Development and Evaluation of a Metamodel to Define Modeling Syntaxes for CouchEdit

Hannah Lappe

Universität Ulm

Fakultät für
Ingenieurwissenschaften,
Informatik und
Psychologie

Institut für
Softwaretechnik und
Programmiersprachen

Oktober 2020

Bachelorarbeit im
Studiengang Informatik

Abstract

Modeling has become an essential factor in many of today's development workflows. Especially recent developments in the area of Model-Driven Engineering, as well as Business Process Modeling, have increased the need for syntactically correct models. Therefore, graphical modeling editors with syntax parsing capabilities are integral. However, most modern model editors do not provide optimal user experience, resulting from the tight coupling between graphical and abstract notations.

COUCHEDIT is a novel modeling language agnostic framework that strives to improve editor usability. To this end, it introduces an architecture that decouples graphical and abstract representations. However, in its current implementation, configuring COUCHEDIT for different modeling languages is a laborious and error-prone task.

This thesis proposes an architecture that can create modeling language configurations for COUCHEDIT. This architecture aims to improve developer accessibility by reducing the amount of code and framework knowledge needed to specify new configurations. While test results indicate that the developed artifact improves usability, the current prototype introduces much performance overhead which has to be addressed in future work.

Gutachter:	Prof. Dr. Matthias Tichy
Betreuer:	Dr. Alexander Raschke
	Michael Stegmaier

Fassung October 14, 2020
© 2020 Hannah Lappe
Satz: PDF-LaTeX 2_ε

Contents

1 Introduction

Modeling languages have long played a vital role in software engineering. Properly designed models can abstract complex systems and provide a visual aid in understanding them. Furthermore, in the form of the Business Process Model Notation (BPMN), they can define and automate processes. Today, with research around Model-Driven Engineering (MDE), a paradigm centered around models, with the intent to generate code bases and whole systems from them, the importance of models is rising even more.

As these tasks require syntactic correctness of used models, modeling tools become an essential part of an engineer's workflow. In theory, visual modeling tools especially provide an intuitive and user-friendly way to design models. However, Current graphical modeling tools often constrain users in unintuitive ways and deliver subpar user experience (UX), commonly stemming from a tight coupling between a modeling tools user interface (UI) and the underlying model. As the model's syntax usually is inflexible, the UI must make restrictions to adhere to this syntax, often creating issues for the user. Typically, this can manifest as connections only being drawable between two existing states or the undesired deletion of child nodes coupled to the removal of the parent.

To amend these usability woes, Nachreiner proposed a novel modeling framework called COUCHEDIT [1]. COUCHEDIT is categorized as a relaxed conformance modeling Framework. This means it can loosen the strictness requirements that are imposed onto the graphic model representation. To this end, the framework decouples user interface and model syntax by introducing different models for both. In the COUCHEDIT architecture, the user interface, instead of relying directly on the syntax of the model to be designed, uses a render-model that specifies only basic graphic objects. On the other hand, the conceptual representation of the model now stands on its own. A third metamodel then connects these two models. At its core, COUCHEDIT is a general-purpose framework, rewritable to adhere to any model syntax. Nevertheless, to realize this in the current implementation, the source code must be changed directly, which is error-prone, convoluted, and requires an understanding of COUCHEDIT's internal architecture.

Aiming to create a more developer-friendly and flexible way of adapting COUCHEDIT to different modeling syntaxes, this design research proposes a new metamodel that can create modeling syntax definitions, which are usable by COUCHEDIT. Furthermore, a conceptual prototype is presented that provides proof of concept on how this newly developed metamodel interacts with the COUCHEDIT architecture.

1.1 Problem Statement

A general-purpose framework should be configurable for multiple use cases in its designated domain. COUCHEDIT, as a general-purpose graphical modeling framework, thus should be configurable for multiple modeling syntaxes. Technically this is possible, as long as one has access to the source code. However, this would mean, every time COUCHEDIT has to support a new modeling syntax, manual changes in the source code have to be made, and the project has to be compiled from sources. Implementing a configuration parser that can interpret modeling syntax definitions at runtime would thus increase flexibility. Furthermore, a well-designed metamodel could reduce the amount of knowledge required about the COUCHEDIT framework when configuring new model syntaxes.

As a relaxed conformance editing framework, COUCHEDIT poses special architectural requirements. It must allow for temporary inconsistencies between concrete syntax (what the user draws) and abstract syntax (what the underlying model looks like). As the concrete representation does not always have to map to a syntactic correct abstract model, the user has more freedom during the modeling process (e.g., dangling transitions[1]).

COUCHEDIT achieves this by building upon the architecture concept of clear separation between concrete and abstract syntax, proposed by Y. Van Tendeloo et al. [2]. Internally, COUCHEDIT builds a hypergraph that maps the given concrete representation to an abstract syntaxes instance. A set of processors, connected in a reactive publish and subscribe pattern, are responsible for building this hypergraph. All processors (and the user interface) are subscribed to a bus (Figure 1.1). If a change (diff) is published to the bus (e.g., the user adds a node to the concrete syntax), all processors that are subscribed to this type of diff are notified and calculate new resulting diffs, these new diffs are then also published to the bus, and all processors interested in them are invoked as well.

Some of these processors are needed for every type of syntax model. For example, the spatial processor processes how nodes in the concrete syntax are positioned to each other (above, besides, etc.). Other processors are specific to the given modeling syntax. For example, a Statecharts syntax would require a state processor that processes if a given graphical object represents a state (usually true if the given node is a rectangle with rounded edges).

A modeling syntax parser for COUCHEDIT would have to generate these syntax specific processors while considering multiple design constraints that result from this architecture. Thus a metamodel is needed that can define the desired abstract syntax and specify a mapping from concrete representations to this abstract syntax. While there is ongoing research in the area of relaxed conformance editing and how to link concrete and abstract syntax, the design of such a metamodel does not immediately become apparent.

[1]Many modeling editors force transition lines to always connect two model elements. If one of the connected elements is removed, the line will also be removed as a result.

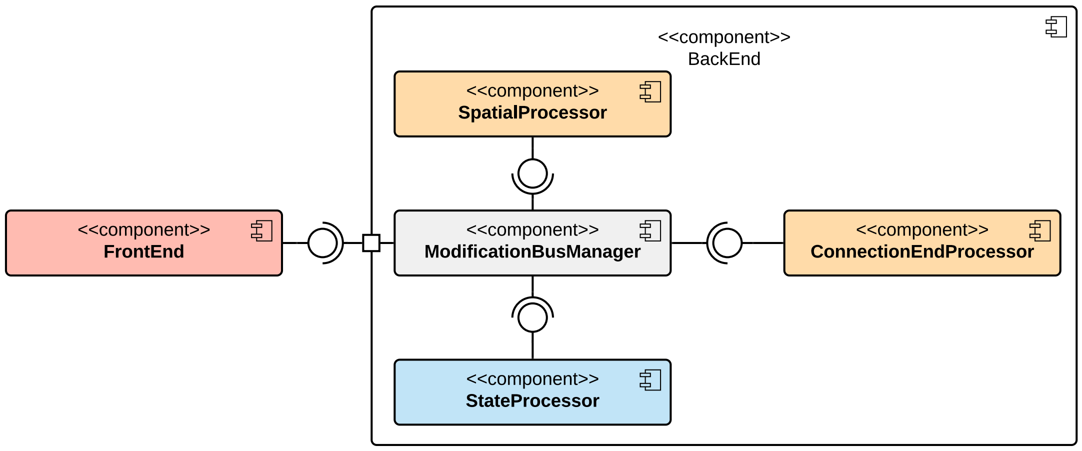


Figure 1.1: Publish Subscribe architecture of CouchEdit

1.2 Purpose of this Study

The primary purpose of this study was to develop and evaluate a metamodel for the CouchEdit framework. This metamodel is supposed to provide a comprehensive and easy to use way for defining new modeling languages. Furthermore, a prototypical code generator was implemented to evaluate the metamodel's applicability. This code generator can comprehend the newly designed metamodel and translate it into a source code implementation, which the current implementation of CouchEdit can use. Additionally, a simple domain-specific language (DSL) was developed that eases the process of specifying instances of this metamodel.

This extension of the CouchEdit framework is supposed to improve developer accessibility and framework flexibility. While a code generator means that the system still has to be recompiled for every modeling syntax, it still becomes more comfortable to support new modeling syntaxes as the metamodel abstracts away from the actual source code implementation. It thus requires less knowledge about the CouchEdit framework, as well as fewer lines of code.

1.3 Research Questions

The primary goal of this research was the development of a metamodel for the CouchEdit framework. To this end, the following questions were defined as a means to guide the research.

RQ1: Is it possible to define a concise metamodel that covers CouchEdit's feature set?

If this is not possible, the following question has to be answered.

RQ1.1: How could the feature set be narrowed down without impacting the framework's capabilities too much?

RQ2: What requirements does the COUCHEDIT architecture impose on a metamodel?

The architecture's characteristics and quirks have to be considered when designing the language. For example, the publish and subscribe pattern could make it easy to introduce nonterminating processing chains, which the metamodel should prevent where possible.

RQ3: How applicable is the designed metamodel for defining new modeling syntaxes?

RQ3.1 Is it possible to express common modeling syntax concepts clearly and concisely?

RQ3.2 Are complex concepts still expressible without causing too much convolution?

1.4 Thesis Structure

This thesis is organized in the following structure: Chapter 2 lays out the fundamentals required in the rest of the thesis. This includes a comprehensive summary of the COUCHEDIT architecture as well as modeling theory. After that, Chapter 3 summarizes related work in the relevant fields. Following that, in Chapter 4, the developed metamodel is described in detail. Chapter 5 then presents the prototype implementation. Chapter 6 provides considerations about the evaluation of the artifact, as well as the evaluation results. Finally, Chapter 7 provides takeaways and perspectives on the future of this project.

2 Fundamentals

This chapter lays out the knowledge foundation required in later sections. First, the methodology that guided this research will be described. Following that, two modeling language notations are presented. Finally, a comprehensive overview of COUCHEDIT's architecture and its features is provided.

2.1 Methodology

This research strived to develop a new metamodel suitable for the COUCHEDIT framework. Therefore, it was conducted following the design science research (DSR) approach. According to V. Vaishnavi et al., a design science research process consists of five steps [3].

2.1.1 Problem Awareness

The first step of design science research is the identification of existing problems. As specified in Section 1.1, it was identified that the current COUCHEDIT implementation lacks a developer-friendly way to configure it for different modeling syntaxes and that it is unclear how to design a metamodel for this architecture.

2.1.2 Suggestion

With a clear definition of the problem, objectives can be proposed which have to be achieved in order to solve the identified problem. The first objective of this research is to develop a metamodel applicable to COUCHEDIT's architecture. To be more precise, a metamodel is to be designed that can be used to specify model syntax definitions that map concrete graphical syntaxes to Abstract syntax models and is applicable to COUCHEDIT's architecture. The second objective is to implement a prototype that provides proof of concept for the designed metamodel's applicability.

2.1.3 Development

The primary goal of design science research is the development of artifacts. The first artifact to be developed in this research will be a metamodel that can be used to define new modeling syntaxes for a relaxed conformance editor and applies to CouchEdit's architecture. The second artifact that is to be developed is a prototypical code generator that can translate the developed metamodel into a CouchEdit implementation, which will be able to process the defined modeling syntax. Furthermore, it was decided to implement a simple DSL that eases the process of defining instances of the given metamodel.

To this end, the first sub-step of the development stage has to be a comprehensive analysis of CouchEdit's architecture. In his work, Nachreiner describes in detail which modeling features the framework covers and how they are implemented [1]. The goal is to develop an approach that can streamline defining modeling syntaxes while adhering to CouchEdit's specification. Related work defined in Chapter 3 serves as a basis from which a first suitable metamodel can be derived.

In the next sub-step, the code generator will be implemented. This is an iterative step. First, an implementation will be developed based on the designed metamodel. This implementation should reveal further requirements imposed by CouchEdit's architecture. The metamodel then has to be adjusted to account for these requirements, which in turn requires changes in the implementation. This iteration has to be repeated until no further requirements are deductible. Because of time constraints, it is not possible to implement all of the metamodel's features. Instead, the prototype covers a minimal feature set that still suffices to demonstrate the metamodel's applicability for selected modeling syntaxes.

2.1.4 Evaluation

Now that the desired artifacts are fully developed, it has to be evaluated how well they do their designated tasks. As the primary artifact of this research, the metamodel has to be evaluated in terms of its applicability to its designated domain. To this end, different modeling syntaxes will be implemented using the developed code generator. This will serve as a demonstration of the developed artifacts and should highlight areas in which the metamodel's design excels, as well as design flaws and limitations. If time allows, potential flaws can be addressed by returning to the development phase and revising the artifacts. Otherwise, flaws are to be highlighted so that future work can address them.

2.1.5 Conclusion

The conclusion stage marks the end of a DSR, and the results have to be communicated. This written thesis as well as all source code and documentation will serve as a means to communicate the results of this research.

2.2 Modeling Languages

This thesis discusses modeling syntaxes and how CouchEdit's architecture handles them. To this end, this section introduces the Petri nets and Statecharts syntaxes. They will serve as examples to demonstrate various problems and applicability of the developed artifacts.

2.2.1 Petri Nets Syntax

Petri nets are easy to understand and provide simple abstract and concrete syntax. Thus they serve as a suitable tool for explaining concepts in the following chapters. Figure 2.1 shows the abstract syntax metamodel of the used Petri nets syntax. A Petri net's conceptual representation consists of *places* and *transitions*. While Both possess a name, places furthermore have a token count. Each place can have a variable amount of incoming and outgoing transitions, while transitions can have any amount of incoming and outgoing places.

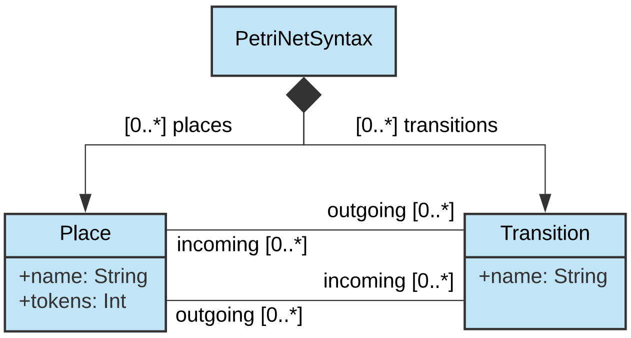
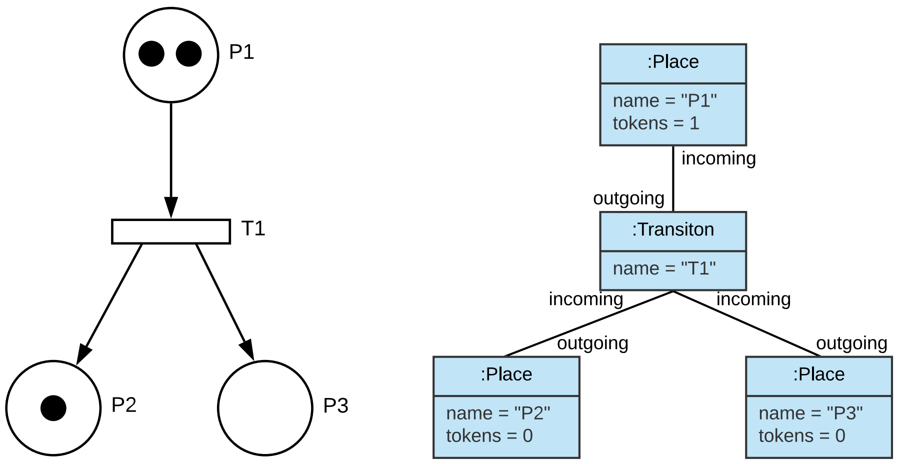


Figure 2.1: Metamodel for a simple Petri nets abstract syntax

The graphic primitives, a concrete representation of Petri nets is composed of, are listed in Table 2.1. Usually, circles without fill color represent places, while slender rectangles represent transitions. The number of small black circles inside a place represents this place's token count. Furthermore, places and transitions possess a label close to their bounding box that determines their name. Lastly, places and transitions have directed connection lines between them. Each connection points from a place towards a transition, or from a transition towards a place. A connection marks the target element as an outgoing reference for the source element and vice versa. Figure 2.2 shows a simple concrete instance and the corresponding abstract representation.

