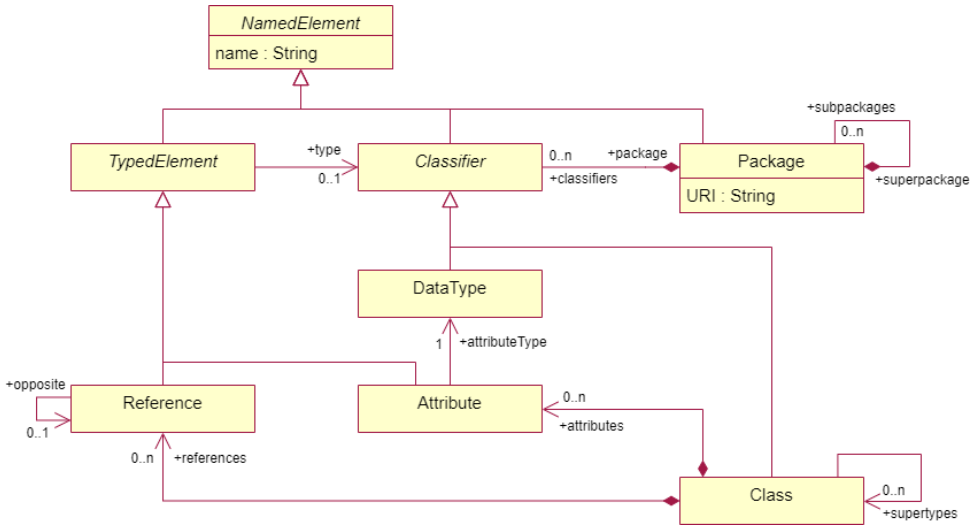


Figure 6.2: Simplified Ecore Model

In this simplified model, certain elements have been scoped out to align with the thesis's specific needs and scope defined in Section 5.1, where the main focus is on demonstrating that *meta models can be better applied to achieve easily configurable diagram editors*.

First, a minor modification was made to the overall model. The 'E' naming prefix on the model elements and data types has been omitted to signal that it is a modified version and not directly the EMF's Ecore model.

In the Ecore model, there are two layers of *EModelElements*: *ENamedElements* and *EAnnotations*. *ENamedElements* are the core modeling elements, and *EAnnotations* are modeling elements describing *ENamedElements*. An *EAnnotation* can, in graphical terms, for example, have the appearance of a note containing a description. However, the annotation layer of diagrams does not contribute to the focus of this thesis, and therefore, the *EModelElement* and *EAnnotation* were omitted.

The simplified Ecore model retains its fundamental elements, such as *Class*, *Attributes*, and *References*. *EOperation* and *EParameter* were also omitted due to the scope of the thesis. However, it is assumed that principles applied to *Attributes* and *References* are also usable when integrating *EOperations* and *EParameters*. The "attributeType" reference from *Attribute* to *DataType* is represented by the reference "type" of *TypedElement*, which *Attribute* inherits from. *Attributes* can have "type" references to both *Datatypes* and *Classes*, which our simplified model reflects. *References*, however, should only have "type" references to *Classes*, as they should not point to any *DataTypes*. To ensure this, we provide an additional constraint as seen in Listing 6.1. The constraint is formulated using our custom constraint language.

```
context Reference inv:
  self.type.isKindOf(Class)
```

Listing 6.1: Additional Constraint for Reference

Another constraint that should be provided to the *Reference* concept is its ”opposite” reference that points to another *Reference*. Here, we should define that the ”opposite” reference can only point to other concepts of type *Reference*. The constraint is presented in Listing 6.2.

```
context Reference inv:
  self.opposite.isKindOf(Reference)
```

Listing 6.2: Additional Constraint for Reference

The *Package* concept is, of course, also kept, as it is the element that stores all *Classes*. The *EPackage* in Ecore does have a URI attribute, but it is abstracted away in their Ecore model. In our simplified model, the URI attribute is shown for simplicity. The URI specifies where the meta model is located. The *DataType* in the universal diagram editor will initially only support the built-in data types: ”String”, ”boolean” and ”int”. To reflect the ”*” (many) relationship boundary in references another built-in type is introduced, ”infinite”.

To use the simplified core model in the software, models are serialized into JSON files that structure elements in objects. To ensure proper object structure, an abstract syntax model of the simplified core model was created using an object diagram. The resulting object diagram for the simplified ecore model is shown in Figure 6.3.

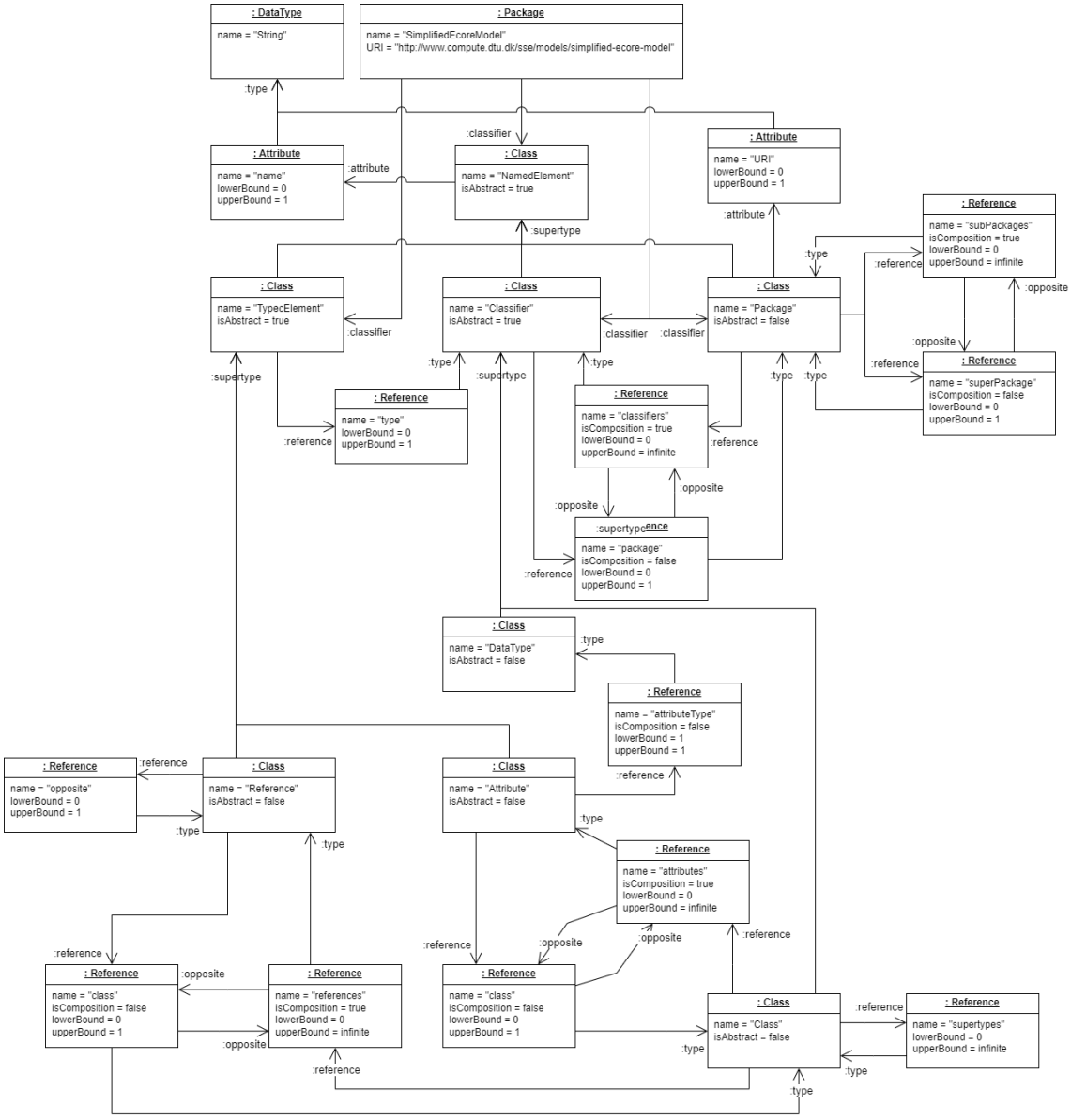


Figure 6.3: Abstract Syntax Model of Simplified Ecore Model as an Object Diagram

Having finalized our core model, we will explore its application in creating meta models in abstract syntax format.

6.3 Meta Model for Abstract Syntax

The creation of meta models (modeling languages) using the Ecore model was demonstrated in the analysis chapter 4.3, where a meta model of the Petri Net modeling language was constructed. That Petri Net meta model can also still be viewed as an instance of the simplified core model, as it does not include any *EOperations* or *EParameters* inside the *EClasses*, nor does it include any *EAnnotations*. There is no major difference in the creation of a meta model when using the simplified core model compared to the original Ecore model already presented. However, to understand the next part of this section, the Petri Net Meta Model is shown again in Figure 6.4.

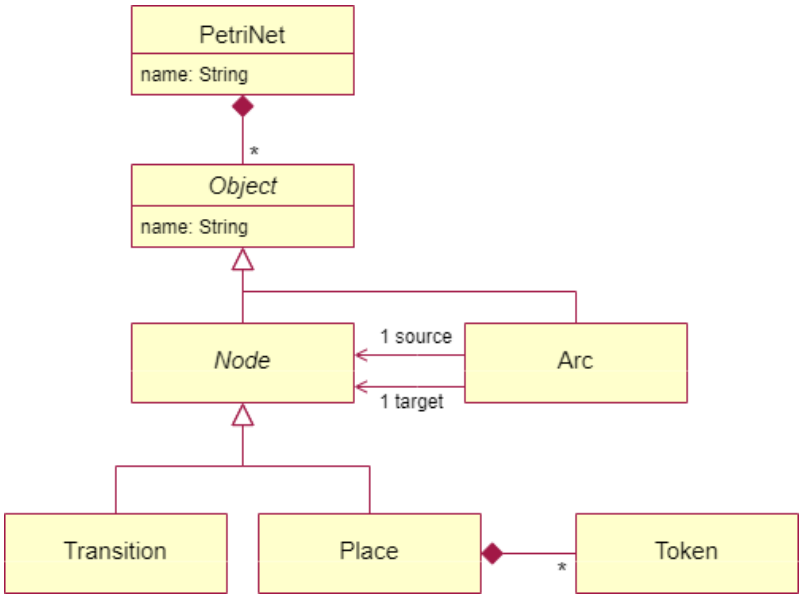


Figure 6.4: Concrete Syntax Model of the Petri Net Meta Model

This section will focus on preparing a meta model structure in abstract syntax (object diagram form). The meta model is derived using the simplified core model and it models the same Petri Net meta model as seen in Figure 6.4, but is represented in abstract syntax rather than concrete syntax. The meta model as an object diagram can be seen in Figure 6.5.

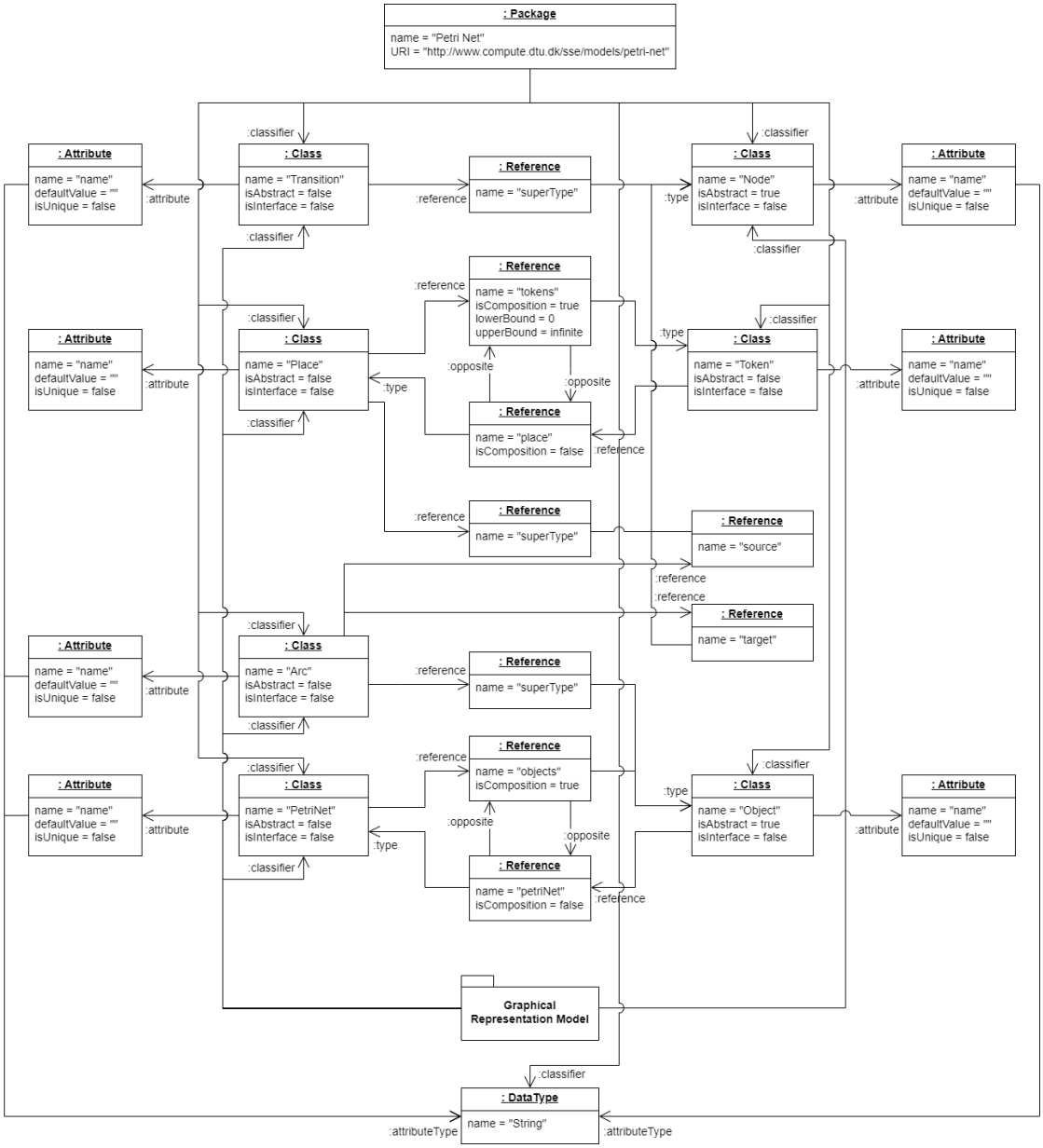


Figure 6.5: Abstract Syntax Model of the Petri Net Meta Model as an Object Diagram