Place	Transition	Token	Label	Connection
○	▭	●	P1	→

Table 2.1: Graphic primitives used to describe Petri nets.



(a) concrete syntax

(b) abstract syntax

Figure 2.2: Concrete and abstract representation for a simple Petri nets example.

2.2.2 Statechart Syntax

As a second example, Statecharts are introduced here. The term Statecharts was first coined in 1987 by D. Harel [4]. Statecharts are an extension of state machines. Their primary feature is that each state can have itself sub-state machines. Figure 2.3 describes the abstract syntax metamodel used in this thesis. This abstract syntax does not represent a complete specification of the Statecharts syntax, rather a subset that can highlight specific areas of the implementation. Statecharts define a more complex syntax than Petri nets mentioned above and thus will show the developed artifacts applicability towards more complex modeling languages.

A Statechart is composed of *state elements* and *transitions*. Transitions might connect state elements. Each transition has exactly one source state element and one target state element. state elements can either be *pseudo states* (e.g., initial state, choice, etc.) or normal *states*. A normal state can either possess a name, a name and a sub-state machine, or multiple *regions*. Each region contains one state machine. regions are marked by splitting up states using dashed lines.

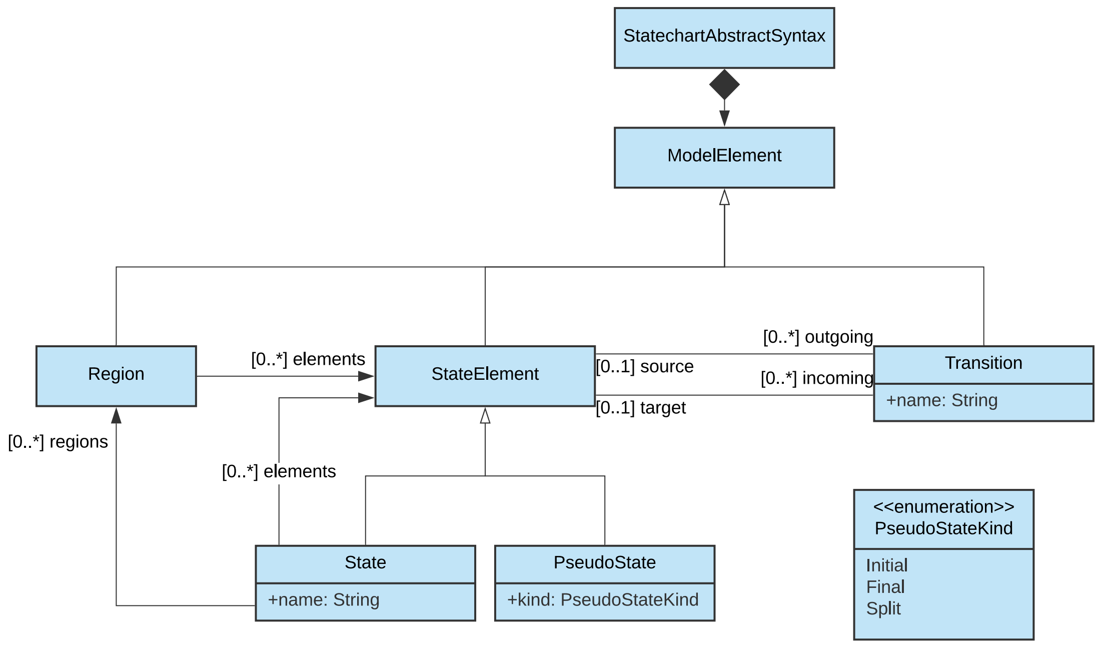


Figure 2.3: Abstract Metamodel of Statecharts

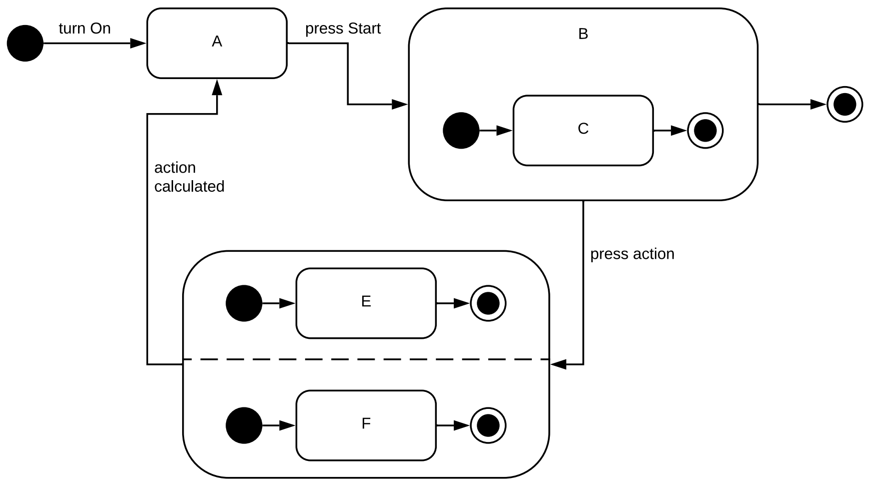


Figure 2.4: Example Statechart

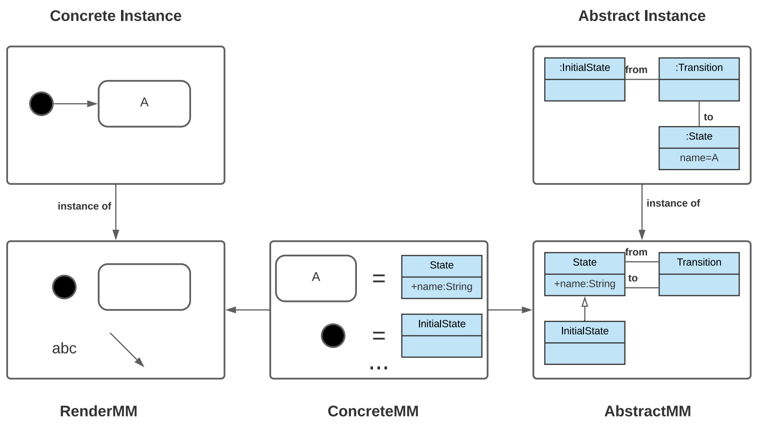


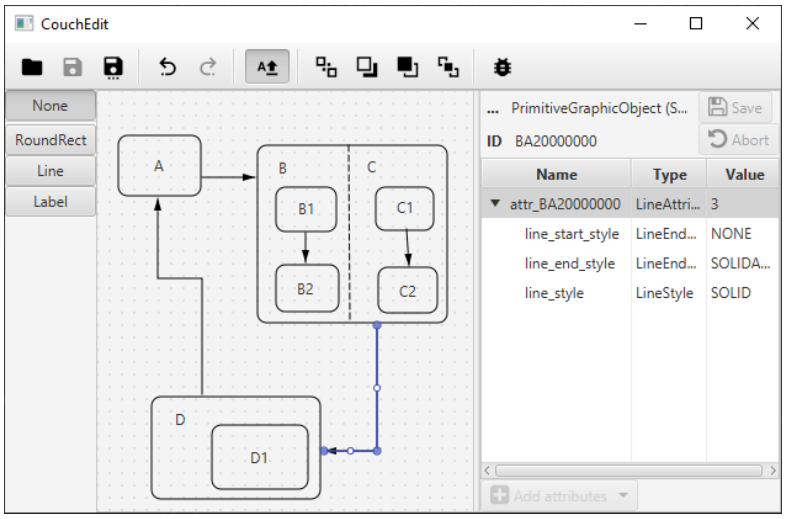
Figure 2.5: Depiction of COUCHEDIT's approach to separating graphical and abstract representation.

2.3 CouchEdit

As noted in Section 1.1, the COUCHEDIT framework builds on the ideas of Van Tendeloo et al. [2]. In their paper, the authors criticize that classic approaches to modeling frontends usually create one monolithic construct, which handles both the graphical model display and the corresponding abstract syntax. This creates tight coupling between the two, which causes low flexibility of the UI and high effort requirements to implement new features.

The authors propose to solve this by defining two metamodels, one describing the graphical syntax and one describing the abstract syntax. The *render syntax metamodel* specifies graphic primitives that a given graphical modeling syntax uses. The frontend manages instances of this metamodel, and a user can manipulate this concrete representation directly by adding, deleting, or modifying elements in the UI (if the frontend allows changes). On The other side stands the *abstract syntax metamodel*, it describes a modeling syntax's conceptual representation. The backend manages instances of this metamodel. A further metamodel, called *concrete syntax metamodel*, can then be defined to connect the render and abstract syntax metamodels. This concrete syntax metamodel specifies a connection that maps the concrete graphical representation onto an abstract representation (Figure 2.5).

Following this architecture, COUCHEDIT separates the frontend from the abstract syntax model. At its core, the frontend is a simple vector drawing application that implements further functionality for diagrams (Figure 2.6). A Backend then has the responsibility to analyze these vector graphics and map them onto a corresponding abstract syntax representation. For this, the backend utilizes a set of independent components, that each is responsible for detecting different aspects of the diagram. Each component contains a separate state. When a component makes changes to its state, it also publishes these changes so that other components interested in these changes can use them to calculate follow-up changes.



... PrimitiveGraphicObject (S...

ID BA20000000

Name	Type	Value
attr_BA20000000	LineAttri...	3
line_start_style	LineEnd...	NONE
line_end_style	LineEnd...	SOLIDA...
line_style	LineStyle	SOLID

Figure 2.6: Frontend of the current COUCHEDIT prototype configured for Statecharts

This architecture has the advantage of being very detangled. Therefore, it is possible to improve components in isolation from the rest of the system. The clear separation of concerns also makes it easier to understand which component is responsible for what task. Furthermore, components can be swapped out depending on the tasks the configured modeling syntax requires.

2.3.1 Data Model

COUCHEDIT uses a hypergraph of elements and relations to represent its data. Each component has its own instance of this graph that only contains the graph area relevant to this component. A component can manipulate its graph and make these changes known to the system so that other components can use them.

2.3.1.1 Elements

Elements are the base type of COUCHEDIT's hypergraph. Every type of node in this graph inherits from the element type. As all hypergraph instances should be independent of each other, it is impossible to access elements via their object reference. Thus every element has a unique id to identify it across multiple graphs reliably. Elements also have a *probability* attribute that denotes how probable it is that this element results from a correct interpretation of the given hypergraph. The probability attribute can also be set to *explicit*, which indicates that components should prioritize the corresponding element regardless of the probability value. Furthermore, element subtypes can introduce further attributes that contain important information for the given type.

2.3.1.2 Relations

Relations represent edges in the hypergraph. Each relation connects a set of source elements to a set of target elements. They also inherit from the element type and thus can recursively be connected with relations themselves. The connected vertices are referenced using the *ElementReference* class. This class holds an element's unique id and its type, which then can be used to find a referenced element in the hypergraph. Figure 2.7 shows the implementation of relations.

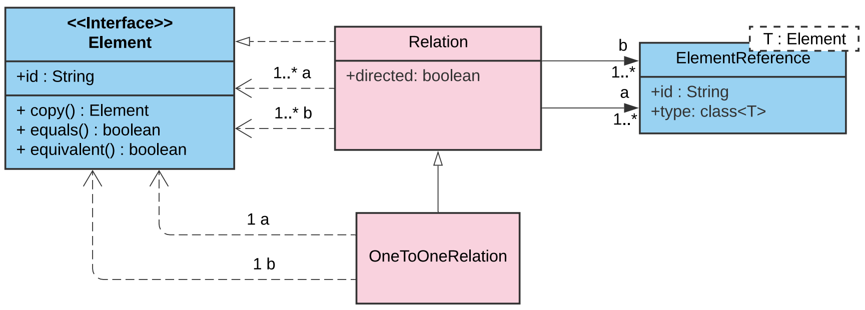


Figure 2.7: Class diagram of Relations

While relations can connect multiple source and target vertices, the most common relation type is the *OneToOneRelation*, it describes a connection between exactly one source and one target element. A typical example of such an OneToOneRelation is the *Contains* relation, which defines that one element encloses another.

2.3.1.3 GraphicObjects

GraphicObjects (GOs) represent atomic elements that the user interacts with via the frontend. When designing an editor, an important question is how granular a singular GraphicObject should be. On a basic level, GOs can consist of either pixels or vectors. A pixel-based approach would require a visual recognition preprocessor that composes the state of pixels into comprehensible graphic primitives. While such a form of image recognition is outside COUCHEDIT's scope, similar projects such as FLEXISKETCH [5], show this approaches' application.

On the other hand, most modeling approaches with abstract syntax processing build specialized graphic elements to represent a specific element of the abstract syntax. Therefore mapping the concrete syntax to an abstract representation is made easy. However, as a result, these graphic elements are stringent in their graphical representation.

COUCHEDIT's goal is to allow users to construct diagrams using elementary, "multi-purpose" GraphicObjects as well as specialized elements (if required by the modeling language) [1].

To this end, COUCHEDIT uses a render metamodel (RenderMM). It defines all building blocks that are usable in a concrete representation. COUCHEDIT builds its RenderMM upon the Classification of Costagliola et al. [6]. This classification defines a graphic element on its lowest level as a primitive graphic shape (e.g., circle, line). Via sub-classing, these primitive GOs can then be extended to form specialized, *ComplexGraphicObjects*.

In his work, The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering [7], Moody specifies eight variables that can be applied to a single graphic primitive. These attributes are separated into factors that directly influence the relations between GOs: *Position* (horizontal and vertical), *shape*, *size*, *orientation*, and attributes that only serve as visual hints and influence a single GO at a time: *color*, *brightness*, *texture*. COUCHEDIT realizes this separation as well. Factors with spatial influence are contained as attributes of the shape. While purely visual attributes are externalized into *AttributeBags*. An AttributeBag is a separate element attachable to GraphicObjects that contains secondary information about the given GO.

2.3.1.4 HotSpotDefinitions

Some modeling notations define relevant zones in a graphical syntax, which are not directly modeled by a GraphicObject and thus are not directly visible to the user but exist due to the visual composition. For example, Bottoni and Grau identify attachment points as such a relevant zone [8]. An attachment point defines an area around a GraphicObject where a connection line can be "attached". COUCHEDIT implements this in a more general form through *HotSpotDefinitions* (HSDs). A HotSpotDefinition is a relation that, similar to GOs, has a shape and can thus be a target and a source of spatial relations.

One of the primary applications of HSDs is in the form of *CompartmentHotSpotDefinitions* (CompHSDs). Compartments split a GraphicObject into multiple logical sub-areas. This is used in multiple advanced notations. For example, the Statechart syntax, discussed in Section 2.2.2, where a state can have multiple regions, marked by dashed lines.

2.3.2 Application Structure

COUCHEDIT builds around independent components, called *processors* that can be enabled or disabled as needed in a given use case. To this end, the centerpiece of COUCHEDIT’s architecture is the *ModificationBusManager*. Every processor (including the frontend) can be connected to this bus manager (Figure 1.1). If one processor publishes a hypergraph change, the ModificationBusManager propagates this change to all connected processors. Each processor can specify which element types it is interested in and will only receive updates for elements that match one of those types.

To communicate these changes in the system, COUCHEDIT uses *ModelDiffs* (diffs). Each diff represents an add, change, or delete action of an element in the hypergraph. This allows processors to calculate correct graph changes based on the incoming changes instead of reevaluating the complete graph. This was added to the system because Nachreiner suspected that reevaluating the complete state would cause much overhead [1]. It turned out that processors which have to calculate spatial relations significantly profited from diff based calculations [1]. However, calculating the correct follow-up state based on a set of diffs turned out to be highly error-prone. Thus it was suspected that it could be beneficial to reevaluate the complete graph when processing lighter tasks, such as abstract syntax processing [1].

2.3.2.1 Core Processors

COUCHEDIT implements a set of processors that are integral to most modeling syntaxes. These so-called *core processors* currently are:

SpatialAbstractor: This processor has the responsibility to calculate how GOs are positioned to each other. Possible position Relations include: *RightOf*, *BottomOf*, *Intersect*.

ConnectionEndDetector: The *ConnectionEndDetector* finds line GOs that could represent a connection to another GO. It then adds *ConnectionEnd* relations from the line to the other GO. Furthermore, the ConnectionEnd relation has the attribute *isEndConnection* that defines if this relation origins from the ending point of the line.

Containment: The *ContainmentProcessor* checks if one GO is contained by another. If one GO completely encompasses another GO, the processor adds a Contains relation from the surrounding GO to the contained one.

2.3.3 Services

Processors may sometimes require to gather additional information. For example, it is often needed to find all elements that are connected to a given element by a certain relation type. To this end, COUCHEDIT provides *services*. A service is a separate object that provides a set of functions. A service can be required by multiple different processors and thus must be effectively stateless.

3 Related Work

This chapter provides information about work in the area of graph translation and concrete to abstract syntax mapping. This primarily encompasses triple graph grammars, as well as work in the area of relaxed conformance modeling.

3.1 Triple Graph Grammars

Triple graph grammars (TGGs) were first introduced by Schürr in 1994 [9]. They formalize a specification that can define bidirectional translations between different graph languages [10]. TGGs consist, as the name implies, of three graphs, often called *source graph*, *target graph*, and *correspondence graph*. The source and target graphs are distinct graphs that are connected by the correspondence graph. The correspondence graph is responsible for realizing correspondence relationships between source and target graphs. Production rules can be defined for a given graph triple. Applying these rules to a given instance of the source graph creates a mapping to the corresponding target graph representation.

This triple graph structure is similar to the architecture employed by CouchEdit, explained in Section 2.3. Thus TGGs are a possible mechanism to specify configurations for CouchEdit. Furthermore, well defined TGGs can generate *forward and backward graph translations*. As CouchEdit strives to provide functionality that requires translation from render model to abstract model, and vice versa [1], the bidirectionality TGGs provide could be beneficial for future implementations. Nonetheless, the approach described in the following thesis does not make use of TGGs. TGG production rules are usually defined as a form of graph grammar rules [10]. In the early development stages of this research, it was evaluated that the most taxing aspect of translating concrete to abstract syntax is the recognition of correct concrete representations in the graphical instance. This means that realizing the translation from render metamodel to abstract metamodel using graph grammar rules is especially verbose on the left-hand side of the production rule.

Furthermore, [10] notes that most TGG implementations use inefficient graph grammar parser algorithms. As CouchEdit has to provide user feedback during the editing process, performance represents an essential aspect of production-ready implementations of CouchEdit. Nonetheless, TGGs could provide benefits for CouchEdit, and it may be worthwhile to investigate their applicability in future work.

3.2 Making Metamodels Aware of Concrete Syntax

In their work, Making Metamodels Aware of Concrete Syntax [11], Fondement and Baar argue that, while abstract syntax definitions are standardized, most language specifications keep the concrete syntax informal. To solve this problem, they propose an approach to defining the concrete syntax and how to link it to the abstract representation.

For this, the authors complement every class of the abstract syntax with a corresponding display scheme. This display scheme is composed of two parts, an iconic and a constraining part. The iconic part defines a set of *DisplayClasses*. These DisplayClasses group GraphicObjects together into visual representation objects. On the other hand, the constraining part links these DisplayClasses to an abstract syntax element. This link is realized using *DisplayManagers* (DM). A DisplayManager serves as a connection between exactly one model element of the abstract syntax and one DisplayClass and syncs the abstract to the concrete representation. Figure 3.1 depicts an example of this architecture for a Petri nets place element. The graphical primitives a place is composed of are mapped to a place DisplayClass, to build a place's iconic part. This iconic representation is then attached to a place model element, using a place DisplayManager.

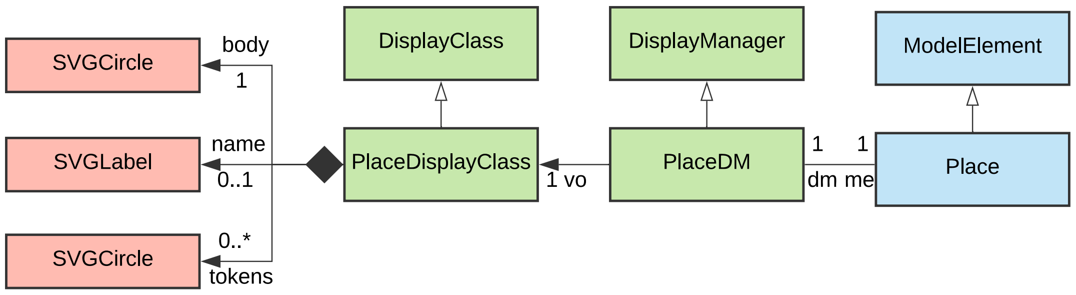


Figure 3.1: Example representation of a Petri nets place element

A DisplayManager has to keep abstract and concrete representation in sync. For this, Fondement and Baar utilize OCL invariants, which are defined on the DisplayerManager. For example, an invariant to sync the name of place display schemes could look as follows:

```
context PlaceDM
inv: self.me.name->exists() implies
        self.me.name = self.vo.name.text
```

Listing 3.1: OCL Invariant that syncs the name attribute of place DisplayClasses and place model elements.

The authors do not specify how the mapping from graphical primitives to a display object could be implemented. While the metamodel proposed in this thesis does not introduce an extra layer of abstraction in the form of these DisplayClasses, it still is inspired heavily by the work of Fondement and Baar. The constraining part that utilizes DMs to sync abstract and concrete syntax served as the fundament for the proposed architecture.

In his work "Correctly defined concrete syntax" [12], Baar furthermore provides an algorithm to check syntactic correctness of the concrete representation. While checking the concrete syntax for correctness is out of this work's scope, a future implementation would have to provide the user with syntax checking capabilities to ensure that the model, designed in the editor, is syntactically correct.

3.3 DiaGen

DiaGen is a project first introduced by Minas and Viehstaedt [13]. It is a system that allows for the specification and generation of modeling editors. Figure 3.2 depicts a Petri nets editor generated with DiaGen. DiaGen supports *free-hand editing*, where the user can manipulate the graphical representation directly. This is the same editing mode that CouchEdit supports [1].

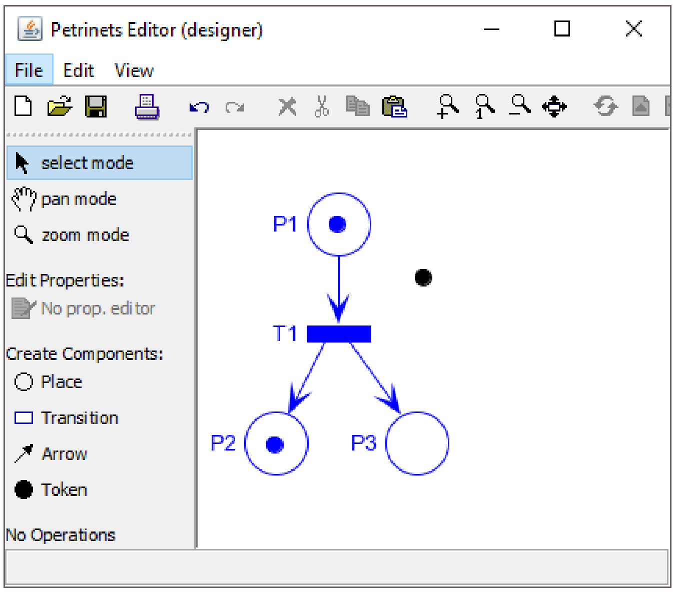


Figure 3.2: Editor generated with DiaGen that can model Petri nets. Elements that are part of a concrete abstract representation are colored in blue.

In DiaGen, a diagram consists of a finite set of diagram components [14]. Each component is defined by its attributes. Furthermore, each component has a set of *attachment points*.

Attachment points define areas at which different diagram components can be connected. DiaGen's architecture generates a hypergraph from the visual diagram consisting of these diagram components. This hypergraph is furthermore reduced using a context-free grammar to receive the abstract representation. DiaGen also supports automatic layouting and user feedback on diagram parts that are not correct under the defined diagram language [14]. DiaGen provides a textual language and a GUI tool, which can be used to specify editors.

While DiaGen provides an interesting example of a textual language to define modeling syntaxes, it does not seem applicable to CouchEdit. DiaGen does not create a hypergraph that contains concrete and abstract syntax. Instead, it parses the concrete syntax to an intermediate hypergraph, which is then reduced to the abstract syntax. Furthermore, as a code generator, DiaGen relies on the possibility to implement certain parts of the definition in source code. While the artifact proposed in this thesis also includes a code generator that uses this advantage, it is ultimately intended to evolve this work into a runtime interpreter that does not necessarily rely on source code implementations.

4 Design Concept

Section 2.3 established that CouchEdit is composed of three distinct metamodels, as depicted in Figure 2.5. Thus, it seems apparent that the proposed concept should also be composed of these three metamodels. The definition of abstract syntax metamodels has already standardized approaches (e.g., Ecore[1]) and thus is mostly ignored. Furthermore, the proposed design currently only supports a render metamodel definition composed of graphic primitives. Follow up work would have to explore how ComplexGraphicObjects integrate. This work focuses on the concrete syntax metamodel that connects render and abstract syntax and proposes an approach to formalize this connection. Figure 4.1 shows the root definition of the CouchEdit configuration metamodel. Throughout the chapter, this root model is gradually being extended with new concepts, until finally, all utilized concepts have been introduced (the complete metamodel is depicted in Appendix A).

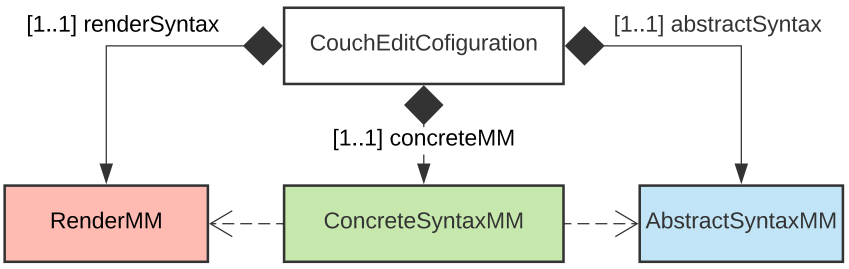


Figure 4.1: Root of the CouchEdit configuration metamodel

The primary goal of the proposed design is to hide CouchEdit's complexity behind more accessible metamodel definitions. To this end, multiple concepts are employed that either abstract away from CouchEdit's implementation details or introduce ways to streamline the translation from concrete to abstract representation. Most of these concepts could introduce performance overhead, but as this work focuses on the design aspect, performance analysis and optimization is subsidiary and subject to further research.

[1] `https://www.eclipse.org/modeling/emf/`

4.1 Render Metamodel

COUCHEDIT's render syntax is realized as a set of GraphicObjects. The framework in its current form primarily focuses on primitive GraphicObjects, with its class structure being derived from Bottoni and Grau's work, "A Suite of Metamodels as a Basis for a Classification of Visual Languages" [1], [8]. While the notion of ComplexGraphicObjects has been introduced in COUCHEDIT's formal definition, the concept for now only exists in theory. Because of time constraints, this thesis does not explore the definition of the concrete syntax all too much and instead focuses only on graphic primitives. Figure 4.2 describes the employed render syntax metamodel. As shown, it facilitates a basic structure to define graphic primitives needed for the described visual language. Future work would have to explore how graphic objects could be customized, by integrating AttributeBag definitions and complex graphic object definitions, to create custom symbols better suited for specific modeling languages.

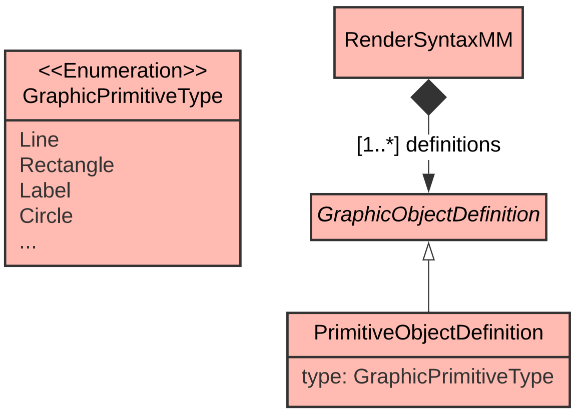


Figure 4.2: Render syntax part of the COUCHEDIT configuration metamodel

4.2 Abstract Syntax

As noted before, the abstract syntax can be defined using existing standards. Nonetheless, the translation metamodel needs to reference the abstract syntax at multiple points in this chapter. Therefore, a barebones specification that is derived from Ecore's metamodel[2] definition is given here (Figure 4.3). Notable concepts needed in the definition of an abstract syntax, are foremost, the definition of classes. This includes the definition of class attributes and concepts such as inheritance and references to other classes. Furthermore, definitions of simple data types, as well as enums, are required.

[2]`https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html`

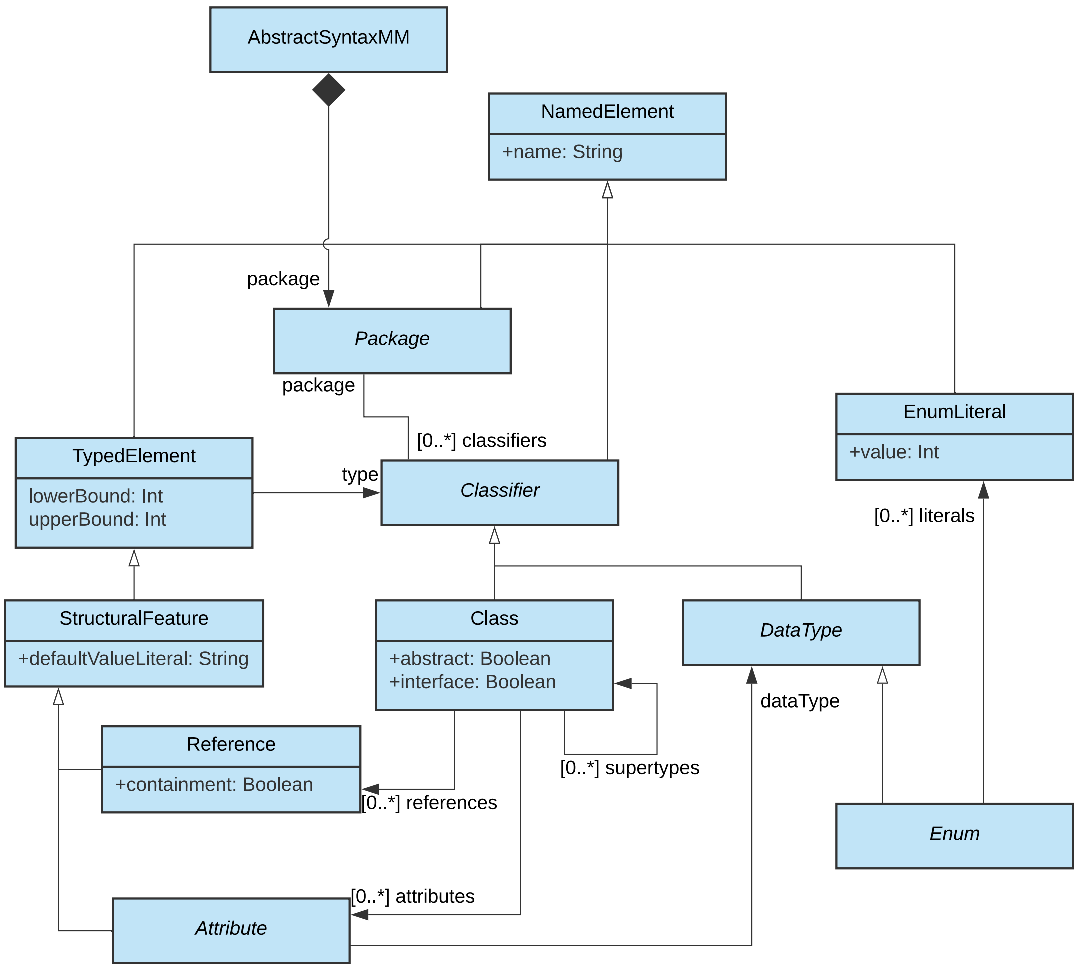


Figure 4.3: Abstract syntax part of the COUCHEDIT configuration metamodel

4.3 Language Concepts

The abstract syntax tree usually differs between programming languages. Therefore, when looking at lower-level concepts found in most programming languages (e.g., arithmetic, branching), their metamodel representation varies. For this reason, and because there is already enough documentation on how to specify abstract syntax trees, common language concepts will not be defined, as this depends on the actual language implementation. Instead, at points where these language concepts would reside, it is only defined what the given metamodel expects from the implementation, and example code snippets are given in an unspecified pseudo-language. Despite this, certain concepts are unique to the COUCHEDIT metamodel, which a language implementation would have to realize. These concepts are presented in the following subsections.