As seen in Figure 6.5, meta models can become verbose when transformed into object diagram form, even for relatively simple examples like the Petri Net meta model. That is why we have the concrete syntax to draw the abstract syntax instead.

Let us briefly review the model. First, a model will always have one top-most package that holds all the model elements; this package has the name of the modeling language, which in this case is Petri Net. The top-most package holds the URI, specifying where

the meta model is located. The classes are represented as objects of type *Class* having default *Class* attributes such as ”isAbstract” given by the underlying structure of the core model. The classes reference other objects of type *Attribute* when referring to the attributes of the classes themselves. In this case, all classes have an attribute with the name ”name” and its datatype ”String”. Additionally, objects of type *Class* that require a graphical representation are referenced by an element in the graphical representation model through the ”classifier” property. As for the rest, the same transforming mechanisms applies as demonstrated in the analysis chapter, in Section 4.4.

The quality of the core model lies in its ability to represent not only all the concepts of the Petri Net meta model but also those of any other modeling languages, including the ones introduced earlier. The core model is even able to model itself, as it is a meta model for class diagrams, and it is already known that the core model itself is already represented as a class diagram. This flexible ability of the core model ensures that it is capable of supporting a wide range of modeling languages, which makes it a solid meta meta model for the universal diagram editor.

Now that we know how to structure our meta models in object format readable by our software, we will look at how our graphical representation model can be structured in a similar way.

6.4 Graphical Representation Model

To maintain a clear separation between the meta model concepts and their appearance, the graphical representation model was separated from the meta model. This design choice follows the separation of concerns principle, which simplifies error identification. For example, if an issue is spotted in the appearance of a graphical notation, it is known that the error is most likely located in the representation model, and oppositely if a concept does not behave as designed, the error is most likely in the meta model.

The graphical representation model, shown in Figure 6.6, was presented in the analysis chapter in Section 4.6. However, it is shown again for easier readability when comparing the abstract syntax graphical representation model for Petri Nets in Figure 6.7.

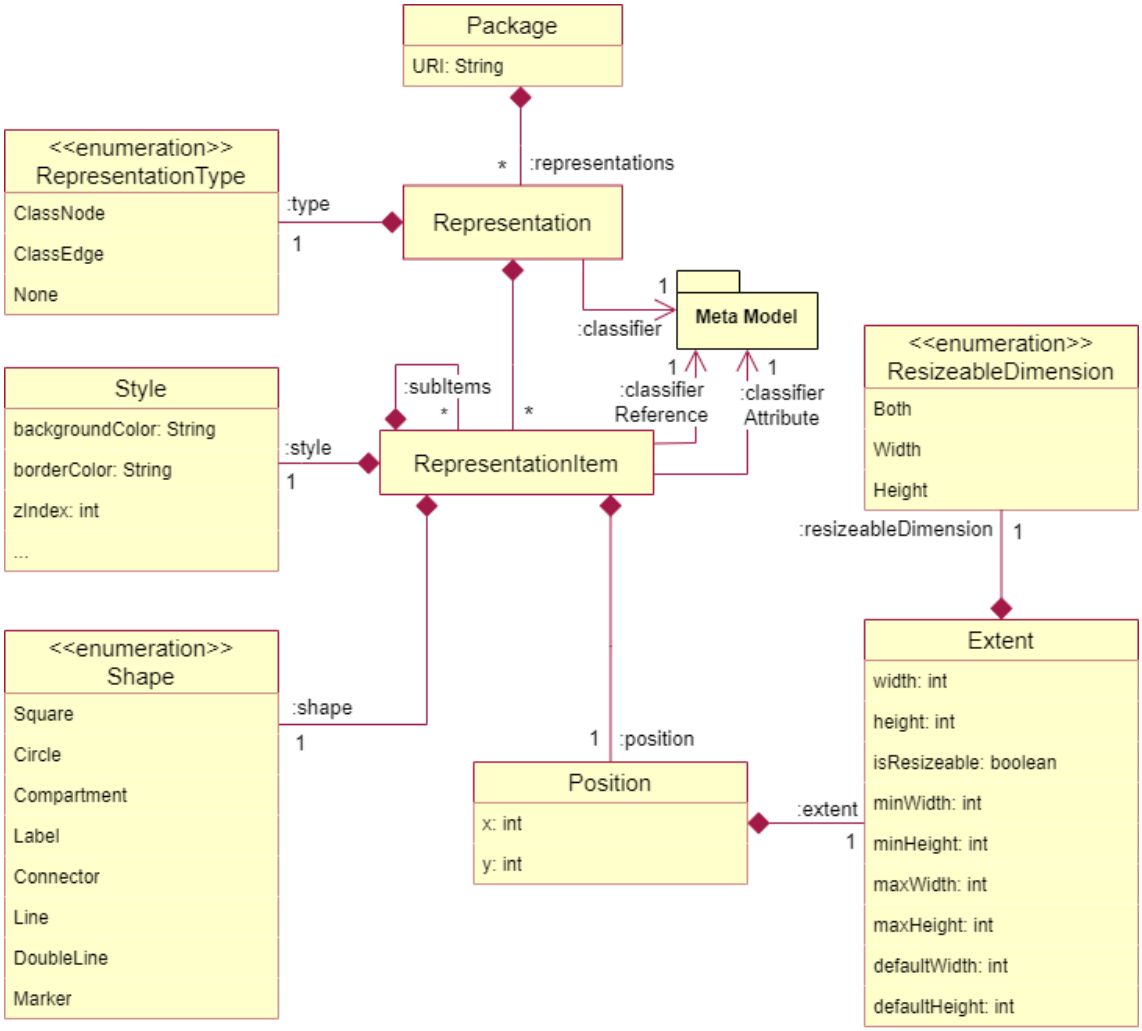


Figure 6.6: Graphical Representation Model

An abstract syntax model is created in object format for the representation model of Petri Nets, and it can be seen in Figure 6.7. The model is derived from our graphical representation model seen in Figure 6.6 together with the Petri Net graphical features identified in the analysis chapter, in Section 4.3. For the model to not become verbose, only the full representation of the *Transition* concept was modeled, which can be seen on the left-hand side of the model. However, the same principles can be applied to the rest of the concepts that require a visual representation, i.e., Place, Token, and Arc.

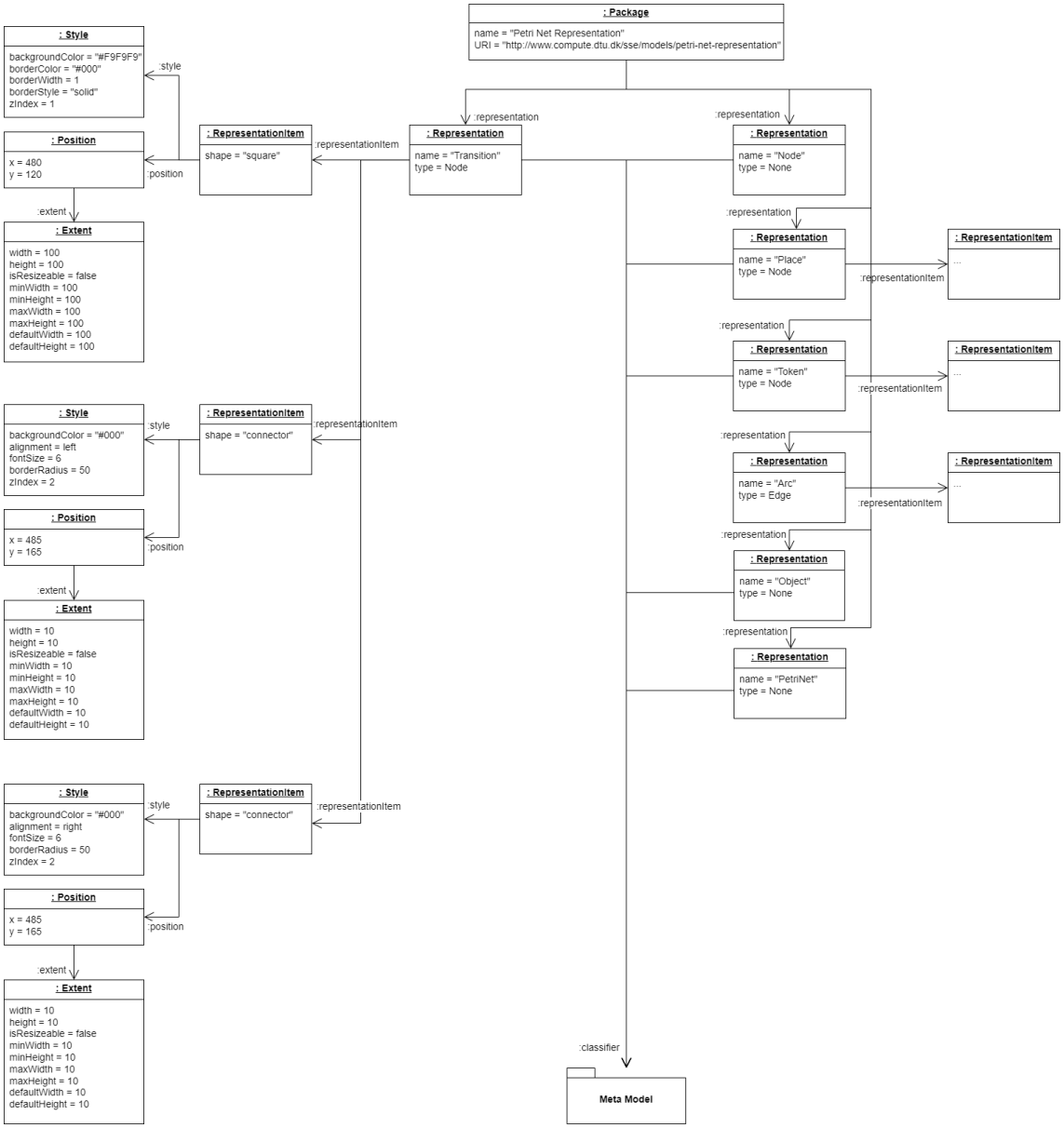


Figure 6.7: Abstract Syntax Model of the Graphical Representation Model for Petri Nets as an Object Diagram

The representation model consists of a top-most package. The package holds multiple *Representation* objects, where each *Representation* points to a meta model element. Furthermore, every *Representation* object consists of layers of shapes called *RepresentationItems*. Each *RepresentationItem* consists of the objects *Style* and *Position*. *Style* specifies how a *RepresentationItem* should appear graphically, together with a *zIndex*

attribute specifying what layer the *RepresentationItem* should be in. *Position* specifies where the top left point of a *RepresentationItem* should be positioned relative to the whole *Representation*. The *Position* also consists of an *Extent* object, which specifies the *width* and *height* of a *RepresentationItem*. The *Extent* object also specifies if the *RepresentationItem* is resizable. If so, the minimum and maximum width and height are given to specify to what extent a *RepresentationItem* can be resized. In addition, a default width and height are given to decide its initial size.

We now have an abstract syntax model of the default representation that a graphical modeling language should have. However, this model serves only as the default representation of the modeling elements, but diagram creators should also be able to customize the representation to fit their needs. Therefore, we need another model, a graphical representation instance model specific to the diagram creator.

6.4.1 Representation Instance Model

When creating diagrams in concrete syntax, diagram creators would change the appearance of graphical notations at run-time. This could, for example, be changing the background color of a square or extending the base size's width and height. An instance model should be integrated to allow diagram creators to do so. Not only is an instance model necessary for having run-time graphical notations customized, but it is also necessary to keep track of where *Representations* are placed on the diagram canvas.

The instance model is essentially a copy of the default representation model but with some additional information regarding the *Representation's* position. The additional position information can be illustrated by extending the representation model as seen in Figure 6.8.

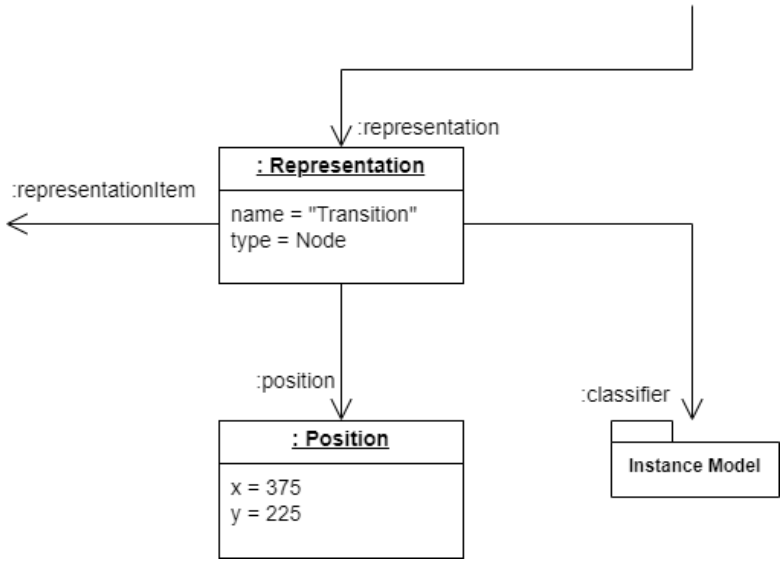


Figure 6.8: Additional Information in the Representation Instance Model

Now that we are located in the representation instance model, the *Representations* points to *classifiers* of the *instance model* instead of the *meta model*.

6.5 Mapping Model

The abstract syntax meta model presented in Figure 6.5 and its corresponding graphical representation model introduced in Figure 6.7, shows that elements in the representation model are associated with elements in the meta model. This association forms the mapping model, which is illustrated in a simplified form in Figure 6.9.

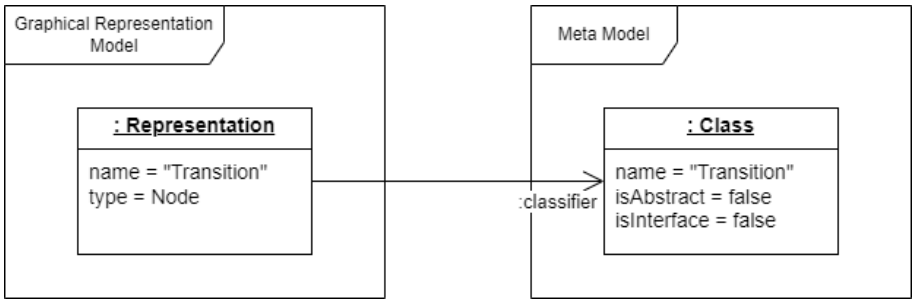


Figure 6.9: Mapping from Representation to Concept

The detailed mappings, including advanced graphical features, have already been explained in the analysis chapter, Section 4.5. Since the same mapping principles are applied in the implementation, this section does not revisit those details. Rather, an additional implementation aspect is introduced: the use of CSS (Cascading Style Sheets). CSS is used in the *Style* object that each *RepresentationItem* associates with. Incorporating CSS for styling aligns with modern web development, ensuring that the graphical representation model is easily understandable for future contributors.

6.6 React Flow

The universal diagram editor will use *React Flow* to implement its diagram editor. React Flow supports built-in concepts like *Nodes*, *Edges*, and *Handles*. [20]

- **Nodes:** A node can render anything, and it can be customized to look and act as we would like.
- **Edges:** An edge connects two nodes. Every edge needs a target and a source node. Edges can be customized to look as we would like.
- **Handles:** A handle is the place where an edge attaches to a node. The handle can be placed anywhere and styled as we would like.

These concepts align closely with the generic diagram concepts identified in Section 4.2, where *Nodes* and *Edges* were core concepts, and *Handles* correspond to *Connection Points* in our analysis.

6.7 Integrating Constraints in Core Model

To enable diagram validation, the custom constraint language introduced in Section 4.7 is integrated directly into the simplified core model.

By embedding constraints in the meta model, the validation process becomes simpler, as constraints are directly linked to their respective modeling concept, which eliminates the need to define a *Context* within the constraint like typically done in OCL.

The simplified core model is extended to include a *Constraint* class as shown in Figure 6.10. This class allows constraints to be directly associated with meta model concepts.

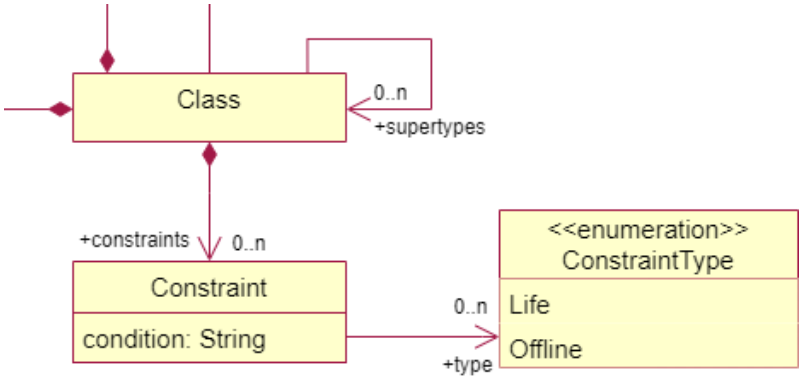


Figure 6.10: Simplified Ecore Model Extended With Constraints

The *Class* is extended to have a reference *constraints* to the *Constraint* class, enabling each *Class* to define multiple constraints. The *Constraint* class includes the *condition* attribute representing the logical rule for the constraint. *Constraint* also has a *Constraint-Type*, which specifies the type of the constraint: ”Life” and ”Offline”. Life constraints enforce rules during diagram creation. Offline constraints mark violations without enforcing them.

6.8 User Interface

This section describes the UI of the universal diagram editor, focusing on its two main pages: the *Graphical Notation Designer Page* and the *Diagram Editor Page*. Design considerations are also discussed.

6.8.1 Graphical Notation Designer Page

The purpose of the graphical notation designer page is to enable graphical notation developers to easily configure the different graphical modeling languages that the diagram editor should support. The page consists of two switchable panels: *Configure* and *Draw*.

Configure Panel
The *Configure* panel allows users to define the meta model of a graphical modeling language. The panel is shown in Figure 6.11.

Figure 6.11: Graphical Notation Designer Page (Configure Panel)

At the top of the panel, users can input a URI to create a new modeling language or select an existing URI to update an existing language. Users can also name the language or delete it entirely from the editor.

Meta model elements can be added or removed, and it can be specified whether these elements should be represented as nodes, edges or remain non-visualized. Attributes and references can also be added or modified to define the relationships and properties of meta model elements.

Draw Panel

The *Draw* panel provides an interface for designing the graphical representation of meta model elements. When designing nodes, users can select shapes from a palette and drag them onto a canvas to create appearances, as shown in Figure 6.12. For edges, users are restricted to static line-based representations as shown in Figure 6.13.

Labels and compartments are interactive elements within the draw panel. Double-clicking a label opens a popup modal, allowing developers to specify which meta model attribute the label should represent. Similarly, compartments can be configured to display references.

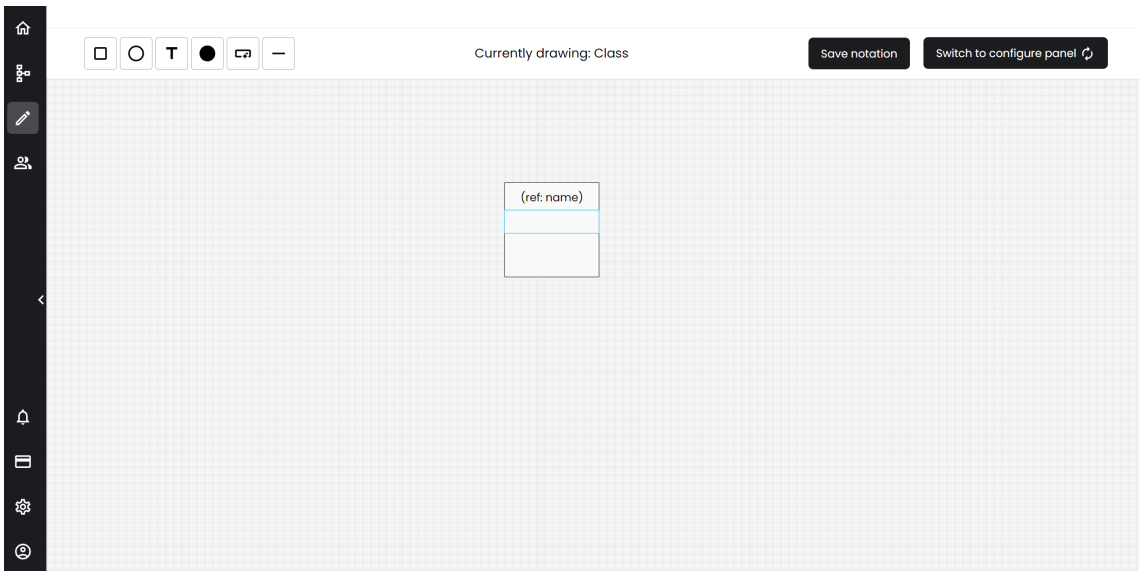


Figure 6.12: Graphical Notation Designer Page (Draw Panel - Node)

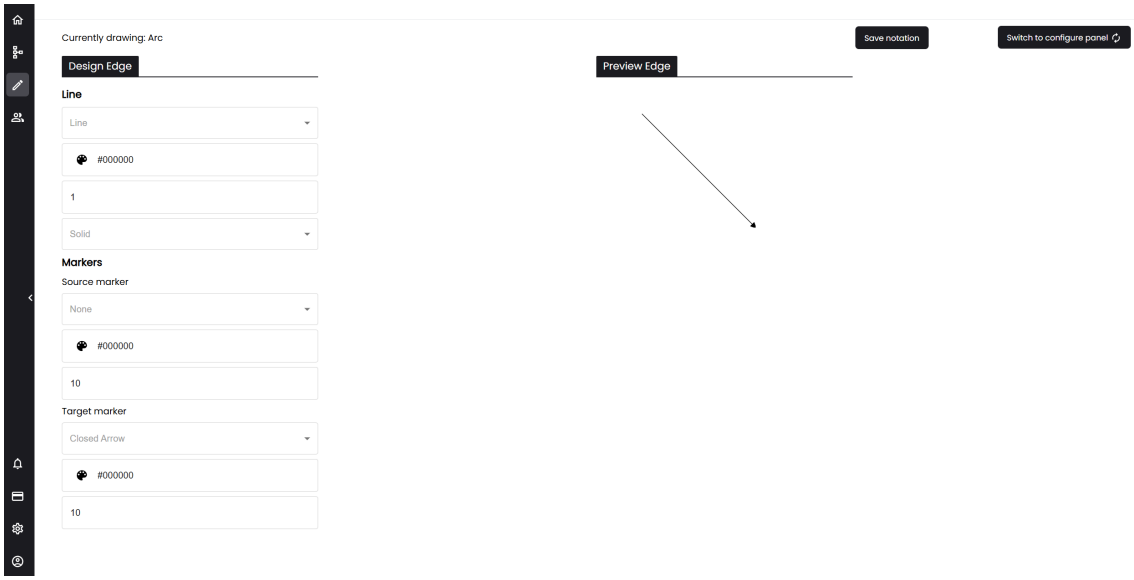


Figure 6.13: Graphical Notation Designer Page (Draw Panel - Edge)

The *Draw* panel ensures that graphical notation developers can create accurate representations for nodes and edges while maintaining an easy-to-use interface.

6.8.2 Diagram Editor Page

The diagram editor page is where diagram creators use the defined graphical modeling languages to create and edit diagrams. The page is shown in Figure 6.14. Users can select a predefined language from the ”Diagram type” dropdown in the top-left corner. Once a language is selected, its graphical modeling elements appear in the left-side palette, allowing users to drag them onto the canvas.