4.3.1 Helper Functions

When defining a translation metamodel, it may become useful to relocate specific procedures or algorithms into separate functions. Reasons for this may be to access it from multiple points in the definition, to implement a recursive structure or, simply to split up complex parts. Thus the metamodel should provide a facility for the definition of custom functions. That being said, the actual metamodel definition is kept ambiguous, as details depend on the given implementation. In the architecture described here, these helper functions are defined on the metamodel's highest level and thus are available in all statement blocks. However, it would also be feasible to move function definitions into sub scopes, should that provide advantages for certain implementations.

4.3.2 Graph Navigation

Something used a lot in the proposed architecture is traversal through the hypergraph. It should be possible to get nodes that are adjacent to any given node easily. In COUCHEDIT's architecture, this is handled by services (Section 2.3.3). This means that the syntax for reaching a node related by a given relation type looks something like this:

```
val containedGOs = relationService
      .getAllElementsRelatedFrom(go, Contains)
```

Listing 4.1: Example of how to get all GOs contained by a given element `go` in the COUCHEDIT architecture

The `getAllElementsRelatedFrom` function takes a GraphicObject `go` and a relation type and returns all GOs that have a relation of the given type, pointing from `go` to themselves. This and similar helper functions are used repeatedly. Thus DSL implementations of the here described architecture should provide some form of abstraction to this service call. In the following code snippets, it is assumed that these functions are masked by member function implementations similar to the following:

```
go.getAllElementsRelatedFrom(Contains)
```

Listing 4.2: The `getAllElementsRelatedFrom` function, implemented as a member function, eases navigation through the hypergraph.

Implementation in the form of such extension functions allows for more natural navigation through the hypergraph, which is especially useful when traversing longer distances, which will become apparent in the following sections. Appendix B contains a list of these navigation functions.

4.4 Syntax Processors

When defining a modeling language with a clear separation of concrete and abstract syntax, the primary concern is how to connect these two distinct models. At this point, the designed architecture draws inspiration from Fondement and Baar [11]. The author's proposed idea of connecting abstract and concrete syntax using DisplayManagers serves as a basis. This approach consists of two parts, recognition and synchronization.

4.4.1 Recognition

The recognition part is concerned with detecting patterns in the graphical syntax that have an abstract syntax representation. For this, Fondement and Baar proposed the introduction of a further abstraction layer. This layer composes the graphic primitives and attributes, which represent a model element, into DisplayClasses. However, the authors keep a possible implementation of this abstraction layer open. Furthermore, it did not seem suitable to introduce a further abstraction layer, as most processing in the COUCHEDIT architecture is applied directly to the concrete syntax hypergraph. Thus mapping GraphicObjects to a new layer would introduce a set of new problems to be solved. Instead, the architecture proposed here uses a different approach. For each type of DisplayManager, a set of *constraints* can be defined that a GraphicObject has to meet. Whenever a GO satisfies all constraints of a given DM type, it is deemed the base element of a concrete representation of this DM, and an instance of the DM and corresponding model element are created and connected to the GraphicObject (Figure 4.4).

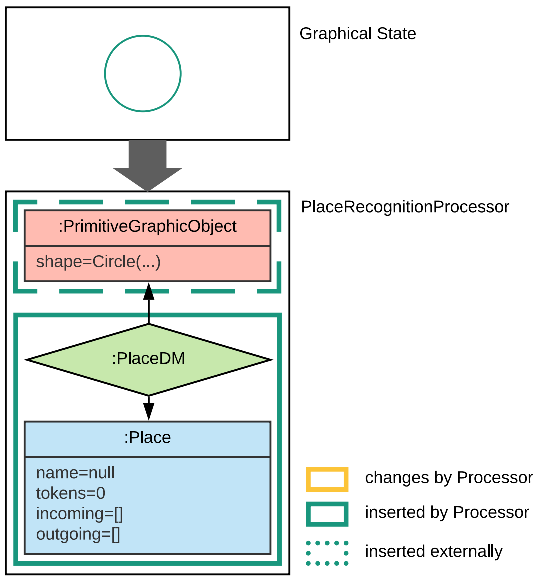


Figure 4.4: PlaceRecognitionProcessor adds PlaceDM to a GraphicObject that satisfies constraints

These constraints can be defined as simple boolean expressions that each GO in the hypergraph is checked against. The constraints for a place element could be defined as follows:

```
go.shape == circle
go.allRelatedTo(Contains).isEmpty()
```

Listing 4.3: Possible constraints to detect GOs representing a place

These constraints first check if the given GO has the shape of a circle. If that is true, it is also checked if the given GO has any `Contains` relations pointing toward itself. If that is the case, the given circle is contained by another element and thus not clearly identifiable as a place, as it could also possibly represent a token. A corresponding definition of transitions could look as follows:

```
go.shape == rectangle
```

Listing 4.4: Simple constraint to check for transition representations

These are barebones requirements to identify graphical representations, and they could be extended by any amount of further constraints to increase the amount of specificity required from the concrete instance.

4.4.2 Synchronization

With the recognition part in place, it is possible to detect graphical representations and attach corresponding abstract instances. Now the synchronization part is responsible for making sure abstract and concrete representation reflect the same state. To this end, the metamodel allows for the definition of rules on a DisplayManager. This step was explained in detail in [11]. Fondement and Baar propose the usage of OCL invariants as a language for defining these rules. As this research does not provide a DSL implementation, simple pseudo-code assignment statements are used here. For a place element, four aspects have to be synced:

1. Name of the given place
2. Token count
3. Incoming transitions
4. Outgoing transitions

Reliably determining a place's name poses some special challenges and requires further concepts introduced later. Determining the token number, however, is easily implemented using the given tools. A simple rule that syncs this attribute could look something like this:

```
dm.me.tokens = dm.go
                   .allRelatedFrom(Contains)
                   .select(go -> go.shape == circle)
                   .count()
```

Listing 4.5: Rule that syncs the token count of a place element

This statement ensures that the token attribute of the model element is always equal to the number of all GOs with the shape circle that are contained by the base GraphicObject. Similarly, incoming and outgoing transitions can be defined:

```
dm.me.incoming =
      dm.go
          .allRelatedTo(ConnectionEnd,
              rel -> rel.isEndConnection)
          .select(go -> go.shape == line)
          .collect(go -> go.relatedFrom(ConnectionEnd))
          .select(go -> go.shape == rectangle)
          .select(go -> go.dm != null)
          .collect(go -> go.dm.me)
```

Listing 4.6: Rule that syncs incoming transitions of a place element

This statement ensures that all transitions connected to the given DM's GO, by a line, are composed into the list of incoming transitions (Figure 4.5). This exhibits the usual approach to syncing concrete and abstract representation.

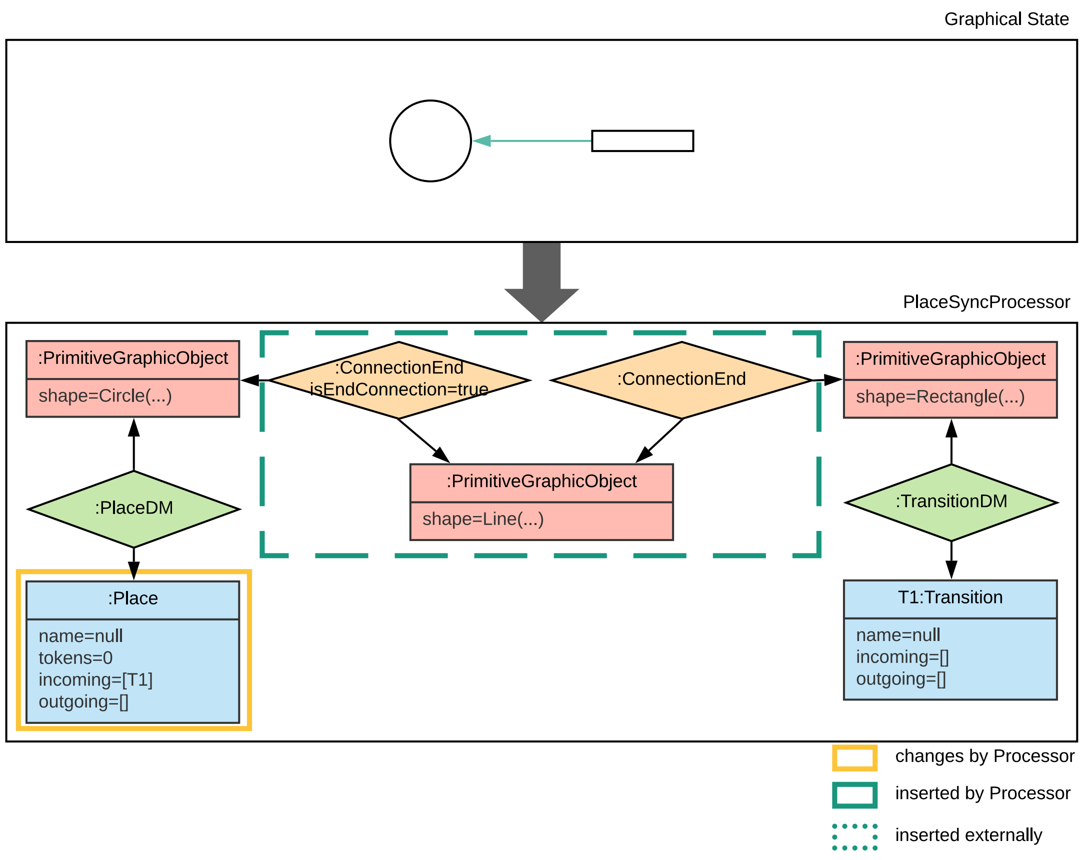


Figure 4.5: A connection line is added to the graph, and the `PlaceSyncProcessor` updates the `Place` model element

4.4.3 Meta-Model

The resulting sub-model for this area of the configuration would look as described in Figure 4.6. The depicted metamodel allows for the definition of DisplayManagers. A DisplayManager definition needs a reference to a model element type from the abstract syntax metamodel. The name of the DM is inferred from this model element. Furthermore, the DM definition needs constraints and rules. The constraints are then used by a *RecognitionProcessor*, to add instances of the inferred DM. On the other hand, rules are used by a *SyncProcessor*, to keep model elements aligned. This is not the final iteration of this part of the metamodel, as will be shown in the next section.

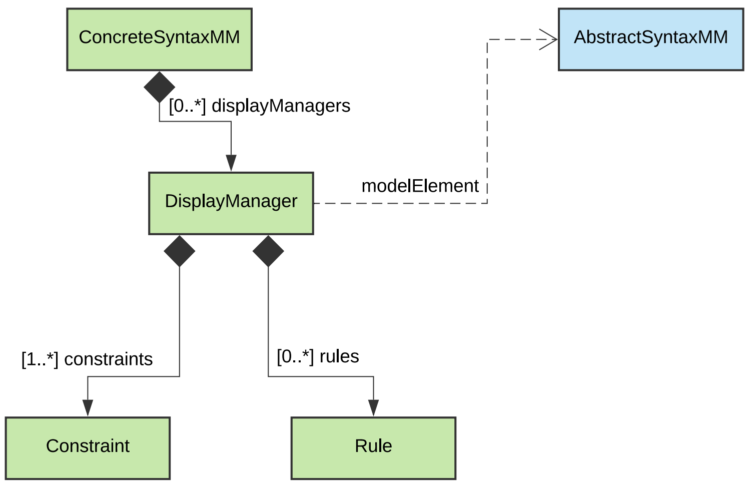


Figure 4.6: First design of the DisplayManager part of the metamodel

4.5 Pattern System

When assessing the incoming transition rule (Listing 4.6), it becomes apparent that checking if the connected GO represents a transition has to be done manually. In the example given, checking if the GO represents a transition is no big task as transition recognition is handled with only one constraint (Listing 4.4). However, when regarding model elements with multiple constraints, this can become a repetitive and inefficient task.

The proposed architecture introduces a *pattern* system to alleviate this problem. A pattern can mark elements of the graph as satisfying a certain set of constraints. A pattern is similar to AttributeBags, a separate element in the graph that is connected to the element it describes, with a relation. It has a single value attribute that indicates its name. Identical to DM definitions, described in the last section, pattern definitions take a set of constraints. Every element in the graph is then checked against these constraints, and if an element satisfies all of them, the pattern is added to the given element.

The DisplayManager recognition part's similarity means that it can be replaced by the Pattern system. Instead of checking constraints themselves, RecognitionProcessors just check if a given GO has a specified pattern type attached, and if this is true, the corresponding DM instance is added. As shown in Figure 4.7, when a GO representing a transition is added, the `TransitionPatternProcessor`, first adds a `PatternType` with the value `Transition`. The `TransitionRecognitionProcessor` then checks the GO, which now matches the `Transition` pattern and thus adds a `TransitionDM`.

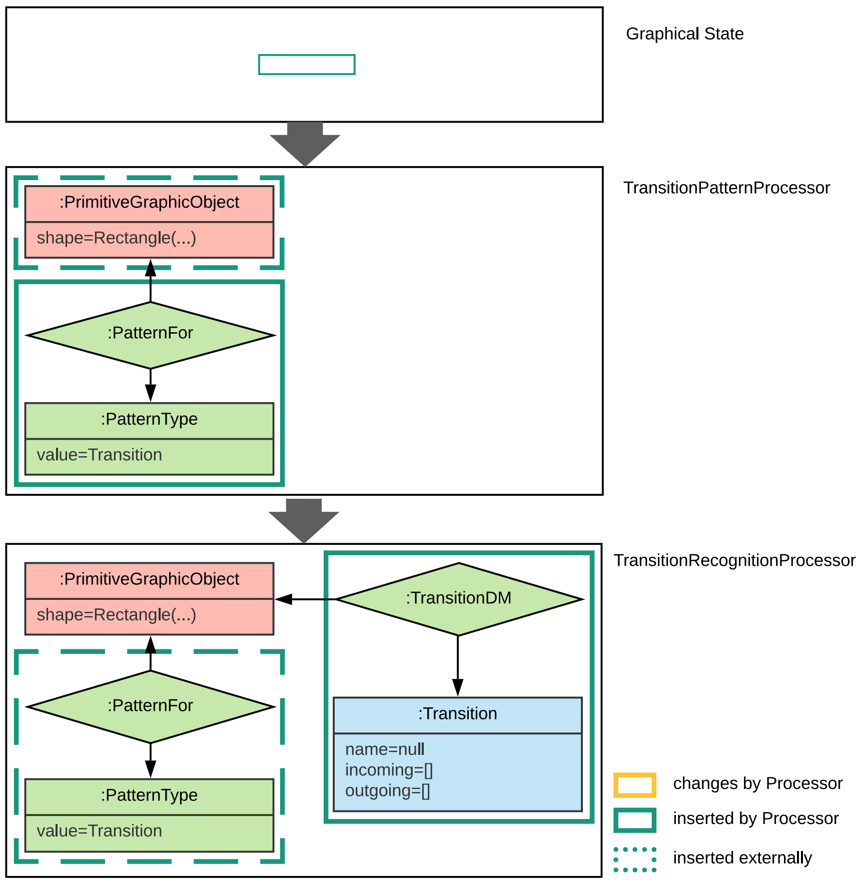


Figure 4.7: `TransitionRecognitionProcessor` adds DM transition `PatternType` was added.

With the definition of a convenience member function that checks if an element has a given pattern, the rule, syncing incoming transitions, can now be rewritten as follows:

```
dm.me.incoming =
    dm.go
        .allRelatedTo(ConnectionEnd,
            rel -> rel.isEndConnection)
        .select(go -> go.shape == line)
        .collect(go -> go.relatedFrom(ConnectionEnd))
        .select(go -> go.hasPattern(Transition))
        .collect(go -> go.dm.me)
```

Listing 4.7: Improved incoming transition rule, which also filters for elements that match the `Transition` pattern.

It is important to note that the pattern system is not exclusive to GOs that have an abstract representation. For GOs with a corresponding DM attached, the DM type can just be checked to find out if it connects model elements of a particular type. Instead, the pattern system can also be used to recognize all sorts of patterns in the graph that would otherwise have to be checked repeatedly. For example, it could be used to define a stricter check on what element in a Petri net graph represents a token.

4.5.1 Meta-Model

As mentioned above, because of the similarity of the constraint mechanism in the pattern and DM systems, patterns can be used to handle the recognition of DMs. This means the metamodel that was introduced in the last section has to be revised. Figure 4.8 shows this new part of the metamodel extended by pattern definitions. In this revised version, DM definitions have a reference to a pattern definition.

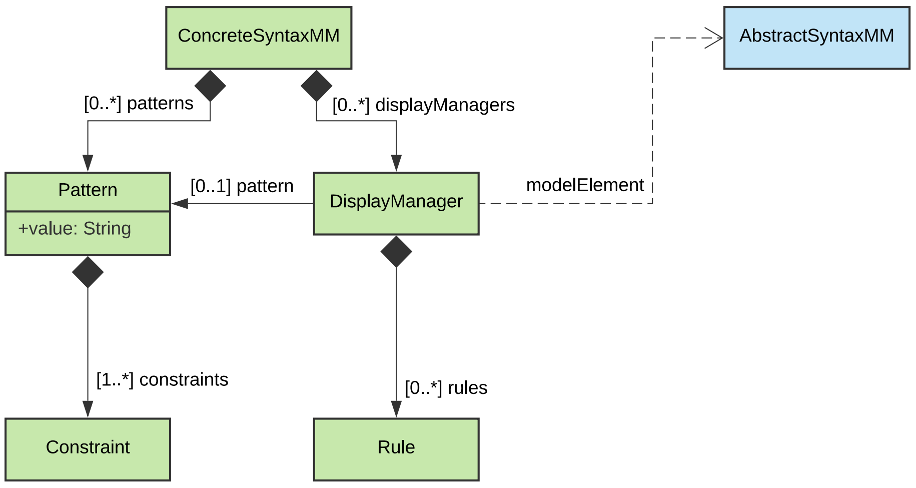


Figure 4.8: Revised SyntaxProcessor metamodel that now includes the Pattern system

4.6 Plugins

While the rule for incoming transitions (Listing 4.6) is sufficient for an initial example, it has two significant flaws. Firstly, it is not checking if the connecting line has an arrow end. Secondly, only lines drawn *from* the transition *towards* the place are treated as incoming lines, as noted in Section 2.3.2.1. Fixing these issues in the form of a rule would be too verbose for such a typical pattern.

For this reason, the plugin system is introduced. Plugins are predefined processors that process specific parts of the hypergraph. These processing areas are not as integral as the ones solved by the core processors and thus are *opt-in*. Furthermore, certain plugin processors can be configured, which allows them to cover a wider field of use cases. One example of such a plugin would be the *ConnectionProcessor*. It looks for line GOs and checks if they connect two other GOs. If that is the case, the processor adds either a *ConnectionTo* relation or a *ConnectionBetween* relation (Figure 4.9), depending on if the line's arrow ends indicate a directed or undirected connection. Adding this plugin allows for a final rewrite of the rule, syncing incoming transitions:

```
dm.me.incoming =
    dm.go
        .allRelatedTo(ConnectionTo)
        .select(go -> go.hasPattern(Transition))
        .collect(go -> go.dm.me)
```

Listing 4.8: Final iteration of the place incoming transition rule

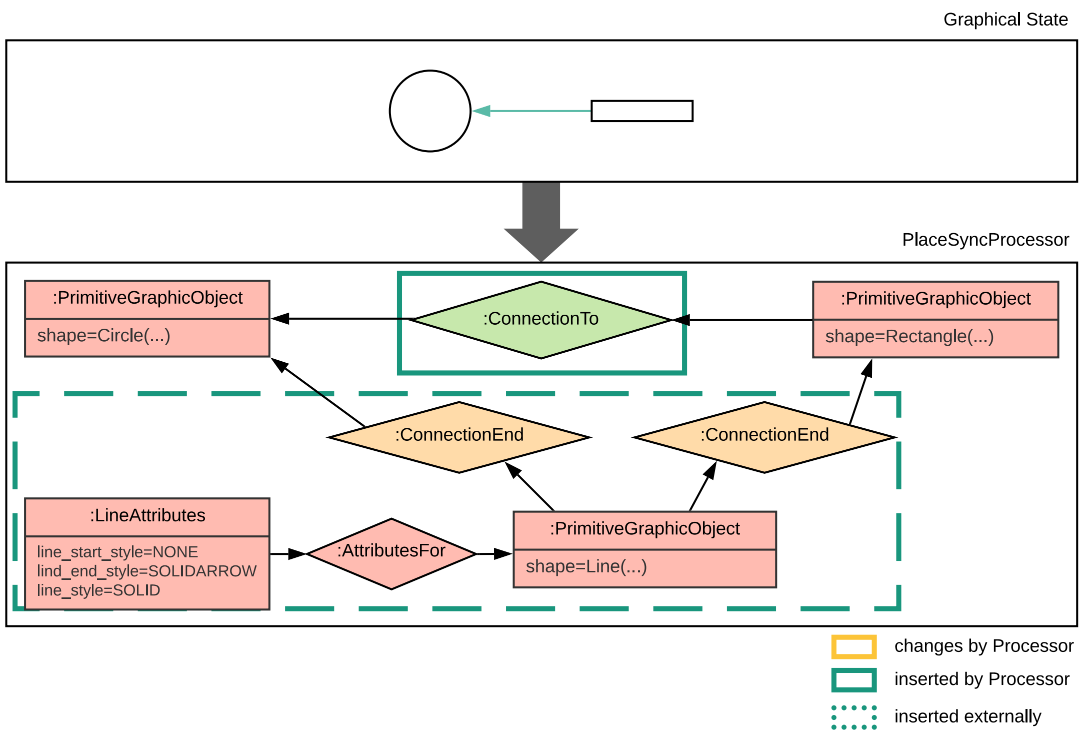


Figure 4.9: `ConnectionProcessor` adds `ConnectionTo` relation on detection of a directed line

4.6.1 Label Processor

Defining a transformation to sync a place's name has been postponed until now because it poses multiple new challenges. When determining if a label represents the name of a place element, multiple aspects are of importance:

1. How close is the label to the place?
2. Are different GOs in the proximity that could be associated with this label?
3. Are there other labels that could represent the name of this place?

While possible, checking all these conditions in a rule would require several dozen lines of code. Furthermore, it would cause much overhead. For every place, whenever syncing its state, the relation possibility to all labels has to be calculated. If one label is a potential candidate, it also must be calculated if the label could be a possible candidate for another element. Thus, the idea of building a plugin that handles these calculations arises.

The basic idea of the *LabelProcessor* is to add *LabelFor* relations from labels to other GraphicObjects. Furthermore, the probability attribute is utilized to define how probable a given LabelFor relation is. For such a plugin to be applicable to multiple concrete syntaxes poses a set of new requirements:

Requirement 1: Modeling syntaxes require different spatial relations between a label and GO (the label can be contained, surrounding, above/below, etc.).

Requirement 2: Different GO shapes require different algorithms to calculate the probability (proximity to a circle cannot be calculated the same way as proximity to a rectangle)

Requirement 3: Not every GO needs a label. For example, in the Petri nets syntax, only GOs representing either a place or transition require a label.

Requirement 4: Some label relations can take precedence over others. For example, in Statecharts, a label contained by a simple state cannot describe a transition trigger event.

4.6.1.1 Label Sub Processor

The LabelProcessor implements a set of sub-processors. Each sub-processor is responsible for different types of LabelFor calculations. For example, the *CircleOuterLabel* sub-processor calculates the LabelFor relation between a circle GO, and a label GO placed somewhere around it. This means that a different sub-processor is required for every combination of primitive shape and spatial relation. Each sub-processor takes a pattern type and a label. It then finds all GraphicObjects with its designated shape type and filters them by the given pattern type. The resulting GOs then receive a LabelFor relation from the given label, with a probability calculated by an algorithm implemented in the sub-processor.

4.6.1.2 Label ProcessingChain

When regarding requirement 4, it becomes clear that simply enabling the sub-processors needed for a given modeling syntax is not enough. A mechanism is needed that allows for different sub-processors to influence each other. Therefore the concept of a *processor chain* is introduced (Figure 4.10). A processor chain allows for the configuration of sub-processors by chaining them together with different conditions. Processor chains can either be *operations* or *terminals*. Terminals represent either a sub-processor or nothing. An operation, on the other hand, defines a link between two processor chains. Operations possess an *operator* that defines how the left-hand processor chain results should affect the results of the right-hand one. For example, the 'and' operator defines that both processor chains are calculated, and their results are aggregated. On the other hand, the 'ifEmpty' operator only calculates the right-hand processor chain if the left-hand one has returned no values. The 'ifAmbiguous' operator ignores the left-hand side if it returns more than one relation and instead calculates the right-hand side. Using this processor chain, the label processing for Petri nets could be configured as follows:

```
(CircleOuterLabel(Place) and
RectangleOuterLabel(Transition)) ifAmbiguous Nothing
```

Listing 4.9: LabelPlugin configuration for a Petri nets syntax.

This causes the LabelProcessor to calculate outer proximity LabelFor relations for all circular GOs with pattern type `Place` and rectangular GOs with pattern type `Transition`. Should this calculation return more than one possible relation for a given label, the relations are ignored, and nothing is returned instead.

This LabelProcessor plugin is by no means a perfect implementation for the given problem. Instead, it provides a general idea of the purpose and applicability of the plugin system and how specialized the definition of plugin configurations can become. The plugin configurations' requirements force DSL implementations of this metamodel to be updated every time a new plugin with custom configuration is added to the system. Alternatively, an approach for more general plugin configuration has to be explored.

4.6.2 Summary

In this chapter, the developed metamodel was introduced step by step. The base idea for this metamodel was derived from Fondement and Baar [11]. This concept is split into two parts, recognition and synchronization. Recognition is handled by the pattern system, which is responsible for detecting elements in the graphic representation that satisfy a defined set of constraints. The synchronization part is responsible for connecting graphic representations that satisfy a given pattern to a corresponding abstract representation. This system is extended by a plugin system that allows additional predefined processors to be enabled.

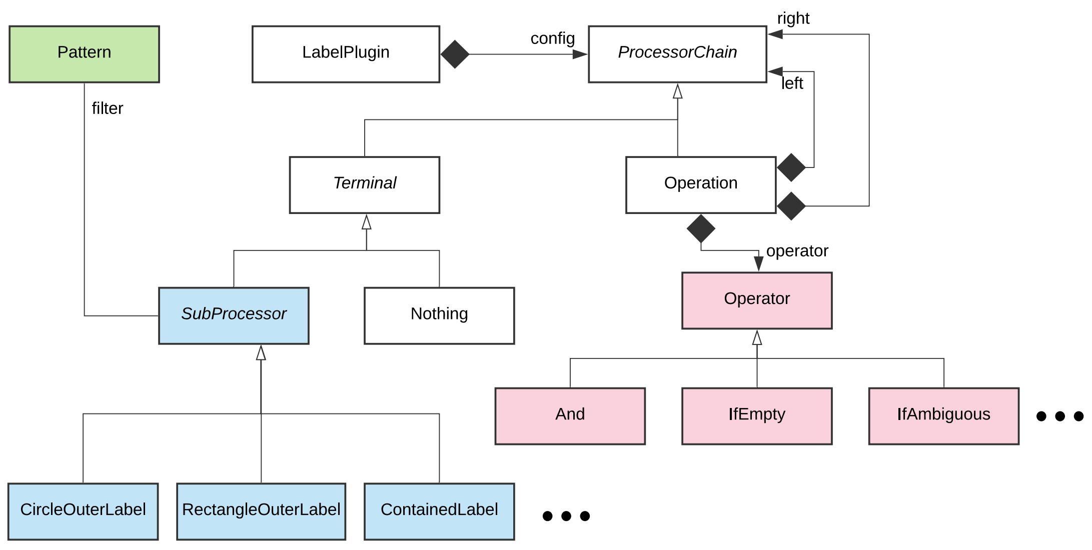


Figure 4.10: Class diagram describing the LabelProcessor’s configuration metamodel

These plugin processors can require configurations that allow a plugin to be applicable to a broader range of modeling languages. The complete metamodel is depicted in Appendix A.

5 Prototype

Alongside the development of the metamodel, described in Chapter 4, a prototype was developed, which serves as proof of concept for design decisions. This prototype is realized as an extension of COUCHEDIT's current existing prototype, developed by Nachreiner and presented in [1].

COUCHEDIT's current running prototype is wholly written in Kotlin[1], a programming language with the ability to compile to the Java virtual machine platform. Thus, Kotlin enables access to Java's ecosystem of libraries and frameworks. TornadoFX[2], a JavaFX wrapper, is used as the UI Framework. As a modeling tool, GEF[3] was integrated. Dependencies are handled using the dependency injection framework Guice[4], which integrates perfectly with COUCHEDIT's modular architecture. Detailed information about the current prototype and its features are presented in [1].

In theory, integrating COUCHEDIT with a parser that can evaluate language configurations at runtime would be preferable. This would make it possible to hot-swap configurations while the software is running. Instead, the prototype was extended by a code generator. This has multiple reasons. The current implementation is relying on the existence of class definitions. Meaning, at many points in the code, object types are checked. A runtime parser would have to either create workarounds for these type checks or compile the needed classes at runtime, using byte code manipulation. A code generator can generate all class definitions directly and thus solves this problem without any extra effort.

Furthermore, a code generator can work with a metamodel that is not fully defined. As noted in Chapter 4, low-level language concepts were not fully defined, as they depend on the actual language implementation. Therefore, the implemented parser accepts strings at places where the metamodel is not entirely defined. Statements and expressions can simply be written directly into these strings, which are then placed into the generated code without further analysis of their correctness. This means that these strings have to contain Kotlin code. Furthermore, needed language concepts (Section 4.3) have to be provided by the Kotlin implementation itself.

The code generator was implemented as a separate sub-module of the existing prototype. It contains its own application entry point that, when executed, will generate a new module. This module provides the language configuration, which is loaded by the model application

[1]`https://kotlinlang.org/`
[2]`https://tornadofx.io/`
[3]`https://www.eclipse.org/gef/`
[4]`https://github.com/google/guice`

when started, and automatically configures the application for this modeling syntax. For code generation purposes, FreeMarker[5] is used. FreeMarker is a language-agnostic template engine and therefore allows for the generation of Kotlin code out of the box.