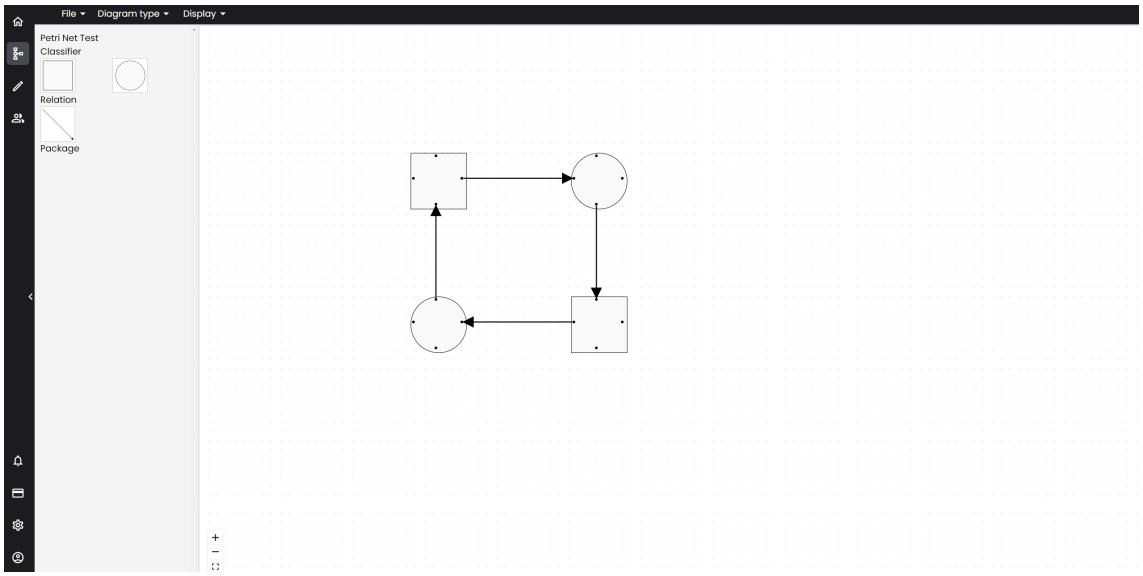


Figure 6.14: Diagram Editor Page

The diagram editor is interactive and ensures that any changes made to the concrete syntax instance model on the canvas are reflected in the underlying abstract syntax instance model. This synchronization allows the abstract syntax model to be saved and later reloaded, ensuring continuous editing.

6.8.3 Constraint Violation Feedback

When a constraint is violated during diagram creation, the editor provides feedback to guide the user. As shown in Figure 6.15, feedback messages are displayed at the bottom of the diagram canvas, making it clear which action caused the violation.

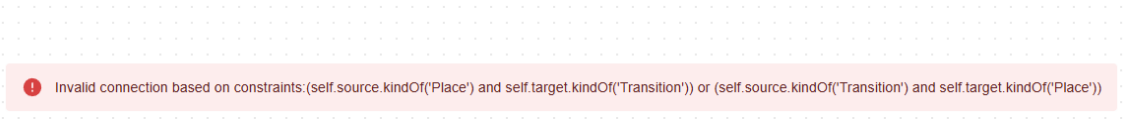


Figure 6.15: Constraint Violation Message

The editor is initially designed to enforce *life constraints*, which prevent invalid actions from being performed in real-time. These constraints are communicated through immediate, single-messages. Future design considerations could include support for *offline constraints*, enabling the system to perform a validation check and display a list of violations.

7 Software Design

This chapter focuses on converting the conceptual design into an implementable software architecture for the universal diagram editor. It outlines the technologies, tools, and frameworks chosen for implementation.

7.1 Technological Stack

The universal diagram editor is built by using different frameworks and libraries. Below is an overview of the technologies used for the different aspects of the system:

- **Frontend:**
  - **React with Typescript:** React is a modern Javascript frontend framework focused on its ability to split UI elements into components for easier scalability and maintainability. In addition to React, Typescript is used to check the type correctness of our Javascript code.
  - **React Flow:** React Flow is the framework used to implement the diagram editor. It provides custom rendering logic for its diagram nodes, edges, and handles.
- **Backend:**
  - **Node.js with Typescript:** Node.js is a popular Javascript backend framework enabling API development. Typescript is also used on top of Node.js for type-checking its Javascript code.
- **Storage:**
  - **File Storage:** To store the abstract syntax meta models of modeling languages, their objects are serialized into JSON files. These files are stored on the sever-side's file storage. This is because modeling languages must be provided to all users.
  - **Local Storage:** To store the abstract syntax instance models of created diagrams, they are also serialized into JSON files. However, these files are stored on the user's local storage. Local storage is a feature of the web browser that allows the solution to store key-value data on the client side. A library called Redux is used to manage local storage.

7.2 System Architecture

The system is divided into two main layers: the *Frontend* and the *Backend*, as shown in Figure 7.1.

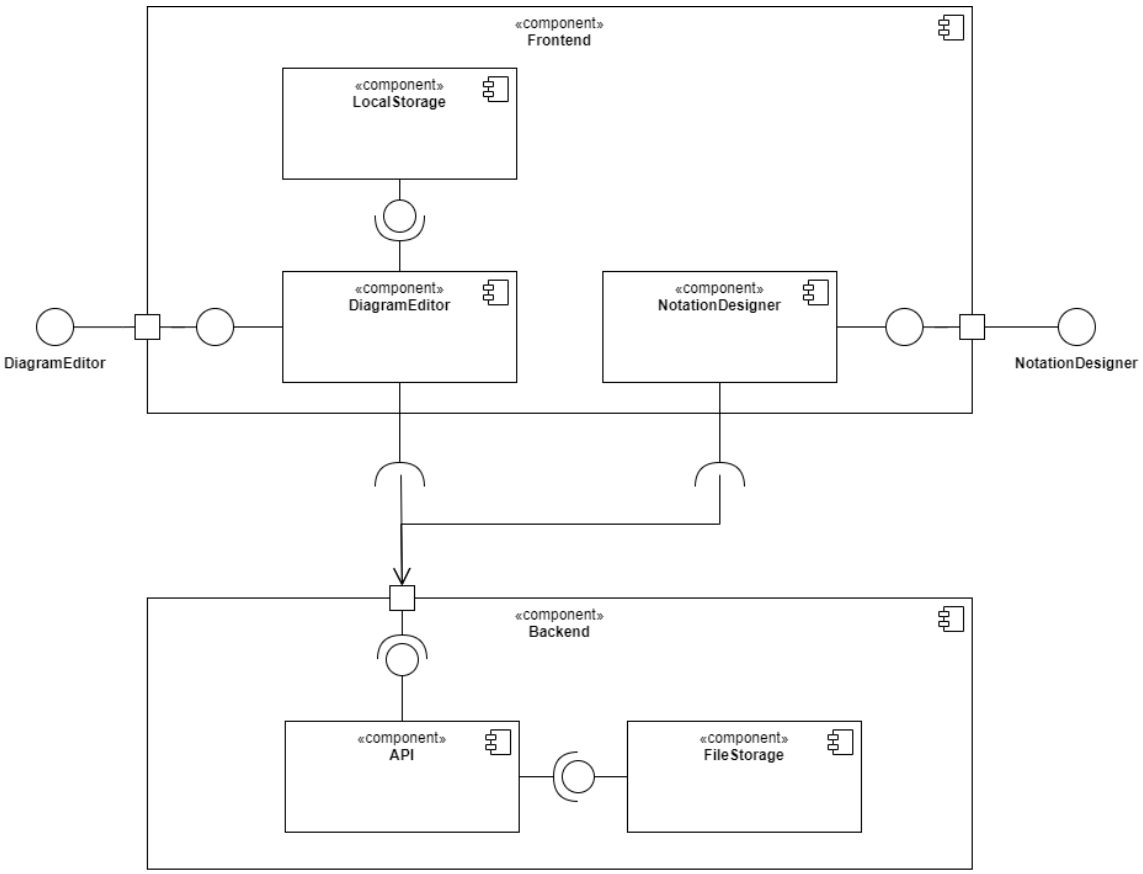


Figure 7.1: System Architecture of the Universal Diagram Editor

The *Frontend* includes the *DiagramEditor* component, which is provided to outside users through a port. This is the component that enables diagram editors to use predefined graphical modeling languages to create and edit diagrams. The *DiagramEditor* requires another component inside the *Frontend*, which is *LocalStorage*. The *LocalStorage* component has the responsibility of storing user initialized diagram models. Lastly, the *Frontend* includes the *NotationDesigner* component, which allows graphical notation developers to define graphical notations. The *DiagramEditor* and *NotationDesigner* communicate with the *API* component in the *Backend* for retrieving and saving meta models.

The *Backend* handles the server-side logic and data storage. It includes the *API* component, which processes requests from the *DiagramEditor* and *NotationDesigner* to save and retrieve data. This process relies on the *FileStorage* component, which is responsible for managing the storage of models as JSON files.

7.3 Dynamic Diagram Editor

To achieve a dynamic diagram editor, it must operate without requiring code generation whenever its modeling languages are reconfigured. This means that we want a static

codebase capable of dynamically loading and rendering graphical notations at run-time.

7.3.1 Loading of Notation Configurations

To enable run-time reloading of notations, the diagram editor is designed as a dedicated service decoupled from the meta models of the modeling languages. Whenever a diagram creator selects a language, the frontend service requests the latest configuration of that notation from the external backend service. This architecture ensures that we can reload notation configurations at run-time.

However, dynamically loading notation configurations is one part. The editor must also be able to interpret these configurations and visualize them as nodes or edges where appropriate.

7.3.2 Mapping Notations to React Flow Components

In the analysis chapter, Section 4.1, we established mappings between graphical features and their generic diagram concepts. Later, in the conceptual design chapter, Section 6.6, we identified that the same mappings can be applied conceptually to the React Flow components: *nodes* and *edges*. While the mappings themselves have already been detailed, the React Flow framework must be structured statically to render the graphical representation shapes of meta model concepts dynamically. To achieve this, the framework's components are designed to render graphical features as layered structures.

Node Component Design

The React Flow *Node* component is designed to render *Representations* through their individual *RepresentationItems*, as shown in Figure 7.2. The specific rendering mechanism follows the approach presented in the analysis chapter, Section 4.1.

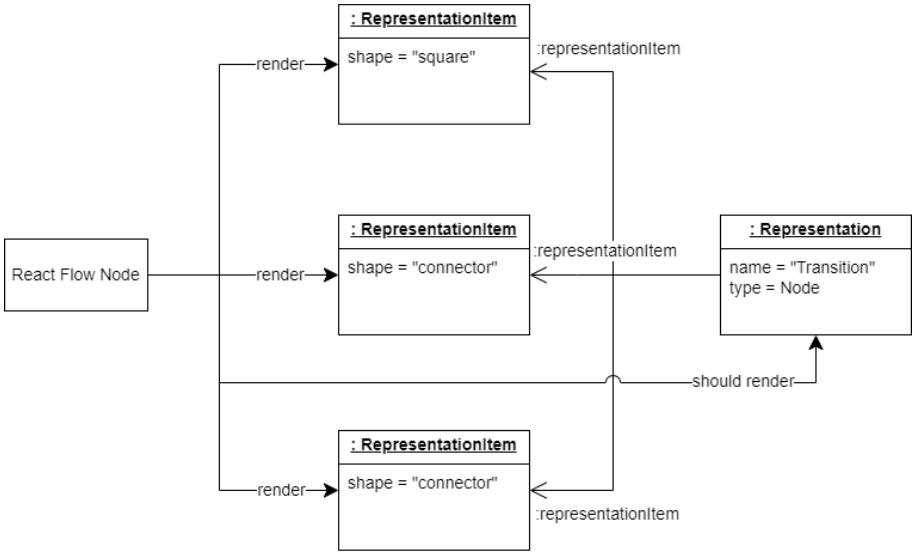


Figure 7.2: React Flow Node Mapping to Graphical Representation Model

As shown in the figure, the *Node* component can visualize layered shapes, such as squares and connectors. It also incorporates properties from the objects that each *RepresentationItem* references, including *Style* and *Position*. For clarity, these referenced

objects are not shown in the figure. The React Flow node only renders *RepresentationItems* that are part of a *Representation* with a *type* attribute set to ”Node”.

Edge Component Design
The React Flow *Edge* component follows a similar design to the *Node* component, as shown in Figure 7.3.

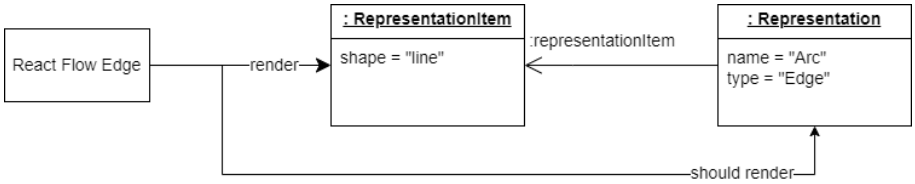


Figure 7.3: React Flow Edge Mapping to Graphical Representation Model

Like the *Node* component, the *Edge* component visualizes shapes defined by *RepresentationItems*. The specific rendering mechanisms follow the approach already outlined in the analysis chapter, Section 4.1.

7.4 Serializing and Deserializing Models

The universal diagram editor manages models in JSON format, including meta models, graphical representation models, and run-time instance models. This section explains how serialization and deserialization are handled.

When the run-time instance model is created as a Javascript object, it is converted into a JSON string using the *JSON.stringify* method. An example can be seen in Listing 7.1.

```
const serializedModel = JSON.stringify(instanceModel);
```

Listing 7.1: JSON Serialization of Javascript Object

This serialized string is then stored in the user’s local storage or sent to the backend for server-side storage.

When the editor needs to load a previously saved model, it retrieves the JSON string from the local storage or the backend and parses it back into a Javascript object using *JSON.parse*. An example can be seen in Listing 7.2.

```
const deserializedModel = JSON.parse(
    localStorage.getItem("representationInstanceModel")!
);
```

Listing 7.2: JSON Deserialization of Saved Object

7.4.1 Content of a Serialized Model

To show what the contents of a serialized model look like, a graphical representation instance model can be used. The model consists of layered representation items. Each item includes properties such as *shape*, *style*, and *position*. For example, a graphical

representation of a *Transition* element might consist of a square with multiple connectors, where their specific style properties are defined, as shown in Listing 7.3.

```
"package": {
  "uri": "http://www.compute.dtu.dk/sse/
          models/petri-net-representation-instance",
  "objects": [
    {
     "name": "Transition-9f9c0faa-17f5-4c56-a2a1-32e9c9829b0c",
     "type": "ClassNode",
     "position": {
        "x": 450,
        "y": 255
     },
      "classifier": {
        "$ref": "http://www.compute.dtu.dk/sse/models/petri-net-
                 instance#/objects/0"
     }
     "graphicalRepresentation": [
        {
         "shape": "square",
         "style": {
            "backgroundColor": "#f9f9f9",
            "borderColor": "#000",
            "borderWidth": 1,
            "borderStyle": "solid",
            "zIndex": 0
         },
         "position": {
            "x": 480,
            "y": 120,
            "extent": {
           "width": 100,
             "height": 100
            }
         }
        },
        {
         "shape": "connector",
         "style": {
            "color": "#000",
            "alignment": "left",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 1
         },
         "position": {
            "x": 485,
```

```
            "y": 165,
            "extent": {
             "width": 10,
             "height": 10
            }
          }
          ...
```

Listing 7.3: Example of a JSON–formatted Graphical Representation Model

Each representation item references the concept for which it provides a graphical appearance. The reference is seen under the ”classifier” property, and it uses the standard JSON *$ref* key feature to point to the concept object. The *Transition* concept that the representation references is seen in Listing 7.4.

```
"package": {
  "uri":
     "http://www.compute.dtu.dk/sse/models/petri-net-instance",
  "objects": [
    {
     "name": "Transition-9f9c0faa-17f5-4c56-a2a1-32e9c9829b0c",
     "type": {
        "$ref": "http://www.compute.dtu.dk/sse/
                 models/petri-net#/elements/0"
     },
     "attributes": [
        {
         "name": "name",
         "value": "transition"
        }
     ],
     "links": []
    },
    ...
```

Listing 7.4: Transition Concept Referenced by its Representation

This instantiated concept also uses the *$ref* key, which points to its meta model concept from which it is derived and instantiated.

8 Implementation

This chapter describes how the universal diagram editor was implemented, showing how the conceptual and software designs were converted into code. It presents technical details, including the rendering of shapes in React Flow, the storage of models, and the implementation of our custom constraint language. In addition, the challenges faced during development are discussed.

The complete source code for the developed universal diagram editor is available under the Apache License 2.0 at the following Github repository:
https://github.com/mikailcengizz/universal-diagram-editor.

8.1 Diagram Editor

The diagram editor was implemented using the React Flow library. This section details how custom React Flow components were defined to meet our requirements.

8.1.1 React Flow Setup

First, the main component of the diagram editor is loaded using the *ReactFlowProvider* and *ReactFlowWithInstance* components, as shown in Listing 8.1.

```
<ReactFlowProvider>
  <PaletteEditorPanel
    title={selectedMetaModel?.package?.name}
    notationElements={selectedMetaModel?.package?.elements as
       Class[]}
    selectedMetaModel={selectedMetaModel}
    selectedRepresentationMetaModel=
    {selectedRepresentationMetaModel}
  />
  <div>
    <div>
      <ReactFlowWithInstance
        nodes={nodes}
        edges={edges.map((edge) => ({
          ...edge,
          data: {
            ...edge.data,
            onDoubleClick: onDoubleClickEdge,
          },
        }))}
        onConnect={onConnect}
        onNodesChange={onNodesChange}
        onEdgesChange={onEdgesChange}
        setReactFlowInstance={setReactFlowInstance}
        onDrop={onDrop}
        onDragOver={onDragOver}
```

```
          nodeTypes={nodeTypes}
          edgeTypes={edgeTypes}
          snapToGrid={true}
          snapGrid={[15, 15]}
          showGrid={showGrid}
          connectionMode={ConnectionMode.Loose}
        />
      </div>
    </div>
  </ReactFlowProvider>
```

Listing 8.1: React Flow Setup

The *ReactFlowProvider* component initializes the React Flow context, allowing components outside *ReactFlowWithInstance* to hook into the internal state of React Flow. This allows, for example, the *PaletteEditorPanel* component to display the nodes and edges of the selected modeling language using the React Flow context.

The *ReactFlowWithInstance* is the core of the diagram editor. It is responsible for rendering all nodes and edges while managing user interactions, including connecting nodes with edges and dropping model elements onto the canvas.

8.1.2 Custom Node and Edge Components

To enable dynamic rendering mechanisms based on data given by our models, custom React components were implemented: *CombineShapesNode* for nodes and *CombineShapesEdge* for edges.

Custom Node Rendering

The *CombineShapesNode* component renders graphical features based on the *RepresentationItems* defined in the graphical representation model for the concept being rendered. These items could, for example, be rectangles, circles, compartments, labels, and handles (connectors), as shown in Listing 8.2.

```
const CombineShapesNode = ({id: nodeId, data: initialData,
   selected}: CombineObjectShapesNodeProps) => {
    ...
    return (
      <div ref={containerRef}>
        <RenderRectangles
          rectangles={rectangles}
        />

        <RenderCircles
          circles={circles}
        />

        <RenderCompartments
          compartments={compartments}
        />
```

```
          <RenderLabels
            labels={labels}
          />

          {!isPalette && !isNotationSlider &&
           connectors.length > 0 && (
            <RenderConnectors
              connectors={connectors}
            />
          )}
      );
};
```

Listing 8.2: Custom React Flow Node Component

The *nodeId* represents the unique identifier of the node, *initialData* contains custom data about the node, and *selected* indicates whether the node is currently selected.

The custom data *ReactFlowComponentData* passed to our React Flow components can be derived from its Typescript type seen in Listing 8.3.

```
export interface ReactFlowComponentData {
  // (meta model and representation model)
  notation?: Notation;
  // meta model concept for this node
  notationElement?: Class;
  // graphical representation of the notation element
  notationElementRepresentation?: Representation;
  // instance object for this node
  instanceObject?: InstanceObject;
  // graphical representation of the instance object
  instanceObjectRepresentation?: RepresentationInstanceObject;
}
```

Listing 8.3: Custom Data for React Flow Components

Furthermore, the node's content is rendered in layers, with each layer representing a specific graphical feature (e.g., rectangles, labels). For example, the code responsible for rendering rectangles onto a node is shown in Listing 8.4.

```
function RenderRectangles({
  isPalette = false,
  isNotationSlider = false,
  rectangles,
  data,
}: RenderRectanglesProps) {
  return (
    <>
      {rectangles.map((rect, index) => {
        return (
```

```
            <div
              key={index}
              style={{
                position: "absolute",
                left: `${rect.position.x}px`,
                top: `${rect.position.y}px`,
                width: `${
                  isPalette
                    ? (rect.position.extent?.width || 100) + "px"
                    : "100%"
                }`,
                height: `${
                  isPalette
                    ? (rect.position.extent?.height || 100) + "px"
                    : "100%"
                }`,
                backgroundColor: rect.style.backgroundColor,
                borderColor: rect.style.borderColor,
                borderWidth: rect.style.borderWidth,
                borderStyle: rect.style.borderStyle,
                borderRadius: rect.style.borderRadius + "px",
                zIndex: rect.style.zIndex,
              }}
            />
          );
        })}
      </>
    );
}
```

Listing 8.4: Render Rectangles for Custom Node

A similar layered approach is used to render the custom edge component *CombineShapesEdge*; therefore, it will not be shown.

8.1.3 Custom Node and Edge Integration

React Flow requires a mapping to associate its node and edge types with our custom components. This mapping is defined as shown in Listing 8.5.

```
const nodeTypes = {
  ClassNode: CombineShapesNode,
};

const edgeTypes = {
  ClassEdge: CombineShapesEdge,
};

<ReactFlowWithInstance
  ...
  nodeTypes={nodeTypes}
```

```
  edgeTypes={edgeTypes}
/>
```

Listing 8.5: Mapping Custom Node and Edge to React Flow

This configuration ensures that React Flow uses the custom *CombineShapesNode* and *CombineShapesEdge* components for rendering its nodes and edges.

8.1.4 Node and Edge Instantiation

Nodes are instantiated by dragging notations from the palette and dropping them onto the canvas, while edges are created by connecting nodes interactively. Listing 8.6 shows the process for instantiating nodes.

```
// Palette Component - Dragging Notation
const onDragStart = (event: React.DragEvent, notationElement:
   Class) => {
  const dragData: DragData = {
    notationElement: notationElement,
  };
  event.dataTransfer.setData("palette-item",
     JSON.stringify(dragData));
};

// Diagram Editor Component - Dropping Notation
const onDrop = useCallback((event: React.DragEvent) => {
  const dragData: DragData = JSON.parse(
    event.dataTransfer.getData("palette-item")
  );
  const { notationElement } = dragData;
  ...
  const newNode: Node = {
    id: uniqueId,
    type: notationElementRepresentation.type, // ClassNode,
                                              // EdgeNode
    handles: nodeHandles,
    position: position,
    data: nodeData as any,
  };
  setNodes((nds) => nds.concat(newNode));
}
```

Listing 8.6: Instantiation of Custom Nodes in React Flow

The *onDragStart* function is triggered when a diagram creator starts dragging a notation element from the palette. The dragged element is serialized into a *dragData* object, and stored in the browser's drag-and-drop API under the key "palette-item".

The *onDrop* function is triggered when the user drops the dragged notation element onto the diagram editor canvas. The *dragData* is retrieved and deserialized using *JSON.parse*. The *notationElement* is extracted and used to create a new node, which is added to the internal state of React Flow using *setNodes*.

Edge instantiation follows a similar process, with the *setEdges* method being used instead of *setNodes*.

8.2 API for Saving Models

The responsibility of the backend is storing and retrieving meta models and graphical representation models whenever requested by the *diagram editor* or *notation designer* components. This is done by exposing API endpoints that enable saving and updating models. An example API endpoint for storing meta models is shown in Listing 8.7. Meta models are stored on the server-side file storage as JSON files.

```
router.post("/save-meta-model-file", (req: Request, res:
    Response) => {
  const { name, uri, elements } = (req.body as
      MetaModel).package;

  if (!name || !uri || !elements) {
    return res.status(400).send("Configuration uri or elements
        are missing.");
  }

  ...

  fs.readdir(configDir, (err: any, files: string[]) => {
    files = files.filter((file: any) =>
        file.startsWith("meta"));

    for (const file of files) {
      const filePath = path.join(configDir, file);
      const content = fs.readFileSync(filePath, "utf-8");
      const config: MetaModel = JSON.parse(content);

      if (config.package && config.package.uri === uri) {
        // match found, update this file
        configFileFound = true;
        configFilename = file;
        break;
      }
    }

    const newConfig: MetaModel = {
      package: {
        name: name,
        uri: uri,
        elements: elements,
      },
    };

    if (configFileFound) {
      // update existing meta file
```

```
      fs.writeFileSync(
        path.join(configDir, configFilename),
        JSON.stringify(newConfig, null, 2)
      );
      return res
        .status(200)
        .send(`Configuration "${name}" updated successfully.`);
    } else {
      // create new meta file
      const newFilename = ...;
      fs.writeFileSync(
        path.join(configDir, newFilename),
        JSON.stringify(newConfig, null, 2)
      );
      return res
        .status(201)
        .send(`Configuration "${name}" saved successfully.`);
    }
  });
});
```

Listing 8.7: API Endpoint for Saving Meta Models

The endpoint ensures that the fields *name*, *uri*, and *elements* are present in the request body. Then, it checks if a file with the same URI already exists. If a match is found, the file is updated. Otherwise, a new file is created with a unique filename derived from the model name. The files are stored in the *configDir* directory in JSON format. The endpoint for saving representation models are similar and, therefore, will not be shown.