It was also decided to implement an internal DSL, using Kotlin's integrated capabilities[6]. While not as versatile as external DSLs, Kotlin's capabilities are sufficient to define lightweight DSLs. They are defined directly in Kotlin source code and thus do not require any extra DSL parsers. Therefore, they are easy to implement while still providing a suitable syntax for defining instances of the developed configuration metamodel.

5.1 Kotlin DSL

The COUCHEDIT DSL comprises three distinct parts, each representing one of the sub metamodels defined by the COUCHEDIT configuration. Each COUCHEDIT configuration consists of three files, `RenderMM`, `ConcreteMM`, and `AbstractMM`, describing each of the three sub-metamodels the COUCHEDIT configuration encompasses.

5.1.1 Render DSL

Given the simplicity of the RenderMetamodel defined in this work, the corresponding DSL does not encompass much grammar. The `RenderMM` DSL provides a set of constants, each representing a shape supported by the framework. A configuration takes a set of these constants. Future work would have to look at how graphical attributes work with the system and the integration of complex graphic objects. An example of such a `RenderMM` instance for a Petri nets configuration is shown in Listing C.1.

5.1.2 Abstract DSL

It was noted in Section 4.2 that the abstract syntax metamodel could be defined using existing tools such as Ecore. As developing a functioning Ecore parser was deemed too complicated, the prototype instead implements a barebones grammar that allows for the definition of a subset of the Ecore metamodel. Notably, the DSL allows for the definition of enums and classes. While interface definitions are worthwhile as well, they are not yet supported.

StructuralFeatures allow for the definition of attributes a class possesses. Most notably, they allow for the definition of attributes and references to other classes. A StructuralFeature holds a name, and upper and lower bound definitions, where an upper bound of '-1' indicates that it is unbound. Furthermore, a StructuralFeature can hold a default value. The DSL allows for the definition of StructuralFeatures in the form of three symbols, `attr`, `ref`, and `comp`. `attr`

[5]https://freemarker.apache.org/

[6]https://kotlinlang.org/docs/reference/type-safe-builders.html

is used to define simple attributes, a class definition can hold. Currently, supported attribute types are: `boolean`, `string`, `integer`, `double`, and `object`, as well as any enums defined. `ref` and `comp` can be used to define references to other classes. Because of limitations imposed by the Kotlin DSL, enum as well as class reference names have to be wrapped into string literals. While `ref` defines standard references, `comp` defines compositions. This plays an important role when generating source code from this configuration. References define connections to objects that can exist by themselves. In the COUCHEDIT architecture, that means they have to exist in the application's hypergraph. Thus they can only be referenced using ElementReferences (Section 2.3.1.2). On the other hand, composition objects do not exist as separate entities in the graph and must not be referenced using ElementReferences. The `AbstractMM` definition for Petri nets is shown in Listing C.2.

5.1.3 Concrete DSL

The concrete syntax metamodel definition is the centerpiece of the developed artifact. Thus defining this part was the main focus during the development process. An example definition for a Petri nets `ConcreteMM` configuration is shown in Listing C.3. Similarly to the metamodel defined in Chapter 4, the `ConcreteMM` DSL comprises three distinct concepts, with an additional possibility of defining helper functions. Each of the concepts is defined in its own scope.

Helpers: The `helpers` scope allows for the definition of helper functions. `helper` definitions take a name, multiple arguments, as well as a body. A function's return value is determined by its body's last statement, as is done in all following statement blocks. Usually, Kotlin can infer the type of this return value by itself. However, in the case a recursive function is implemented, the return type has to be specified as Kotlin will run into recursive type checking problems. Because of the nature of a variable argument count, introduced by `param` definitions, body and return type have to be specified using Kotlin's named parameter syntax[7]. To reduce the overhead of implementing functions, parameters, and the function body, have to be written in string literals. Thus these values can then later be inserted into the generated Kotlin source code. This enables access to the full Kotlin functionality without having to do any extra work. On the flip side, that means an IDE cannot give any syntax checking for code segments encoded into string literals.

Plugins: The `plugins` scope allows available plugins to be enabled. The plugins scope, similar to the `RenderMM` DSL, provides a set of constants, each representing an available plugin. Furthermore, plugins that require additional configuration have their own scope, which provides the grammar required to specify a configuration for this plugin.

Patterns: The `patterns` scope allows for the definition of new pattern types. Each new pattern type is marked by a `def` block that contains the pattern's name, followed by a new scope. The scope contains all constraints that a GO has to satisfy in order to be marked with this pattern. A constraint is identified by a '+' followed by a string literal. The string literal

[7] `https://kotlinlang.org/docs/reference/functions.html#named-arguments`

represents a complete statement block and can contain multiple statements where the last statement represents the return value. These constraint blocks must return a boolean value and, thus, have to always end in a boolean expression. Furthermore, the constraints are completely isolated from each other. This means, casting the GO in one constraint will not affect the following constraints. On the other hand, these constraints are executed in the order they are defined. If one constraint fails, the following constraints will not be checked. This offers the possibility to safely cast a GO in all constraints when a previous constraint has verified the object's type.

DisplayManagers: The `displayManagers` scope allows for the definition of new Display-Managers. A `def` block marks a new DM definition, which contains a string literal specifying the abstract model element and possibly one literal specifying the corresponding pattern. Rules are defined in the same way as pattern constraints are. A rule's statement block, on the other hand, has no specific return type. Instead, they directly act on a given DisplayManager and manipulate the corresponding model element. Should two different rules manipulate the same attribute, the rule defined last will overwrite changes done by the first one.

5.2 Example Configurations

This section exhibits two implementation examples, based on the model syntaxes presented in Section 2.2.

5.2.1 Petri Nets

As a first example, a Petri nets configuration was implemented. This configuration follows the model defined in Section 2.2.1, while the complete implementation is shown in Appendix C.1. The complete configuration script, composed of `RenderMM`, `AbstractMM`, and `ConcreteMM`, is 106 lines long[8]. The Kotlin code generated by this configuration encompasses 708 lines.

Shapes configured in the `RenderMM` (Listing C.1) are `circle`, `rectangle`, `line`, and `label`. With the future development of a more sophisticated RenderMetamodel, the circle shape could be defined separately, once as a small black circle and once as a bigger circle without fill color.

The `AbstractMM` (Listing C.2) definition has two class definitions. The `Place` class has one attribute name of type `string`, and another attribute tokens with type `integer` which has its bound fixed to "1" and a default value "0". Furthermore, the `Place` class has two references, incoming and outgoing, with the type `Transition` and an unlimited upper bound, indicated by "-1". The Transition class is defined in the same fashion.

The `ConcreteMM` (Listing C.3) definition has the ConnectionPlugin, and LabelPlugin, mentioned in Section 4.6 enabled. The ConnectionPlugin, in its current iteration, does not take

[8]These metrics were counted with the cloc tool: `https://github.com/AlDanial/cloc`

a configuration. The LabelPlugin, on the other hand, is configured to detect labels that are close to a graphic object with the pattern `Place` or `Transition`. If this results in more than one possible LabelFor relation, the result is discarded as noted by "`ifAmbiguous nothing`". The `patterns` scope has three patterns defined, `Place`, `Token`, and `Transition`. `Place` and `Transition` constraints are defined similarly to the examples given in Section 4.4.1. However, the `Transition` pattern contains an additional constraint that checks if a rectangle has one side at least double the other side's length. The `Token` pattern is used to detect elements that represent a token to ease the process of syncing the token count. The `DisplayManagers` scope has two definitions, one `PlaceDM`, and one `TransitionDM`. The `PlaceDM` connects the `Place` pattern to the `Place` model element. It has four rules, one for each of its attributes. Similarly, the `TransitionDM` connects the `Transition` pattern to the `Transition` model element and has three rule definitions.

The Petri nets configuration provides an example of how to use the developed prototype to define CouchEdit configurations.

5.2.2 Statecharts

As a second example, the Statecharts syntax defined in Section 2.2.2 was realized. However, the configuration does leave out some of the described functionality. This was done to align this implementation closer with the example presented in [1]. Therefore this configuration can then later serve as a point of evaluation. Notably, the implementation of pseudo states was left open. The complete source code of this implementation is listed in Appendix C.2. While the Petri nets example was explained in detail to demonstrate the usage of the developed DSL, this section only describes the implementation's noteworthy aspects.

Similarly to the Petri nets implementation presented in the prior section, this configuration has the LabelPlugin and ConnectionPlugin enabled. Furthermore, the *CompartmentDetectorPlugin* is enabled, which enables compartmentalization, as described in Section 2.3.1.4.

Detecting regions poses some special challenges. It is not only possible to split a state into two regions. It is also possible to further split up one region into two. This means that each CompHSD itself can have two more CompHSDs. Therefore the deepest CompHSDs have to be found as only they represent a region. To solve this problem, the implementation makes use of two different patterns, `DashedRegion` and `Region`. The `DashedRegion` pattern is responsible for detecting if a CompHSD is created through a *dashed* line and is a compartment of a State object. The `Region` pattern is then responsible for checking if a CompHSD that satisfies the `DashedRegion` pattern has no further sub-compartments, making it a region of the parent State.

In the initial design, the LabelPlugin was configured to prioritize states over transitions. This was deliberately designed to solve problems, such as shown in Figure 5.1. While a human can easily interpret this concrete representation, it would become problematic for the LabelPlugin to assign the label without a precedence rule correctly. However, this precedence rule turns out to be problematic when handling sub-state machines. With this rule, it becomes

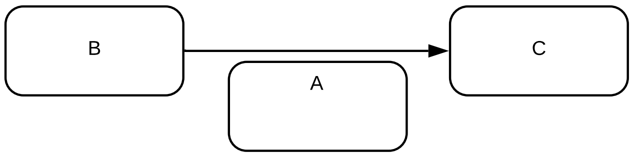


Figure 5.1: Example of problematic labeling without precedence rules.

impossible to label transitions that are part of a sub-state machine. The State precedence rule will block all transition labeling for sub-states. This makes the rule not applicable to solve the given problem and thus mostly invalidates the usefulness of the LabelPlugin for the given problem. Instead, the LabelPlugin has to be configured to check state and transition labels independently from each other. This means, to solve the problem depicted in Figure 5.1, LabelFor relations have to be checked manually. To this end, the helper `hasSameParent` is defined. This function checks if two elements are contained by the same parent or no parent at all. If this is true, both elements are part of the same hierarchy. This function is used to check if label and transition are part of the same state machine and could be associated with each other. This provides a functional workaround to solve the given problem. However, the drawback of this is that most of the functionality the LabelPlugin intends to provide cannot be used. Instead, when finding a corresponding label, all LabelFor relations have to be scanned and filtered in the relevant rule definitions, which increases the implementation's complexity.

6 Evaluation

One of the essential parts of a design science research process is the evaluation. The developed artifact has to be analyzed, and it has to be established how well the artifact realizes the defined goals.

The main criteria that has to be evaluated is the applicability of the developed artifact. Meaning, it has to be analyzed how well the developed artifact satisfies the defined goal. To this end, it seems worthwhile to evaluate the prototype's performance as well as its developer usability.

6.1 Performance

To provide performance optimization possibilities, CouchEdit was developed with the diff system in mind. Diffs provide the possibility to calculate changes without the need for reevaluating the complete hypergraph. On the flip side, this means that all possible states of the graph have to be minded. Using this diff based approach was evaluated as laborious and error-prone but proved invaluable to improve the performance of specific processing tasks [1]. Nachreiner's test results showed that especially language-specific processing tasks only took up a small amount of total processing time. Based on this result, it was decided that the developed artifact, primarily concerned with language processing, can produce components that reevaluate the complete graph on every change. Plugin processors, such as the LabelProcessor that is expected to cause high load, are still implemented on the application level and can use the performance benefits granted by the diff system.

While this research does not aim to produce detailed performance results, it was nonetheless attempted to provide a simple indication of the artifact's impact on performance. To this end, the `StateGridTest` defined in [1] (Appendix D) was adapted to work with the code generated by the Statecharts implementation (Section 5.2.2). The StateGridTest was then run 50 times for both the reference implementation of [1] and the implementation of Section 5.2.2. The tests were carried out on a modern system containing an Intel Xeon E2-1231 v3 CPU with 3.40GHz clock and 16GB DDR3 RAM with 1600MHz clock. The system was running Windows 10 version 2004. The results of this test suit are depicted in Figure 6.1. It becomes immediately apparent that the generated code performs worse than the reference implementation. Especially with higher counts of GraphicObjects, the generated implementation needs exponentially more time than the reference values. During test runs, it became clear that the generated implementation starts to max out CPU capacity very early on. Thus the rapid growth in processing time can be attributed to hardware bottlenecks.

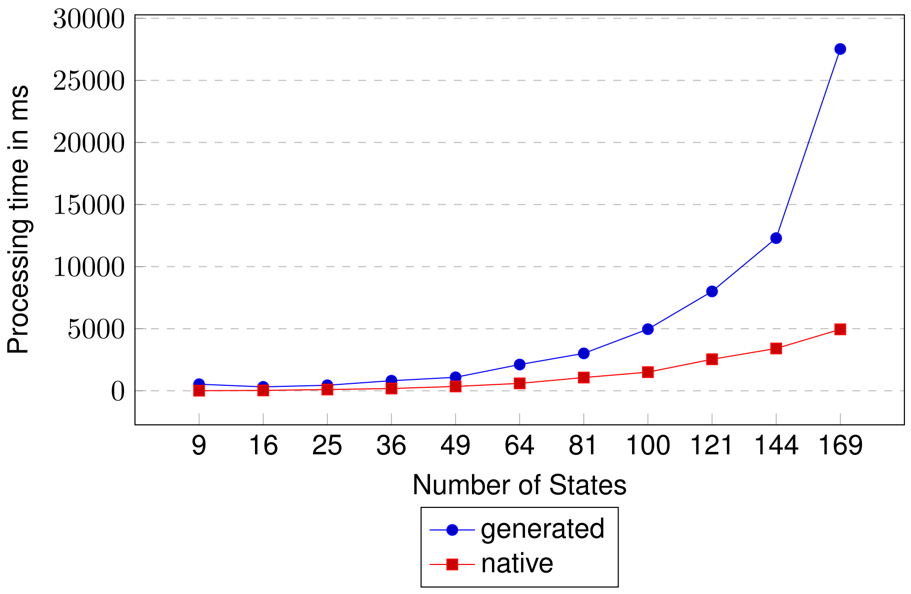


Figure 6.1: Performance comparison between reference implementation and generated implementation in the StateGridTest.

While it is not clear which part of the implementation is responsible for causing this much CPU load, multiple points could contribute. First and foremost, reevaluating the complete hypergraph in the current implementation is probably accountable for a significant increase in processing time. In the undirected architecture of CouchEdit, publishing a single change to the hypergraph means that every processor interested in this change will be triggered. The current implementation of the code generator cannot prune the elements a processor is interested in. This would require an analysis of constraints and rules, which requires a complete metamodel definition of this part. Therefore every change published triggers all processors no matter if a processor is interested in the type of change. This, combined with the fact that each generated processor reevaluates the complete state, causes much overhead. Future implementations could amend this problem by introducing the constraint and rule analysis mentioned above and using it to prune each processor's hypergraph. Furthermore, it could be possible to reintroduce the diff based architecture into generated processors. This requires that it is possible to determine which parts of the hypergraph are actually affected by a given diff. Lastly, the developed metamodel introduces a more structured ordering of processors. This could be used to split up processors into subgroups. This way, processors in the same group will calculate changes until no processor can produce new changes. All changes calculated are then bundled and passed to the next group. This could reduce the number of times a generated processor is triggered and has to reevaluate the state.

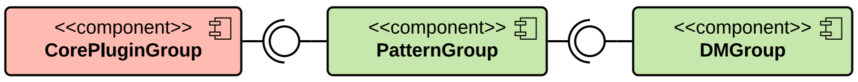


Figure 6.2: Processors split into subgroups, where one group is only activated if the previous group has finished processing.

6.2 Usability

This research's main focus was to develop an artifact that simplifies the process of implementing new modeling syntax configurations for CouchEdit and therefore improves the usability and accessibility of CouchEdit for developers. To this end, it has to be evaluated how well the developed artifact achieves this goal. A user study would be required to find a comprehensive answer to this question. Given the state of the developed artifact and the time required to conduct a survey, this goes far beyond this project's scope. Therefore, alternative metrics have to be dissected in order to determine an inclination regarding this question.

One metric to highlight is conciseness. As an abstraction of the CouchEdit architecture, the developed artifact should allow implementing similar concepts in fewer lines of code. The example implementations reflect this. The Petri nets implementation is 106 lines long and produces 708 lines of code. Analogously, the Statecharts implementation consists of 169 and generates 1154 lines. In both examples, on average, over six lines of source code are generated per line of the configuration. Of course, the code generator most likely generates a more verbose code than an equivalent written by a developer. Therefore, this metric is flawed and only serves as a suggestion towards the artifact's usability.

To further reinforce conciseness claims, the Statecharts example (Section 5.2.2) was modeled as closely as possible after the Statecharts configuration implemented in [1]. An exact replica of the modeling syntax implemented by Nachreiner is not possible because the implementation uses concepts that are not expressible using Ecore's functionality. The abstract syntax metamodel, defined by Nachreiner utilizes relations (Section 2.3.1.2), which the abstract metamodel defined in Section 4.2 does not support. Furthermore, the concrete metamodel abstracts away from implementation details and can naturally not provide the same flexibility as an application-level implementation can. Nonetheless, the application-level implementation is 1900 lines long [1] and was realized here in 169 lines. This can primarily be contributed to the fact that implementations using the metamodel do not have to mind every possible state of the graph, as well as a reduction in boilerplate code.

While the artifact abstracts away from a processors' implementation details, the developer still requires knowledge about the framework. Most prominently, a developer has to understand the hypergraph. Constraints and rules usually require querying of the hypergraph. Therefore it is inevitable to know the existing element types as well as possible relations and their meaning. Furthermore, it has to be understood which changes different plugins apply to the graph, as well as what functions are available. Nonetheless, the amount of knowledge required is still reduced.

6.2.1 Plugins

The plugin system proved as a worthwhile addition to the architecture. The example language configurations described in Section 5.2 showed how complex tasks could be trivialized if the correct plugin is provided. With the plugin system, new processors can be added to the library of existing processors should problems arise that are difficult to solve within the generalized metamodel architecture. Plugins are developed on the application level and thus have access to COUCHEDIT's complete feature set. However, this means, developing plugins requires knowledge of the complete COUCHEDIT framework, which the developed artifact is trying to abstract away from. Therefore plugins should be implemented as general as possible to be reusable as much as possible. Therein lies the challenge of the plugin system. The usefulness of a plugin depends on how well it can be adapted to different modeling syntaxes. This means that plugins have to be developed with great care. In the example Statecharts implementation (Section 5.2.2), the LabelProcessor prototype failed to solve all labeling requirements imposed by the syntax. This demonstrates how bad design can significantly reduce a plugin's usefulness. On the other hand, the ConnectionPlugin, thanks to its simplicity, provided a straightforward solution for its given domain.

6.3 Discussion of DisplayClasses

During the initial evaluation of the architecture proposed by Fondement and Baar [11], it was decided not to adopt the concept of DisplayClasses, proposed by the authors. This decision was made because their implementation details were not sure. It was estimated that DisplayClasses would introduce further complexity without providing any significant advantages. Concerning the developed artifact, this turned out as mostly true. Nonetheless, there are scenarios and considerations towards future work that could make an implementation utilizing DisplayClasses relevant.

When tightening requirements towards the concrete syntax, DisplayClasses could prove useful. For example, if it is required that a Place in the Petri nets syntax always has a name, the same part of the graph has to be queried two times. First, the PlacePatternProcessor would have to check if a label is assigned to a place before adding the pattern element of type Place. Then the PlaceRecognitionProcessor can add the PlaceDM. Afterwards, the PlaceSyncProcessor has to query the same part of the graph that the PlacePatternProcessor already queried to find the corresponding label. This double querying could be prevented, using DisplayClasses, as the PlaceSyncProcessor could get the Label connected to the DisplayClass already, similar to the invariant shown in Listing 3.1.

Another scenario that could make DisplayClasses relevant is concerned with syntax feedback in the frontend. DIAGEN marks graphical components that are part of a syntactic correct concrete representation [14]. Such a feature could be conceivable in future iterations of frontends for COUCHEDIT. With the pattern system developed in this thesis, there is no direct connection between all GOs and the abstract representation they are part of. Using DisplayClasses, every GO has a direct connection to the abstract representation they are mapped to. Thus, DisplayClasses would trivialize the implementation of a feature similar to DIAGEN's.

7 Conclusion

This work presented an experimental architecture that can be used to define modeling syntax configurations for COUCHEDIT. To this end, the approach of Fondement and Baar [11] was adapted and extended to fit the needs of the COUCHEDIT architecture. Their approach is composed of two parts, recognition and synchronization. These two mechanisms were integrated into the developed artifact as the core concept. This concept was then extended by a plugin system, which allows for modular components that can be configured to satisfy different modeling syntaxes' requirements.

It was shown that the developed artifact seems applicable for the task of specifying modeling syntaxes for COUCHEDIT. However, while it is possible to define modeling syntaxes, especially the definition of well designed plugins is key to general applicability and developer experience. Furthermore, performance test results indicate that the developed artifact in its current iteration produces configurations that increase processing overhead tenfold compared to a diff-based implementation on the application-level.

7.1 Future Work

This research poses a set of new questions that have to be answered. First and foremost, it has to be evaluated if the system's performance can be improved to reach acceptable levels. Towards this goal, multiple mechanisms could be researched.

One of the next major iterations of this artifact would be to develop a complete DSL implementation. This would allow for multiple optimizations. First, the hypergraph each processor requires could be pruned to contain only the elements relevant to a given processor. In its current iteration, the code generator has no information about the defined constraints and rules. This means that each processor has to be configured to potentially use all possible elements of the hypergraph. Therefore processors are triggered even when changes are published that have no importance for them. Furthermore, it could be researched if the extra information given by a complete metamodel definition can be utilized to generate processors using ModelDiffs. The results provided in Section 6.1 indicate that evaluation of the complete hypergraph does not scale well. If it is possible to reevaluate only parts of the hypergraph that are actually affected by incoming changes, this could improve performance considerably.

The plugin system proved to be a critical mechanism for the applicability of the developed metamodel. Plugins can take up many tasks from the user and bring further performance improvements as they are implemented on the application-level. Therefore, they can make use of the available optimization schemes. Nevertheless, Section 5.2.2 showed that a plugin's design is crucial to ensure the applicability for a wide range of modeling syntaxes. Thus it has to be evaluated which plugins are needed and how they could be implemented. The LabelPlugin was shown off as a plugin that is integral to many modeling languages and thus has to receive further attention to reach its full potential. Moreover, the CompartmentPlugin, as well as the ConnectionPlugin, could receive further improvements by introducing configuration possibilities. Besides the existing plugins, it also seems worthwhile to further investigate mechanisms that could be implemented as plugins. To this end, the works of [2] and [6] could provide a basis to build upon.

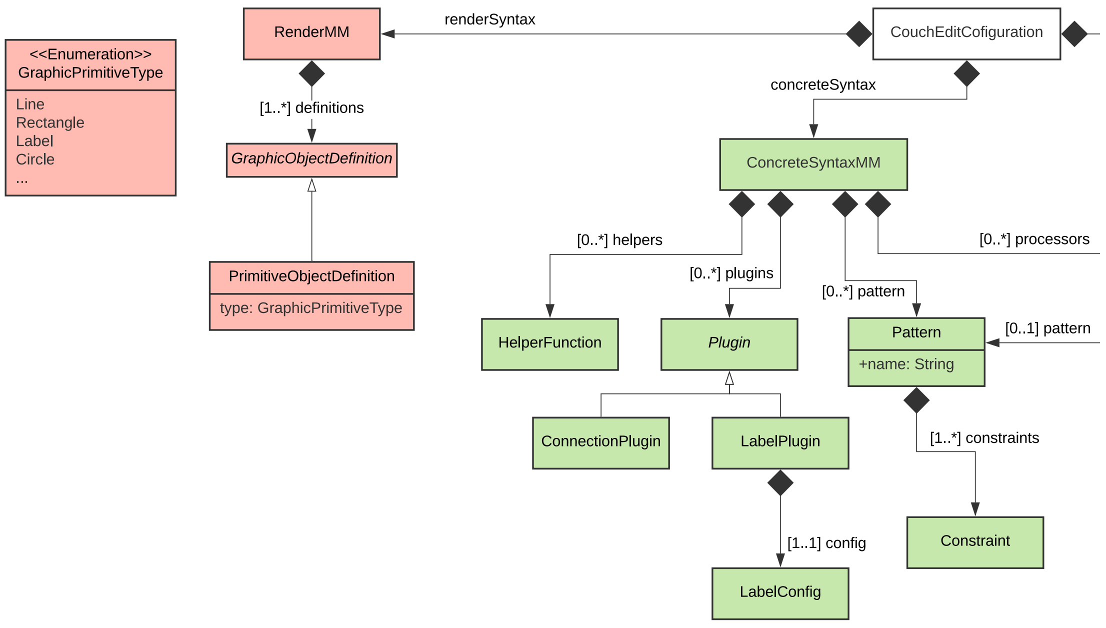
It is also essential to investigate the artifact's applicability towards correctness checking. It is critical for editor usability to convey to the user if the designed concrete model can be mapped to a correct abstract representation. To this end, it was evaluated that the introduction of DisplayClasses could result in a more explicit connection of abstract and concrete syntax and thus make it easy to decide which graphic objects art part of an abstract representation. Additionally, the work of Baar [12] could be used as a foundation to introduce correctness checking into the framework.

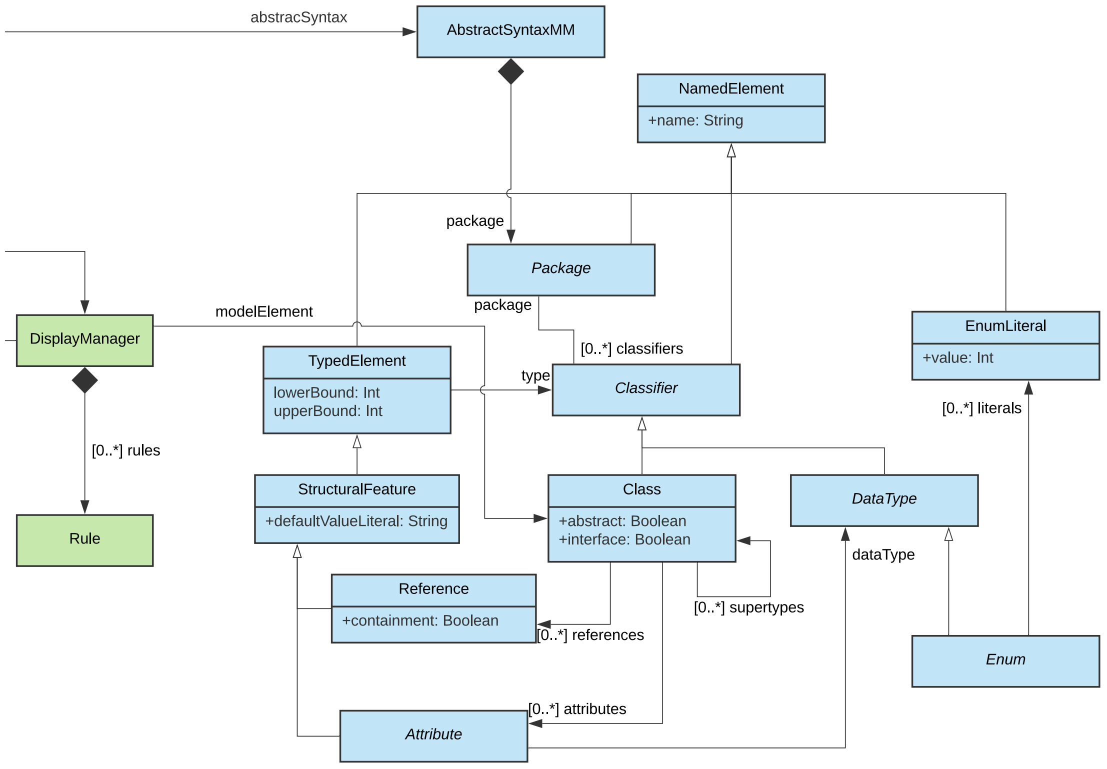
Distributed modeling is one of the potential applications of COUCHEDIT, which means development using multiple frontends that only share an abstract representation. For this to be possible, the architecture has to be able to translate from abstract to concrete syntax. In its current Iteration, the architecture is not able to do this. The artifact generates processors that strictly translate from concrete to abstract syntax. Therefore it has to be evaluated if it is possible to introduce bidirectionality into the prototype.

A critical concept theorized for the COUCHEDIT framework is the general model action mechanism [1]. In the form of *Suggestions*, they can provide the user with options that can be used to disambiguate the hypergraph or provide quick fixes that give the user fast actions to transform a malformed graph into something correct. The developed artifact currently does not support any functionality for model actions. To this end, further research is needed to find out how this functionality could integrate with the system.

In the early stages of this research, it was decided to use the approach proposed by Fondement and Baar [11]. Developing an alternative architecture that builds upon TGGs could bring multiple benefits. Especially the bidirectionality TGGs introduce promises advantages in regards to the future of this project. Nonetheless, TGGs would introduce a new set of problems that would have to be solved.

A CouchEdit Configuration Metamodel




abstracSyntax
AbstractSyntaxMM
NamedElement
+name: String
package
Package
package
modelElement
DisplayManager
TypedElement
lowerBound: Int
upperBound: Int
type
[0..*] classifiers
Classifier
EnumLiteral
+value: Int
[0..*] literals
[0..*] rules
Rule
StructuralFeature
+defaultValueLiteral: String
Class
+abstract: Boolean
+interface: Boolean
DataType
dataType
Reference
+containment: Boolean
[0..*] references
[0..*] supertypes
Enum
[0..*] attributes
Attribute

B Navigation functions

To ease traversal of the hypergraph, a DSL implementation of the COUCHEDIT configuration should provide implementations

relatedTo: For the given Element B, returns an Element A that has a OneToOneRelation of the given type pointing from A to B.

Receiver:	Element B
Params:	`clazz`: Relation type that connects A and B.
	`includeSubTypes`: Whether to include relations that subtype the given type.
Returns:	Element A where A has a relation Pointing towards B or null if no such element exists.
Throws:	`IllegalStateException`: If there are more than one Possible solutions for A.
Example:	`B.relatedTo(Contains)`

relatedFrom: For the Given Element A, returns an Element B that has a OneToOneRelation of the given type pointing from A to B.

Receiver:	Element A
Params:	`clazz`: Relation type that connects A and B.
	`includeSubTypes`: Whether to include relations that subtype the given type.
Returns:	Element B where B has a relation Pointing from A or null if no such element exists.
Throws:	`IllegalStateException`: If there are more than one Possible solutions for B.
Example:	`A.relatedFrom(Contains)`

allRelatedTo: For the given Element B, returns all Elements A that have a OneToOneRelation of the given type pointing from A to B.

Receiver: Element B

Params: `clazz`: Relation type that connects A and B.

`includeSubTypes`: Whether to include relations that subtype the given type.

`predicate`: Predicate used to filter Relations

Returns: Set of Elements A where each A has a relation Pointing from A towards B.

Example: `B.allRelatedTo(ConnectionEnd, rel -> rel.isEndConnection)`

allRelatedFrom: For the given Element A, returns all Elements B that have a OneToOneRelation of the given type pointing from A to B.

Receiver: Element A

Params: `clazz`: Relation type that connects A and B.

`includeSubTypes`: Whether to include relations that subtype the given type.

`predicate`: Predicate used to filter Relations

Returns: Set of Elements B where each A has a relation Pointing from A towards B.

Example: `A.allRelatedFrom(ConnectionEnd, rel -> rel.isEndConnection)`

dm: For a given GraphicObject, returns the corresponding DisplayManager if it exists.

Receiver: GraphicObject

Returns: DisplayManager attached to the given Element or null if none exists

Example: `go.dm`

C Kotlin DSL Configurations

C.1 Petri Nets Configuration

This Section contains the complete source code of the example implementations that are represented in Section 5.2.

Render Metamodel

```
package petrinets

import de.uulm.se.couchedit.dsl.dsl.concretesyntax.renderMM

renderMM {
  circle
  rectangle
  line
  label
}
```

Listing C.1: `RenderMM` definition for Petri nets

Abstract Metamodel

```
package petrinets

import de.uulm.se.couchedit.dsl.dsl.abstractsyntax.abstractMM

abstractMM {
  Class("Place") {
    attr("name", STRING)
    attr("tokens", INT, 1, 1, 0)
    ref("incoming", "Transition", -1)
    ref("outgoing", "Transition", -1)
  }

  Class("Transition") {
    attr("name", STRING)
    ref("incoming", "Place", -1)
    ref("outgoing", "Place", -1)
  }
}
```

Listing C.2: `AbstractMM` definition for Petri nets

Concrete Metamodel

```
package petrinets

import de.uulm.se.couchedit.dsl.dsl.translation.concreteMM

concreteMM {

  plugins {
    connection
    label {
      (circleOuterLabel("Place") and
          rectangleOuterLabel("Transition")) ifAmbiguous
          nothing
    }
  }

  patterns {
    def("Place") {
      +"""go.shapeClass == Circle::class.java"""
      +"""go.allRelatedTo(Contains::class.java).isEmpty()"""
    }

    def("Token") {
      +"""go.allRelatedTo(Contains::class.java).filterPattern
         ("Place").size == 1"""
      +"""go.allRelatedFrom(Contains::class.java).isEmpty()
         """
      +"""go.shapeClass == Circle::class.java"""
    }

    def("Transition") {
      +"""go.shapeClass == Rectangle::class.java"""
      +"""(self as GraphicObject<Rectangle>).shape.run { h *
         2 < w || w * 2 < h }"""
    }
  }

  displayManagers {
    def("Place", "Place") {
      +"""
      dm.me.tokens = dm.go
          .allRelatedFrom(Contains::class.java)
          .filterPattern("Token")
          .count()
      """

      +"""
      dm.me.name = dm.go
          .allToRelations(LabelFor::class.java)
          .filter { it.isUnambiguous() }
          .takeIf { it.size == 1 }
          ?.first()
          ?.getA()
          ?.shape
          ?.text
      """

      +"""
      dm.me.incoming = dm.go
           .allRelatedTo(ConnectionTo::class.java)
           .filterPattern("Transition")
           .mapNotNull { it.dm<Transition>()?.me?.ref() }
      """

      +"""
      dm.me.outgoing = dm.go!!
          .allRelatedFrom(ConnectionTo::class.java)
          .filterPattern("Transition")
          .mapNotNull { it.dm<Transition>()?.me?.ref() }
      """
    }

    def("Transition", "Transition") {
      +"""
      dm.me.name = dm.go!!
          .allToRelations<LabelFor>()
          .filter { it.isUnambiguous() }
          .takeIf { it.size == 1 }
          ?.first()
          ?.getA()
          ?.shape
          ?.text
      """

      +"""
      dm.me.incoming = dm.go
          .allRelatedTo(ConnectionTo::class.java)
          .filterPattern("Place")
          .mapNotNull { it.dm<Place>()?.me?.ref() }
      """

      +"""
      dm.me.outgoing = dm.go
          .allRelatedFrom(ConnectionTo::class.java)
          .filterPattern("Place")
          .mapNotNull { it.dm<Place>()?.me?.ref() }
      """
    }
  }
}
```

Listing C.3: `ConcreteMM` definition for Petri nets

C.2 Statecharts Configuration

Render Metamodel

```
package statechartsnew

import de.uulm.se.couchedit.dsl.dsl.concretesyntax.renderMM

renderMM {
  line
  label
  roundedRectangle
}
```

Listing C.4: `RenderMM` definition for Statecharts

Abstract Metamodel

```
package statechartsnew

import de.uulm.se.couchedit.dsl.dsl.abstractsyntax.abstractMM

abstractMM {

  AbstractClass("ModelElement")

  AbstractClass("StateElement") {
    superTypes("ModelElement")
    ref("incoming", "Transition", -1)
    ref("outgoing", "Transition", -1)
  }

  Class("State") {
    superTypes("StateElement")
    attr("name", STRING)
    ref("regions", "Region", -1)
    ref("elements", "StateElement", -1)
  }

  Class("Region") {
    superTypes("ModelElement")
    ref("elements", "StateElement", -1)
  }

  Class("Transition") {
    superTypes("ModelElement")
    attr("event", STRING)
    ref("source", "StateElement")
    ref("target", "StateElement")
  }
}
```

Listing C.5: `AbstractMM` definition for Statecharts

Concrete Metamodel

```
package statechartsnew

import de.uulm.se.couchedit.dsl.dsl.translation.concreteMM

concreteMM {

  plugins {
    connection
    compartmentDetector
    label {
      (containedLabel("SimpleState")) and
          lineCenterLabel("Transition")
    }
  }

  helpers {
    helper("CompartmentHotSpotDefinition.isDashed", body =
       """
    getBSet()
          .first()
          ?.findAttribute(GraphicAttributeKeys.LINE_STYLE)
          ?.value
          ?.equals(LineStyle.Option.DASHED)
          ?: false
    """)

    helper("haveSameParent", "first: Element?", "second:
       Element?", body = """
      first?.relatedTo(Contains::class.java)?.id ==
      second?.relatedTo(Contains::class.java)?.id
    """)
  }

  patterns {
    def("StateElement") {
      +"""go.hasPattern("State") || go.hasPattern("
         PseudoState")"""
    }

    def("State") {
      +"""go.shapeClass == RoundedRectangle::class.java"""
    }

    def("SimpleState") {
      +"""go hasPattern "State" """
      +"""go.dm<State>()?.me?.elements?.isEmpty() ?: false"""
      +"""go.dm<State>()?.me?.regions?.isEmpty() ?: false"""
    }

    def("DashedRegion") {
      +"""go is CompartmentHotSpotDefinition"""
      +"""(go as CompartmentHotSpotDefinition).isDashed()"""
      +"""
        go as CompartmentHotSpotDefinition
        val aSet = go.getASet()
        if (aSet.size == 1) {
          aSet.first() hasPattern "State"
        } else {
          aSet.any {
              it hasPattern "DashedRegion"
            }
        }
      """
    }

    def("Region") {
      +"""go hasPattern "DashedRegion" """
  +"""go.allFromRelations(CompartmentHotSpotDefinition::class.java).isEmpty()"""
 }

 def("Transition") {
  +"""go.shapeClass == StraightSegmentLine::class.java"""
  +"""go.relatedTo(ConnectedBy::class.java) is ConnectionTo"""
 }
}

displayManagers {
 def("State", "State") {
  +"""
   val hsds = dm.go.allFromRelations(CompartmentHotSpotDefinition::class.java)
   dm.me.name = when {
     dm.go hasPattern "SimpleState" -> dm.go.relatedTo(LabelFor::class.java)
     else -> dm.go
           .allRelatedFrom(Contains::class.java)
           .filter { it.shape is Label }
           .map { it as GraphicObject<Label> }
           .firstOrNull {
              it.allFromRelations(LabelFor::class.java)
               .none { r -> r.probability!!.probability > 0.9 }
           }
   }
     ?.shape
     ?.text
  """

  +"""
   dm.me.regions = dm.go
     .allFromRelations(CompartmentHotSpotDefinition::class.java)
     .filterPattern("Region")
     .mapNotNull { it.dm<Region>()?.me?.ref() }
  """

  +"""
```

```
        dm.me.elements = dm.go
            .allRelatedFrom(Contains::class.java)
            .filterPattern("StateElement")
            .mapNotNull { it.dm<StateElement>()?.me?.ref() }
      """

      +"""
        dm.me.outgoing = dm.go.allFromRelations(ConnectionTo::class.java)
                    .mapNotNull { it.relatedFrom(ConnectedBy::class.java) }
                    .filterPattern("Transition")
                    .mapNotNull { it.dm<Transition>()?.me?.ref() }
      """

      +"""
        dm.me.incoming = dm.go.allToRelations(ConnectionTo::class.java)
                    .mapNotNull { it.relatedFrom(ConnectedBy::class.java) }
                    .filterPattern("Transition")
                    .mapNotNull { it.dm<Transition>()?.me?.ref() }
      """
    }

    def("Region", "Region") {
      +"""
        dm.me.elements = dm.go
            .allRelatedFrom(Contains::class.java)
            .filterPattern("StateElement")
            .mapNotNull { it.dm<StateElement>()?.me?.ref() }
      """
    }

    def("Transition", "Transition") {
      +"""
        dm.me.event = dm.go
            .allToRelations(LabelFor::class.java)
            .filter { it.probability!!.probability > 0.9 }
            .map { it.getA() }
            .filter { haveSameParent(it, dm.go) }
            .takeIf { it.size == 1 }
            ?.first()
            ?.shape
            ?.text
      """

      +"""
        val connection = dm.go.relatedTo(ConnectedBy::class.
           java) as? ConnectionTo
        val getSE = {go: GraphicObject<*>? ->
          go?.takeIf { it.hasPattern("StateElement") }
            ?.dm<StateElement>()?.me?.ref()
        }
        dm.me.source = getSE(connection?.getA())
        dm.me.target = getSE(connection?.getB())
      """
    }
  }
}
```

Listing C.6: `ConcreteMM` definition for Statecharts

D Test Setup

The test setup in this section is an exact replica of the StateGridTest described in [1]. This test serves as common ground to compare COUCHEDIT's reference implementation with the implementation produced in this Thesis.

D.1 StateGridTest

D.1.1 Scenario Parameters

- `gridSizeX` and `gridSizeY`: Number of rows and columns in the generated state representation grid, respectively.
- `numberOfStatesToMove`: Amount of state representations in the concrete syntax that will be selected in step 3 and moved to the right of all other state representations.
- `additionalGridSizeX and additionalGridSizeY`: Size of the grid of additional states that will be inserted in step 5.
- `toRemoveGridSizeX` and `ToRemoveGridSizeY`: Number of columns respectively rows of the grid that should be deleted in its top left corner in step 7.

D.1.2 Test Steps

1. **Process**: Generate a grid of `gridSizeX` x `gridSizeY` concrete syntax state representations (rounded rectangle + label) and insert them as one DiffCollection.
2. **Assert** that for every concrete syntax representation, a State abstract syntax Element has been created and that the State name property is equal to the concrete syntax representation's label's content text.
3. **Process**: Move a set of `numberOfStatesToMove` concrete syntax state representations out of the middle of the grid so that they now are to the right of the previously right border of the state representation grid.
4. **Assert** that all moved Elements still represent the same state, as such purely graphical changes should not change the abstract syntax representation of the statechart.

5. **Process**: Insert a new grid of `additionalGridSizeX` x `additionalGridSizeY` state representations to the bottom right of the existing grid.
6. **Assert** that for all of the new grid elements of step 5, an abstract syntax State Element has also been inserted.
7. **Process**: Remove `toRemoveGridSizeX` x `toRemoveGridSizeY` state representations from the top left of the grid.
8. **Assert** that for all of the removed state representations, the corresponding State abstract syntax Element has also been deleted.

List of Figures

List of Listings

Bibliography

[1] L. Nachreiner, “CouchEdit - a modular graphical editing architecture for flexible modeling,” Masters Thesis, University of Ulm, Ulm, Feb. 17, 2020. [Online]. Available: `http://dx.doi.org/10.18725/OPARU-25291`.

[2] Y. Van Tendeloo, S. Van Mierlo, B. Meyers, and H. Vangheluwe, “Concrete syntax: A multi-paradigm modelling approach,” in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2017*, Vancouver, BC, Canada: ACM Press, 2017, pp. 182–193, ISBN: 978-1-4503-5525-4. DOI: `10.1145/3136014.3136017`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3136014.3136017` (visited on 03/15/2020).

[3] V. Vaishnavi and W. Kuechler. (Jan. 2004). “Design research in information systems,” [Online]. Available: `http://desrist.org/design-research-in-information-systems`.

[4] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987, Publisher: Elsevier.

[5] D. Wüest, N. Seyff, and M. Glinz, “FlexiSketch team: Collaborative sketching and notation creation on the fly,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE ’15, Florence, Italy: IEEE Press, May 16, 2015, pp. 685–688. (visited on 08/26/2020).

[6] G. Costagliola, A. Delucia, S. Orefice, and G. Polese, “A classification framework to support the design of visual languages,” *Journal of Visual Languages & Computing*, vol. 13, no. 6, pp. 573–600, Dec. 2002, ISSN: 1045926X. DOI: `10.1006/jvlc.2002.0234`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S1045926X0290234X` (visited on 04/27/2020).

[7] D. Moody, “The “physics” of notations: Toward a scientific basis for constructing visual notations in software engineering,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, Nov. 2009, ISSN: 0098-5589. DOI: `10.1109/TSE.2009.67`. [Online]. Available: `http://ieeexplore.ieee.org/document/5353439/` (visited on 04/23/2020).

[8] P. Bottoni and A. Grau, “A suite of metamodels as a basis for a classification of visual languages,” in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Rome: IEEE, 2004, pp. 83–90, ISBN: 978-0-7803-8696-9. DOI: `10.1109/VLHCC.2004.5`. [Online]. Available: `http://ieeexplore.ieee.org/document/1372303/` (visited on 04/23/2020).

[9] A. Schürr, “Specification of graph translators with triple graph grammars,” in *Graph-Theoretic Concepts in Computer Science*, E. W. Mayr, G. Schmidt, and G. Tinhofer, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1994, pp. 151–163, ISBN: 978-3-540-49183-5. DOI: `10.1007/3-540-59071-4_45`.

[10] A. Schürr and F. Klar, “15 years of triple graph grammars,” in *Graph Transformations*, H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 411–425, ISBN: 978-3-540-87405-8. DOI: `10.1007/978-3-540-87405-8_28`.

[11] F. Fondement and T. Baar, “Making metamodels aware of concrete syntax,” in *Model Driven Architecture – Foundations and Applications*, vol. 3748, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 190–204, ISBN: 978-3-540-30026-7 978-3-540-32093-7. DOI: `10.1007/11581741_15`. [Online]. Available: `http://link.springer.com/10.1007/11581741_15` (visited on 04/23/2020).

[12] T. Baar, “Correctly defined concrete syntax,” *Software & Systems Modeling*, vol. 7, no. 4, pp. 383–398, Oct. 2008, ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-008-0086-z`. [Online]. Available: `http://link.springer.com/10.1007/s10270-008-0086-z` (visited on 04/27/2020).

[13] M. Minas and G. Viehstaedt, “DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams,” in *Proceedings of Symposium on Visual Languages*, ISSN: 1049-2615, Sep. 1995, pp. 203–210. DOI: `10.1109/VL.1995.520810`.

[14] M. Minas, “Concepts and realization of a diagram editor generator based on hypergraph transformation,” *Science of Computer Programming*, vol. 44, no. 2, pp. 157–180, Aug. 2002, ISSN: 01676423. DOI: `10.1016/S0167-6423(02)00037-0`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0167642302000370` (visited on 04/07/2020).

Name: Hannah Lappe   Matrikelnummer: 922114

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den . . . .

Hannah Lappe