8.3 Local Storage Mechanisms

The universal diagram editor utilizes different storage mechanisms to manage models based on their type and usage. Instance models, which represent diagrams created by users, are stored on the frontend using the browser's local storage managed by Redux.

Listing 8.8 shows how Redux is used to handle instance models in the user's local storage.

```
const fallbackInstanceModel: InstanceModel = {
  package: {
    uri: "",
    objects: [],
  },
};

const storedInstanceModel: InstanceModel =
  JSON.parse(localStorage.getItem("instanceModel")!) ||
  fallbackInstanceModel;

localStorage.setItem("instanceModel",
   JSON.stringify(storedInstanceModel));
```

```
const initialState = {
  model: storedInstanceModel,
};

const metaInstanceModelReducer = (state = initialState, action:
   any) => {
  switch (action.type) {
    case UPDATE_INSTANCE_MODEL:
      const updatedModel = action.payload;
      localStorage.setItem("instanceModel",
         JSON.stringify(updatedModel));
      return { ...state, model: updatedModel };
  }
};
```

Listing 8.8: Redux Local Storage Management

A fallback instance model is defined for cases where no model exists in local storage. The existing model is loaded from *localStorage* and parsed into a Javascript object. When changes are made to the model, the updated version is stored back into *localStorage*.

The JSON-formatted contents of these instance models can be seen in Appendices A.7 and A.8.

8.4 Custom Constraint Language

To validate diagram instance models, a custom constraint language was implemented using an AST structure. This approach allows parsing the condition string into an AST representation, which is then traversed and evaluated node-by-node. Each node is processed according to its type, such as *BinaryExpression* for logical operations, *CallExpression* for method invocations, and others for literals or variables. The traversal logic is shown in Listing 8.9.

```
static parseConstraint = (condition: string) => {
  // convert the constraint condition string into an AST
  return jsep(condition);
};

static evaluateExpression = (
  node: any,
  context: Context,
  metaModel: MetaModel
): any => {
  switch (node.type) {
    case "BinaryExpression":
      const left = this.evaluateExpression(node.left, context,
         metaModel);
      const right = this.evaluateExpression(node.right,
         context, metaModel);
      switch (node.operator) {
```

```
          case "and":
          case "&&":
            return left && right;
          case "or":
          case "||":
            return left || right;
        }
      case "CallExpression":
        const methodName =
          node.callee.name ||
          node.callee.property?.name;
        const objName =
          node.callee.object?.name ||
          node.callee.object?.property?.name;
        const typeName = node.arguments[0]?.value;
        const object = objName === "source"
          ? context.self.source
          : context.self.target;

        return methods[methodName](object, typeName, metaModel);
      case "Identifier":
        return context[node.name as keyof Context];
      case "Literal":
        return node.value;
      case "Compound":
        return node.body.map((statement: any) =>
          this.evaluateExpression(statement, context, metaModel)
        );
      case "SequenceExpression":
        return node.expressions
          .map((expr: any) => this.evaluateExpression(expr,
             context, metaModel))
          .pop();
    }
  };
}
```

Listing 8.9: AST Traversal of Constraint Condition String

To use methods within our condition string, static methods can be implemented, such as *kindOf*. These methods provide reusable logic for validating specific conditions. Listing 8.10 demonstrates the implementation of the *kindOf* method.

```
const methods: any = {
  kindOf: (obj: InstanceObject, type: string, metaModel:
     MetaModel) => {
    const objType =
       ModelHelperFunctions.findClassFromInstanceObjectMetaModel(
      obj,
      metaModel
    )?.name;
```

```
    return objType === type;
  },
  // add more methods here
};
```

Listing 8.10: AST Structure of Custom Constraint Language

The *obj* parameter represents the instance object being validated. The *type* parameter specifies the type to check against. The *metaModel* parameter provides the meta model for resolving the instance object's concept type.

These methods are called during the *CallExpression* evaluation in the AST traversal.

8.5 Challenges

The implementation of the universal diagram editor presented several challenges. These challenges are summarized below:

1. React Flow Internal State
    - **Problem:** Implementing custom React Flow components was complex due to the need to synchronize these components with React Flow's internal state. This dependency causes issues when rendering components dynamically, especially in the palette editor.
    - **Solution:** Dummy checks and additional internal state management were implemented before rendering custom components.
2. Limited Control of React Flow Handles
    - **Problem:** React Flow's built-in handles require edges to connect from and to handles that has "source" and "target" as types. However, as handles are loaded first, and afterward edges are drawn, it is not known which handle becomes the source or target.
    - **Solution:** The React Flow editor was configured to use the "Loose" connection mode, which allows edges to connect to any type of handle. This is still fine as the underlying instance model keeps track of the source and target of edges.
3. Edge paths
    - **Problem:** Creating custom edge paths with anchor points and curved connections was challenging, as custom path algorithm needed to be implemented.
    - **Solution:** None. Edge paths are currently rendered as straight lines without anchor points or curved paths.
4. JSON References Across Models
    - **Problem:** The *$ref* mechanism for JSON object referencing only worked within a single JSON model. Referencing objects across different JSON model files was supported.
    - **Solution:** A custom helper function, *resolveRef*, was implemented to traverse a JSON model and find its correct object based on the *$ref* string.

9 Evaluation

This chapter describes how the universal diagram editor was evaluated. The evaluation consists of user testing, performance metrics, and a discussion of threats to validity.

9.1 Evaluation Methodology

The evaluation of the tool was conducted with two distinct user groups to gather diverse perspectives on the tool's usability and functionality. The methodology combined structured tasks, questionnaires, and interviews to collect qualitative feedback. The two user groups were:

1. **Diagram Creators**
    - Profile: Non-technical users familiar with basic computer use.
    - Tasks: Create diagrams using a predefined graphical notation. These tasks focused on drag-and-drop functionality, adding, editing, and arranging diagram elements.
    - Preparation: Participants received a brief tutorial introducing the diagram editor and its basic features.
2. **Graphical Notation Developers**
    - Profile: Technical users with software engineering backgrounds and basic UML modeling knowledge. These participants were chosen as a close alternative to actual graphical notation developers, who were unavailable.
    - Tasks: Define a simple meta model and its graphical representation, then validate the created notation by using it to create diagrams.
    - Preparation: Participants received a step-by-step introduction to the notation designer, including examples of predefined meta models and their graphical representation.

A total of five participants were selected, two as diagram creators and three as graphical notation developers. While diagram creators completed their tasks independently with minimal interference, graphical notation developers required significant guidance when defining meta models, which they found challenging. However, designing the graphical representation of meta model elements required only minimal support, as participants found the drag-and-drop interface more straightforward.

All participant interactions with the tool were observed, and feedback was collected through questionnaires and interviews.

9.1.1 Questionnaire

Participants completed a questionnaire to evaluate the tool in the following areas:

- **Purpose** of the tool.
- **Ease of use** of the diagram editor and graphical notation designer.

- **Satisfaction** with the performance and responsiveness.

The questionnaire consisted of closed and open questions. Closed questions used a five-point Likert scale, ranging from 1 (Strongly Disagree) to 5 (Strongly Agree). The full list of questions is as follows:

General Impressions

- **Q1:** The purpose of the tool was easy to understand.
- **Q2:** The user interface for customizing model elements was intuitive.
- **Q3:** The performance and responsiveness was satisfying.

Graphical Notation Designer (not asked to diagram creators)

- **Q4:** It was easy to define the meta model of a modeling language.
- **Q5:** The user interface of dragging-and-dropping shapes to create the graphical representation of model elements was intuitive and easy to use.

Diagram Editor

- **Q6:** It was easy to use model elements to create diagrams.
- **Q7:** The exportation of a diagram in picture format worked well.

Open Feedback

- **Q8:** What did you most like about the tool?
- **Q9:** What could be improved to make the tool more effective or easier to use?
- **Q10:** Is there any feature you expected but did not find in the tool?

The key observations made from the questionnaire were that graphical notation developers liked the drag-and-drop functionality for creating graphical representations, but they found it challenging to define meta models. Diagram creators thought that the editor was highly responsive and easy to use, but some areas such as resizing model elements and exporting of concrete syntax to picture format could be improved.

The questionnaire responses can be seen in Appendix A.5.

9.1.2 Interviews

In addition to the questionnaire, interviews were conducted with all participants to collect qualitative feedback on their experience.

Diagram creators found the diagram editor straightforward and user-friendly when creating diagrams. However, they also pointed out some limitations, such as errors when resizing model elements, where it would sometimes resize some graphical features more than others instead of keeping the same resize scale for all graphical features in one model element. Another aspect was the export and import of created diagrams in some XML format, so that it is possible to save and load the diagram to work on it at a later point.

Graphical notation developers liked the separation between the diagram editor and the graphical notation designer. However, all participants found the process of defining meta models difficult, with one stating, *"The meta modeling steps needs to be clearer or have*

some tooltip for beginners". Despite this challenge, all participants found it very cool how easy it was to have custom model elements directly appear in a diagram editor.

9.2 Performance Metrics

The system's performance was assessed manually without including any participants. The following metrics were used: load times, responsiveness, and export functionality. The results of these evaluations provide insight into the scalability of the tool.

9.2.1 Load Times

The time taken to render model elements in the palette was measured by switching between a small modeling language and a large modeling language:

- **Small Language:** Fewer than 5 modeling elements.
- **Large Language:** More than 5 modeling elements.

The results can be seen in Table 9.1.

Language Size	Load Time (ms)	Notes
Small (< 5 elements)	2 ms	Perceived as instant.
Large (> 5 elements)	2 ms	Perceived as instant.

Table 9.1: Rendering Model Elements in Diagram Editor Palette

There were no differences when loading model elements of a small modeling language compared to a large. The measurements were performed by implementing code in the tool for time tracking from where a modeling language is requested until all its model elements are loaded.

9.2.2 Responsiveness

Responsiveness during interactions such as dragging and zooming on the canvas was tested using canvas's that has multiple model elements.

Language Size	Notes
Dragging elements	Smooth with around 30 elements on the canvas.
Zooming canvas	Smooth with around 30 elements on the canvas.

Table 9.2: Canvas Responsiveness

9.2.3 Export of Concrete Syntax

The export of the concrete syntax of the diagrams as JPEG images was tested. The exported images were verified to ensure all diagram elements were included and correctly represented. Out of four different diagram exportations, all had 100% of their diagram

elements included. Furthermore, export times were consistent and was averaging between 0.10 ms and 0.50 ms depending on their diagram size. The measurements were performed by implementing code in the tool for time tracking from where the exportation is triggered until the picture is created.

9.2.4 Scalability

To assess scalability, stress tests were conducted by incrementally increasing the number of elements in a diagram to 200 elements. In Table 9.3 the results are shown.

Language Size	Notes
Dragging elements	Becomes laggy with around 70 elements on the canvas.
Zooming canvas	No lag.
Exporting concrete syntax	Times still averaging between 0.10 ms and 0.50 ms, but exportation did not include all elements in the JPEG picture file when placing many elements within a larger area.
Dropping elements from palette	Becomes laggy with around 150 elements on the canvas.

Table 9.3: Diagram Editor Stress Test

The performance evaluation showed that the system performs well under typical usage conditions, with acceptable load times and responsivenss. However, as diagrams grow in size, performance begins to decline.

9.3 Threat to Validity

The evaluation included several potential threats to validity, which are addressed below.

9.3.1 Limited Sample Size

The evaluation was carried out with only five participants, two as diagram creators and three as graphical notation developers. Although their feedback provided qualitative insights, the small sample size limits the broader perspective of the results. In addition, participants were selected based on personal relations rather than systematic selection, which may have introduced bias. A larger sample size would have provided a more thorough evaluation.

9.3.2 User Profiles

Due to the unavailability of actual graphical notation developers, software engineers with basic UML modeling knowledge were selected as a close alternative. While their background allowed them to some extent simulate tasks expected of graphical notation developers, their lack of domain-specific and meta modeling experience may have influenced the results. For instance, participants struggled with defining meta models,

which might not have been an issue for experienced graphical notation developers.

9.3.3 Scope of Languages

The evaluation primarily focused on the modeling languages Petri Nets and UML class diagrams, as these were prioritized and covered the most. However, the tool did demonstrate limitations within these languages.

Petri Nets could be modeled, but transitions were restricted to square boxes and could not be changed to bar boxes. Arcs were supported, but tokens were not.

UML class diagrams could be modeled, but classes could only display its attributes and not its operations, as these were excluded from the core model. In addition, not all types of associations were supported.

This limited scope leaves out specific graphical features and other types of notations, such as sequence diagrams, flowcharts, and other non-graph-related modeling languages. While the tool to some extent supports graphical notations requiring nodes and edges, expanding its scope to include additional graphical features and notations is crucial to make the editor more universal.

9.3.4 Ultimate Goal

The ultimate goal of the universal diagram editor is to enable graphical notation developers to create and modify meta models and graphical representations of a modeling language without requiring programming knowledge or generating editor code. This thesis demonstrates that this is achievable. However, achieving a high degree of customization, flexibility and simplicity while covering a wide variety of notations beyond Petri Nets and UML class diagrams remains a challenge. For instance, supporting notations that require nested or hierarchical structures would require improvements and are not fully supported in the current implementation.

9.4 Discussion

The evaluation of the universal diagram editor demonstrates its strengths while also highlighting areas for improvement. Compared to existing tools reviewed in the related work Chapter 3.1, the universal diagram editor offers several advantages.

A key strenth is the tool's ability to easily and dynamically configure graphical notations without requiring code generation or complex procedural setups. This feature eliminates the dependency on software developers to modify or implement graphical notations, which is a significant limitation of tools like Eclipse GMF. GMF depends on generating a complete diagram editor codebase for every customization made in a modeling language. Additionally, the universal diagram editor offers a more intuitive interface for defining meta models compared to the GOPRR model of MetaEdit+, which can be unmanageable due to its many-layered structure of objects, roles, and relationships.

However, several limitations were identified in the evaluation. The tool's current scope is limited to certain graph-related modeling languages, namely UML class diagrams and Petri Nets, excluding other types of model languages that may have unique graphical features. Features such as nested or hierarchical structures and complex graphical

behaviors important for notations like sequence diagrams or flowcharts are not fully supported. Similarly, while the lightweight diagram validator supports basic custom constraint language commands, it lacks the advanced validation mechanisms provided by OCL in GMF.

Overall, the universal diagram editor demonstrates significant improvements in the process of configuring graphical notations and using them in a diagram editor. Two modeling languages, Petri Nets, and UML class diagrams, could be successfully defined. Although these languages had some limitations, as described in Section 9.3.3, they proved that the customization of notations could be performed easily without programming or generating code for a diagram editor.

9.5 Future Works

The evaluation identified several areas for future improvement. These areas are outlined below:

9.5.1 Improved Meta Model Configuration

A significant improvement would be the introduction of a more user-friendly configuration panel for defining meta models. Currently, the graphical notation designer page is an intuitive textual UI that represents the meta model in a simple form, but the configuration process is still challenging. A dedicated meta model diagram editor that abstracts away from textual definitions entirely could make the tool more accessible. For instance, always providing a UML class diagram graphical modeling language as a foundation could allow graphical notation developers to create meta models in a diagram-based format (concrete syntax). These diagrams could then be exported as abstract syntax and imported into the graphical notation designer page for further customization.

9.5.2 Support for Additional Graphical Features

Expanding the tool's functionality to support non-graph-related modeling languages, such as sequence diagrams and flowcharts, would significantly broaden its applicability. This would involve introducing new graphical features and behaviors specific to these modeling languages. For example, sequence diagrams may require support for lifelines and messages, while flowcharts could include node elements with indirect relations without visualizing edges.

9.5.3 Advanced Constraint Validation

The current implementation supports basic custom constraint language commands but lacks advanced validation mechanisms. Extending this functionality to include a wider range of constraints, similar to OCL in GMF, would allow forcing more modeling rules. This could also include an additional panel for viewing all offline constraint violations in a created diagram, as the current tool only supports life constraints.

9.5.4 Improved Diagram Saving

Currently, the system only saves the latest created diagram in the user's local storage, limiting users to work on one diagram at a time. Implementing support for XMI (XML Metadata Interchange) saving and loading the abstract syntax instance model of

a created diagram would allow users to save multiple diagrams and switch between them.

9.5.5 Customizable Arrow Heads

The current implementation uses static arrowhead types like ”openArrow” and ”closedArrow”. Introducing drawable SVG-based arrowheads would allow users to customize and define new types of arrowheads, improving the tool’s ability to create unique diagram elements.

9.6 Conclusion

The evaluation of the universal diagram editor confirmed its ability to allow graphical notation developers to define meta models and graphical representations without programming or code generation. This approach simplifies the current complex process of customizing diagram editors and eliminates the dependency on software developers, addressing limitations in existing tools like Eclipse GMF.

The tool’s current scope is limited to graph-related modeling languages, such as UML class diagrams and Petri Nets, and lacks support for nested structures and advanced constraint validation.

This project demonstrates that easy configurability of graphical notations in diagram editors is achievable. Others who develop diagram editors can build on these findings to create editors that meet the needs of all users.

10 Conclusion

This thesis explored how graphical notation developers can define and customize graphical notations without programming and then use these notations in a diagram editor where its codebase does not need to be generated. The goal was to design and implement a universal diagram editor using MBSE principles and meta models. This work addressed the identified research questions and provided practical contributions to the field.

Research Question 1: *What are the key challenges in developing a universal diagram editor, and how can they be addressed?*

Key challenges included creating a flexible and simple core meta model, enabling direct reflection of graphical notation customizations in a diagram editor without code generation, and ensuring usability for non-technical users. These challenges were addressed by using the Ecore model for meta modeling, implementing a dynamic mapping model mechanism, and creating an intuitive web interface. User testing confirmed that these solutions allowed users to independently customize and use graphical notations.

Research Question 2: *How can we allow graphical notation developers to easily define and customize graphical notations without programming?*

The solution allows graphical notation developers to define meta models and their graphical representations through a user-friendly interface. Customization is simplified with drag-and-drop tools and automatic mapping between meta model concept and graphical representation model element. Additional advanced mappings are defined within the graphical designer drawing page, maintaining the easy usability and low-code approach.

Research Question 3: *How can a diagram editor be implemented in a dynamic way such that when its graphical notations are customized, the customizations are automatically reflected in the editor without having to generate new code for the editor?*

The universal diagram editor achieves its dynamic behavior by decoupling the diagram editor application from the meta models while integrating real-time mapping mechanisms. Unlike existing tools like Eclipse GMF, which require static code generation, this approach ensures that any customization to graphical notations is instantly reflected in the diagram editor, eliminating the need for code generation.

The tool currently supports two modeling languages: Petri Nets and UML class diagrams. For Petri Nets, places, transitions, and arcs are supported, but tokens and bar-shaped transitions are missing. For UML class diagrams, classes, and attributes can be modeled, but operations and some association types are not yet supported. These limitations highlight areas for further development. The system performs well under typical usage conditions, but responsiveness declines with larger diagrams.

This thesis contributes to the improvement of diagram editor system design by demonstrating how MBSE and meta models, combined with low-code approaches, can enable easy configuration of graphical notations and their immediate use in a diagram editor. By simplifying the process of defining and customizing graphical notations, this work improves accessibility for individuals interested in creating modeling languages and using them in diagram editors. Future work can build on this foundation by extending support for non-graph-related notations, improving meta model creation for graphical notation developers, and implementing more advanced constraint validation.

Bibliography

[1] Eclipse. *Eclipse Modeling Framework (EMF)*. Eclipse. Rond Point Schuman 11, Brussels, 1040, Belgium, 2024. https://eclipse.dev/modeling/emf/.

[2] Eclipse. *Graphical Modeling Project (GMP)*. Eclipse. Rond Point Schuman 11, Brussels, 1040, Belgium, 2024. https://eclipse.dev/modeling/gmp/.

[3] Roberto Rodriguez-Echeverria et al. *Towards a Language Server Protocol Infrastructure for Graphical Modeling*. Association for Computing Machinery, 2018.

[4] Eclipse. *Rich Client Platform*. Eclipse. Rond Point Schuman 11, Brussels, 1040, Belgium, 2024. https://wiki.eclipse.org/Rich_Client_Platform/.

[5] OMG. *OMG Meta Object Facility (MOF) Core Specification*. Object Management Group. 140 Kendrick Street, Needham, MA 02494, USA, 2019. https://www.omg.org/spec/MOF/2.5.1/PDF.

[6] Enrico Biermann et al. *Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework*. ResearchGate, 2006.

[7] Eclipse. *Graphical Modeling Framework/Documentation*. Eclipse. Rond Point Schuman 11, Brussels, 1040, Belgium, 2024. https://wiki.eclipse.org/GMF_Documentation.

[8] OMG. *Object Constraint Language*. Object Management Group. 140 Kendrick Street, Needham, MA 02494, USA, 2014. https://www.omg.org/spec/OCL/2.4/PDF.

[9] Heiko Kern. *The Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF Using M3-Level-Based Bridges*. University of Leipzig, 2008.

[10] Scott A. Helmers. *Microsoft Visio 2013 Step by Step*. O'Reilly Media, 2013.

[11] Heiko Kern and Stefan Kuhne. *Integration of Microsoft Visio and Eclipse Modeling Framework Using M3-Level-Based Bridges*. University of Leipzig, 2009.

[12] Dimitris Karagiannis and Harald Kühn. *Metamodelling Platforms*. Springer-Verlag Berlin, 2002.

[13] Paul Johannesson and Erik Perjons. *An Introduction to Design Science*. Springer, 2014.

[14] Alan Hevner and Samir Chatterjee. *Design Research in Information Systems*. Springer, 2010.

[15] Hatice Koç et al. *UML Diagrams in Software Engineering Research: A Systematic Literature Review*. MDPI, 2021.

[16] Stephen White. *Introduction to BPMN*. IBM Corporation, 2004.

[17] James Peterson. *Petri Nets*. Association for Computing Machinery, 1977.

[18] Visual Paradigm. *UML Class Diagram Tutorial*. Visual Paradigm. 1802, Laford Centre, 838 Lai Chi Kok Road, KLN, Hong Kong, 2024. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/.

[19] Ekkart Kindler. *Model-based Software Engineering and Process-Aware Information Systems*. Springer, 2009.

[20] React Flow. *React Flow Documentation*. xyflow. 2024. https://reactflow.dev/learn.

A Appendix

A.1 Source code for the Universal Diagram Editor

The codebase for the universal diagram editor developed in this thesis is publicly available as an open-source repository on Github under the Apache License 2.0: https://github.com/mikailcengizz/universal-diagram-editor.

A.2 Ecore Model in Abstract Syntax

This section presents the Ecore model represented in abstract syntax.

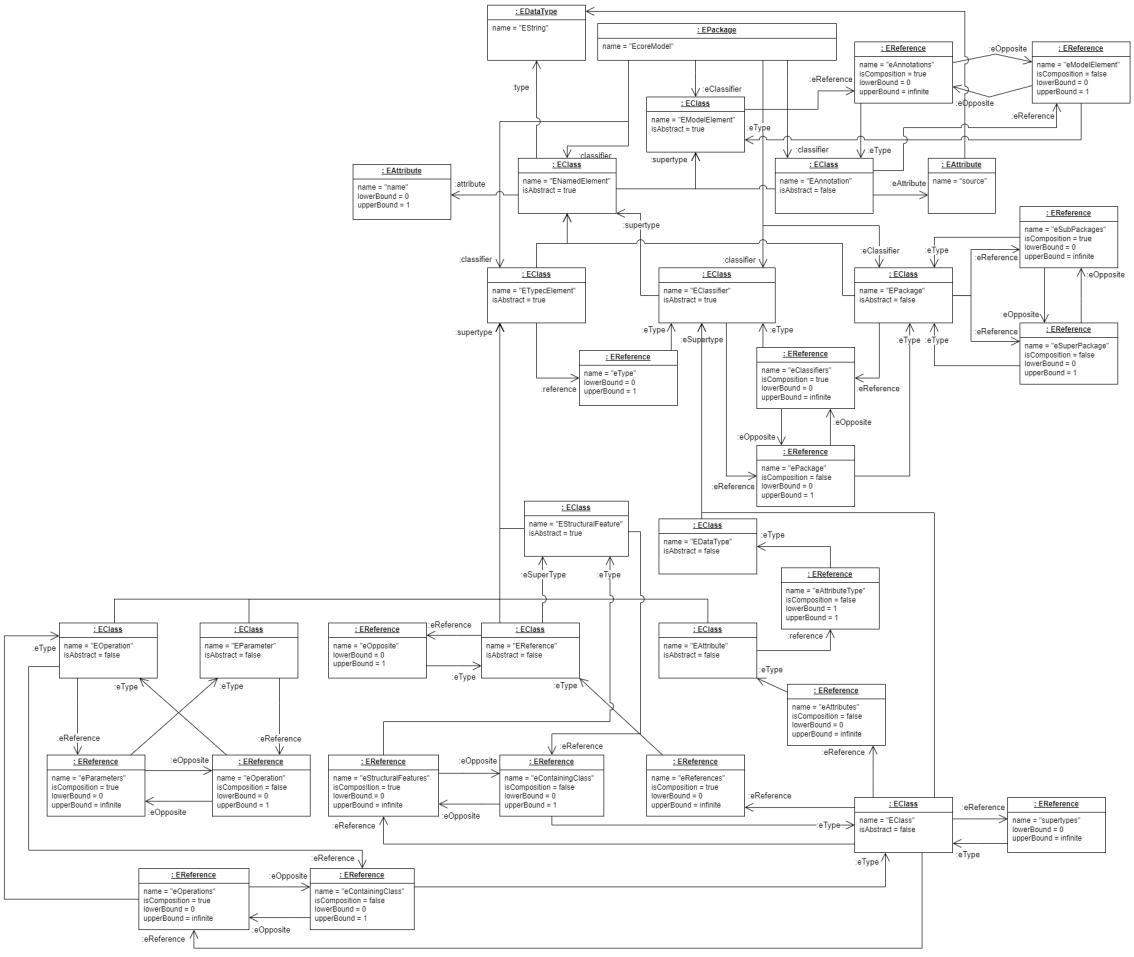


Figure A.1: Ecore Model in Abstract Syntax

A.3 Snippet of the GOPRR Meta Model

This section presents a snippet of the GOPRR meta model used by MetaEdit+.

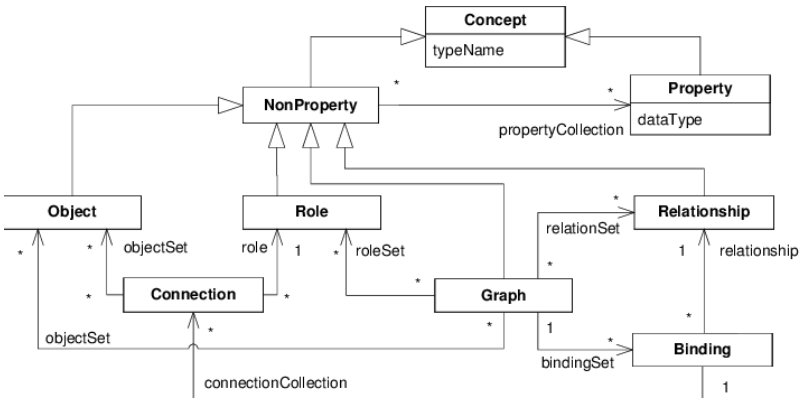


Figure A.2: Snippet of the GOPRR Meta Model

A.4 Snippet of the Visio Object Meta Model

This section presents a snippet of the Visio Object Meta model used by Microsoft Visio.

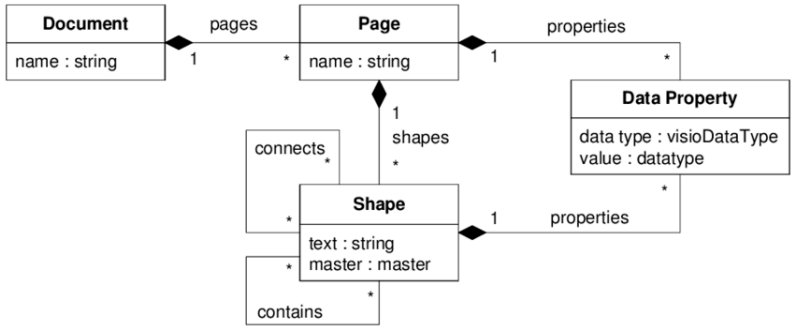


Figure A.3: Snippet of the Visio Object Meta Model

A.5 Questionnaire: Evaluating the Universal Diagram Editor

This section presents the questionnaire used to evaluate the usability and functionality of the universal diagram editor.

Below is a list of questions to evaluate your experience using the diagram editor or the graphical notation designer. Please rate each question on a scale from 1 to 5, where:
1: Strongly Disagree
2: Disagree
3: Neutral
4: Agree
5: Strongly Agree

General Impressions
Q1: The purpose of the tool was easy to understand
Q2: The user interface for customizing model elements was intuitive
Q3: The performance and responsiveness was satisfying

Graphical Notation Designer (if you have used it)
Q4: It was easy to define the meta-model of a modeling language

Q5: The user interface of dragging-and-dropping shapes to create the graphical representation of model elements was intuitive and easy to use

Diagram Editor
Q6: It was easy to use model elements to create diagrams
Q7: The exportation of a diagram in picture format worked well

Open Feedback
Q8: What did you most like about the tool?
Q9: What could be improved to make the tool more effective or easier to use?
Q10: Is there any feature you expected but did not find in the tool?

A.6 Questionnaire Answers

This section presents the collected responses from participants who tested the universal diagram editor. The answers are shown in the Excel sheet in Figure A.4.

Date	User	General Impressions		Graphical Notation Designer		Diagram Editor		Open Feedback		
		The purpose of the tool was easy to understand	The performance and responsiveness was satisfying	It was easy to define the meta model of a modeling language	The user interface of dragging-and-dropping shapes to create the graphical representation of model elements was intuitive and easy to use	It was easy to use model elements to create diagrams	The exportation of a diagram in picture format worked well	What did you most like about the tool?	What could be improved to make the tool more effective or easier to use?	Is there any feature you expected but did not find in the tool?
20/11-2024	1 (Graphical Notation Developer)	4	4	4	5	4	3	The drag and drop interface to create the visual representation of a model element	Add more tooltips for beginners	Expected more export format options
25/11-2024	2 (Diagram Creator)	3	5	N/A	N/A	4	4	Pretty UI and it felt smooth and responsive to use	Resizing of diagram elements is sometimes weird when dragging	Maybe could also import diagrams to canvas
26/11-2024	3 (Graphical Notation Developer)	3	4	3	5	5	5	When you first have the meta model defined it is fast and easy to customize the model elements	Defining the meta model could be improved as it was hard to understand	Highlighting diagram elements and moving them all at once
26/11-2024	4 (Graphical Notation Developer)	3	5	3	5	5	2	That you can customize model elements and use them directly	Could be more shapes when designing the visual representation	More customizations in the diagram editor
28/11-2024	5 (Diagram Creator)	4	4	N/A	N/A	5	3	Designing the model elements is very cool	The diagram export includes the whole canvas, could be nice if it only included the area where the diagram elements are	Online collaboration when creating diagrams

Figure A.4: Test users' answers to the questionnaire.

A.7 JSON Formatted File Representing a Petri Net Instance Model

This section presents an example of how the system stores models in JSON format. Listing A.1 represents a Petri Net instance model.

```
{
  "package": {
    "uri": "http://www.compute.dtu.dk/sse/
            models/petri-net-instance",
    "objects": [
      {
        "name": "Transition-010e95b9-8b85-
                 42ab-901d-60beebfc1dfe",
        "type": {
         "$ref": "http://www.compute.dtu.dk/sse/
                   models/petri-net#/elements/0"
        },
        "attributes": [
         {
          "name": "name",
            "value": "transition"
         }
        ],
        "links": [],
      },
      {
        "name": "Place-e760226e-1761-4e67-
                 8f84-e75a8b404ba9",
        "type": {
         "$ref": "http://www.compute.dtu.dk/sse/
                   models/petri-net#/elements/6"
        },
        "attributes": [],
        "links": [],
      },
      {
        "name": "Arc-4a0b8bcd-bb11-4c3b-
                 83ec-01ada8c41c35",
        "type": {
         "$ref": "http://www.compute.dtu.dk/sse/
                   models/petri-net#/elements/7"
        },
        "attributes": [],
        "links": [
         {
            "name": "source",
            "target": {
             "$ref": "http://www.compute.dtu.dk/sse/
                       models/petri-net-
                       representation-instance#/
```

```
                    objects/0/representation/
                    graphicalRepresentation/4"
            }
          },
          {
            "name": "target",
            "target": {
             "$ref": "http://www.compute.dtu.dk/sse/
                    models/petri-net-
                    representation-instance#/
                    objects/1/representation/
                    graphicalRepresentation/3"
            }
          }
       ],
      }
    ]
  }
}
```

Listing A.1: JSON Formatted File of a Petri Net Instance Model

A.8 JSON Formatted File Representing a Petri Net Graphical Representation Instance Model

This section presents an example of how the systems stores graphical representation models in JSON format. Listing A.2 shows a graphical representation instance model for a Petri Net.

```
{
 "package": {
  "uri": "http://www.compute.dtu.dk/sse/
          models/petri-net-representation-instance",
  "objects": [
    {
     "name": "Transition-010e95b9-8b85-42ab-901d-60beebfc1dfe",
     "type": "ClassNode",
     "position": {
        "x": 480,
        "y": 270
     },
      "classifier": {
        "$ref": "http://www.compute.dtu.dk/sse/models/petri-net-
                instance#/objects/0"
     },
     "graphicalRepresentation": [
        {
         "shape": "square",
         "style": {
            "backgroundColor": "#f9f9f9",
            "borderColor": "#000",
```

```
            "borderWidth": 1,
            "borderStyle": "solid",
            "zIndex": 1
        },
        "position": {
            "x": 480,
            "y": 120,
            "extent": {
        "width": 100,
             "height": 100
            }
        }
    },
    {
        "shape": "connector",
        "style": {
            "color": "#000",
            "alignment": "left",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
        },
        "position": {
            "x": 485,
            "y": 165,
            "extent": {
             "width": 10,
             "height": 10
            }
        }
      },
      {
        "shape": "connector",
        "style": {
         "color": "#000",
            "alignment": "right",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
        },
        "position": {
            "x": 565,
            "y": 165,
            "extent": {
             "width": 10,
             "height": 10
```

```
            }
          }
        },
        {
          "shape": "connector",
          "style": {
            "color": "#000",
            "alignment": "top",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
          },
          "position": {
            "x": 525,
            "y": 125,
            "extent": {
              "width": 10,
              "height": 10
            }
          }
        },
        {
          "shape": "connector",
          "style": {
            "color": "#000",
            "alignment": "bottom",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
          },
          "position": {
            "x": 525,
            "y": 205,
            "extent": {
              "width": 10,
              "height": 10
            }
          }
        }
      ]
    },
    {
      "name": "Place-e760226e-1761-4e67-8f84-e75a8b404ba9",
      "type": "ClassNode",
      "position": {
        "x": 480,
```

```
    "y": 510
},
 "classifier": {
    "$ref": "http://www.compute.dtu.dk/sse/models/petri-net-
            instance#/objects/1"
},
"graphicalRepresentation": [
    {
     "shape": "circle",
     "style": {
        "backgroundColor": "#f9f9f9",
        "borderColor": "#000",
        "borderWidth": 1,
        "borderStyle": "solid",
        "borderRadius": "50%",
         "zIndex": 1
        },
     "position": {
        "x": 590,
        "y": 160,
        "extent": {
         "width": 100,
         "height": 100
        }
     }
    },
    {
     "shape": "connector",
     "style": {
        "color": "#000",
        "alignment": "left",
        "fontSize": 6,
        "borderRadius": 50,
        "borderColor": "#000",
        "borderWidth": 1,
        "zIndex": 2
     },
     "position": {
        "x": 595,
        "y": 205,
        "extent": {
         "width": 10,
         "height": 10
        }
     }
 },
    {
     "shape": "connector",
     "style": {
        "color": "#000",
```

```
            "alignment": "right",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
        },
        "position": {
            "x": 675,
            "y": 205,
            "extent": {
             "width": 10,
             "height": 10
            }
        }
       },
       {
        "shape": "connector",
        "style": {
         "color": "#000",
            "alignment": "top",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
        },
        "position": {
            "x": 635,
            "y": 165,
            "extent": {
             "width": 10,
             "height": 10
            }
        }
       },
    {
        "shape": "connector",
        "style": {
            "color": "#000",
            "alignment": "bottom",
            "fontSize": 6,
            "borderRadius": 50,
            "borderColor": "#000",
            "borderWidth": 1,
            "zIndex": 2
        },
        "position": {
            "x": 635,
            "y": 245,
```

```
            "extent": {
             "width": 10,
             "height": 10
            }
         }
        }
     ]
    },
    {
     "name": "Arc-4a0b8bcd-bb11-4c3b-83ec-01ada8c41c35",
     "type": "ClassEdge",
     "position": {
      "x": 0,
        "y": 0
     },
      "classifier": {
        "$ref": "http://www.compute.dtu.dk/sse/models/petri-net-
                 instance#/objects/2"
     },
     "graphicalRepresentation": [
        {
         "shape": "line",
         "style": {
            "color": "#000000",
            "width": 1,
            "lineStyle": "solid"
         },
         "markers": [
            {
             "type": "none",
             "style": {
                "color": "#000000",
                "width": 10
             }
            },
            {
             "type": "closedArrow",
             "style": {
                "color": "#000000",
                "width": 10
             }
            }
         ],
         "position": {
            "x": 0,
            "y": 0
         }
        }
     ]
    }
```

```
    ]
  }
}
```

Listing A.2: JSON Formatted File of a Petri Net Graphical Representation Instance Model

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk