Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Ein kollaborativer Web-Editor zum Bearbeiten und Simulieren von RENEW Referenznetzen

Laszlo Korte

MIN-Fakultät, Fachbereich Informatik
Studiengang Informatik
Laszlo.Korte@studium.uni-hamburg.de
Matr.-Nr. 6329857

Erstgutachter: Dr. Daniel Moldt
Zweitgutachter: Marcel Hansson

Abgabe: 21.08.2025

Show me your flowcharts and conceal your tables, and I shall continue to be mystified.
Show me your tables, and I won't usually need your flowcharts; they'll be obvious.
– *Fred Brooks*

Zusammenfassung

Petrinetze sind eine Gruppe von grafischen Modellierungssprachen für die Beschreibung und Verifikation von nebenläufigen Prozessen und dynamischen Systemen. Mit Hilfe von Petrinetzen lassen sich sowohl Berechnungsprozesse, Geschäftsprozesse, biologische als auch logistische Prozesse modellieren.

Renew (Reference-Net-Workshop) ist ein Werkzeug zum Erstellen, Verifizieren und Simulieren von Petrinetzen. Es wird seit 25 Jahren als Java-Projekt entwickelt und umfasst inzwischen einen großen Funktionsumfang, der sich auch mittels Plugin-System stetig erweitern lässt. Renew bietet sowohl eine grafische Benutzeroberfläche zum Zeichnen von Netzen als auch eine Simulation-Engine zur Simulation und Verifikation der gezeichneten Modelle. Inzwischen umfasst Renew neben der Verarbeitung von Petrinetzen auch das Zeichnen und Simulieren von endlichen Automaten, Agent-Interaktion-Protokollen und einigen weiteren Formalismen.

Anwender:innen sind es von modernen Grafikprogrammen gewohnt, sowohl an ihrem Desktop-Computer als auch unterwegs an ihrem Smart-Device arbeiten zu können. Außerdem gehört ein kollaborativer Arbeitsprozess heutzutage für viele Anwendungen zur erwarteten Funktionalität. LaTeX-Dokumente können in Overleaf von mehreren Anwenderinnen und Anwendern gemeinsam erstellt werden. In Figma können mehrere Designer gemeinsam an einem Grafikentwurf arbeiten.

Die grafische Benutzeroberfläche von Renew hingegen stammt im Kern aber noch aus dessen Entstehungszeit. Renew bietet keine Möglichkeit zum kollaborativen Arbeiten und lässt sich auch nicht auf mobilen Geräten verwenden. In vergangenen Projekten wurde schon auf verschiedene Weise versucht, die Benutzeroberfläche von Renew zu überarbeiten und beispielsweise das kollaborative Bearbeiten von Petrinetz-Dokumenten zu ermöglichen. Allerdings war keiner dieser Versuche erfolgreich genug, um sich in dem Funktionsumfang von Renew zu etablieren.

Eine Hürde bei der Überarbeitung der Renew Benutzeroberfläche ist es, dass diese auf Java basiert und stark mit anderen Komponenten von Renew verbunden ist.

Inhalt dieser Arbeit ist die Entwicklung eines Prototyps eines webbasierten Editors mit grafischer Benutzeroberfläche zum kollaborativen Bearbeiten und Simulieren von Renew Dokumenten. Der Editor soll sich sowohl auf Desktop- (Windows, Mac, Linux) als auch auf Mobilgeräten (Android, iOS) verwenden lassen. Der Editor wird als eigenständige Anwendung unabhängig von Renew entwickelt, um nicht von der durch Renew gegebenen Softwarearchitektur eingeschränkt zu sein. Allerdings wird der Entwicklungsprozess durch die von Renew gegebenen Rahmenbedingungen geleitet, um eine einfache Interaktion zwischen dem neuen Editor und der bewährten Renew Simulation- und Verifikation-Engine zu gewährleisten.

Ziel der Arbeit ist es, sowohl einen funktionsfähigen Editor zu entwickeln, der in der Praxis sinnvoll eingesetzt werden kann, als auch die Erkundung von Ideen zur Verbesserung der Softwarearchitektur der Renew eigenen Benutzeroberfläche.

Inhaltsverzeichnis

Inhaltsverzeichnis

Teil I.

Einleitung und Grundlagen

1. Einleitung

Der Inhalt dieser Arbeit ist die Entwicklung einer webbasierten Anwendung zum kollaborativen Bearbeiten und Simulieren von Petrinetzen. Im Folgenden wird zunächst die Problemstellung motiviert und dann genauer erläutert. Anschließend wird ein Überblick über die Struktur der Arbeit gegeben.

1.1. Motivation

Petrinetze werden sowohl in der Lehre, in der Forschung als auch in der Industrie eingesetzt, um komplexe dynamische und verteilte Systeme zu modellieren und zu verifizieren. Renew ist ein vielseitiges und mächtiges Werkzeug zur Erstellung, Simulation und Verifikation von Petrinetzen [KW99]. Es wird seit 25 Jahren weiterentwickelt und lässt sich dank seiner Plugin-Architektur stetig um neue Funktionalität erweitern [Sch03].

Die Benutzeroberfläche von Renew bringt jedoch einige Altlasten mit sich, die trotz einiger Versuche bisher nicht abgelegt werden konnten.

Als Java-Desktop-Anwendung lässt sich Renew beispielsweise nicht auf Mobilgeräten unter Android oder iOS verwenden. Besonders für Einsteiger stellt dies eine Hürde dar, da Unterrichtsmaterialien nicht ad-hoc digital bearbeitet werden können, ohne zunächst eine Installation von Renew auf einem kompatiblen Gerät durchgeführt zu haben.

In vergleichbaren Softwareanwendungen aus anderen Anwendungsbereichen gehört es zum gewohnten Funktionsumfang, dass mehrere Benutzer:innen gleichzeitig gemeinsam an einem Projekt oder einem Dokument arbeiten können. In Renew ist es bisher nicht gelungen, die Benutzeroberfläche zufriedenstellend für die Unterstützung solcher kollaborativen Arbeitsabläufe zu erweitern.

Des Weiteren ist die Benutzeroberfläche von Renew durchzogen von einigem Fehlverhalten, welches für Einsteiger:innen verwirrend ist. Aufgrund der Softwarearchitektur der Benutzeroberfläche lässt es sich nicht einfach beheben, sodass sich erfahrenere Benutzer:innen einfach daran gewöhnen. Ein Beispiel hierfür ist, dass sich auf der Zeichenfläche platzierte Elemente aus dem Sichtbereich des Benutzenden schieben lassen und dann dauerhaft unzugänglich sind.

Zusammenfassend lässt sich also sagen, dass Renew ein mächtiges Modellierungswerkzeug ist, dessen Nutzen besonders für Einsteiger, aber auch im professionellen Einsatz, durch die vorhandene Benutzeroberfläche stark geschmälert wird. Der in dieser Arbeit entwickelte kollaborative Editor soll als alternative Benutzeroberfläche sowohl Einsteigerinnen und Einsteigern die Arbeit an Renew-Dokumenten erleichtern als auch zur Inspiration für die Weiterentwicklung der Renew-eigenen Benutzeroberfläche dienen.

1.2. Problemstellung

In diesem Abschnitt wird die Problemstellung, mit der sich diese Arbeit befasst, genauer spezifiziert. Insbesondere soll zunächst die Überarbeitung der bestehenden Renew-Benutzeroberfläche gegenüber der Entwicklung einer neuen eigenständigen Anwendung abgewogen werden.

Renew als Softwareprojekt existiert bereits seit 25 Jahren. Das heißt zum einen, dass schon sehr viele Arbeitsjahre von vielen Menschen in die Entwicklung geflossen sind, um den derzeitigen Funktionsumfang zu erreichen. Zum anderen heißt es auch, dass bereits sehr viele bestehende Projekte und Dokumente in Renew erstellt wurden, und dass bereits eine Gemeinschaft aus erfahrenen Benutzer:innen für Renew existiert.

Es hätte also keinen Zweck, in dieser Arbeit einen komplett neuen Petrinetz-Editor unabhängig von Renew zu entwickeln. In dem begrenzten Zeitrahmen kann gar keine Anwendung entwickelt werden, die in ihrer Leistungsfähigkeit auch nur ansatzweise mit Renew konkurrieren könnte. Außerdem könnte auch die existierende Nutzergemeinschaft mitsamt ihrem Know-how nicht dafür gewonnen werden, an diesem neuen, alternativen Projekt zu arbeiten.

Die in Renew bestehende Benutzeroberfläche zu überarbeiten, um die in der Motivation genannten Schwachstellen und Probleme zu beheben, scheint aber ebenfalls keine umsetzbare Strategie zu sein. Die derzeitige Benutzeroberfläche von Renew ist in Java Abstract Window Toolkit (AWT) entwickelt und lässt sich somit grundsätzlich nicht auf Android- oder iOS-Geräte portieren, ohne von Grund auf neu entwickelt zu werden. Es wäre zwar denkbar, die gesamte Softwarearchitektur der Benutzeroberfläche innerhalb von Renew zu überarbeiten, um diese sauberer von anderen Komponenten zu trennen und strukturelle Fehler zu beheben. Allerdings müsste solch eine massive Änderung von allen anderen Komponenten von Renew abgestimmt werden, und es wäre nicht absehbar, ob das Vorhaben erfolgreich sein wird.

Zusammengefasst besteht die Problemstellung dieser Arbeit also darin, einen Editor mit moderner Benutzeroberfläche zum Bearbeiten und Simulieren von Petrinetzen zu erschaffen, ohne dabei den Anschluss an die bestehende Renew-Nutzergemeinschaft zu verlieren, ohne die von Renew geleistete Arbeit zu ignorieren und ohne eine massive zu koordinierende Änderung an Renew durchzuführen.

1.3. Zielsetzung und Forschungsfragen

Dieser Abschnitt konkretisiert die Zielsetzung dieser Arbeit basierend auf der zuvor erläuterten Problemstellung und Motivation.

In der Motivation wurde bereits das benutzerfreundliche Bearbeiten und Simulieren von Petrinetzen mittels Renew in den Fokus gesetzt. In der Problemstellung wurde weiter erklärt, wie weder die Neuentwicklung einer Anwendung Renew-Konkurrent, noch die

komplette Überarbeitung der Renew-Benutzeroberfläche ein im Zeitrahmen praktikabler Ansatz sind, um die in Renew aufgezeigten Probleme zu beheben.

Für diese Arbeit wird ein Kompromiss gewählt. Es soll eine eigenständige Anwendung entwickelt werden, die sich mittels gemeinsamer Datenformate sowohl von Einsteiger:innen als auch von erfahrenen Renew-Benutzer:innen ergänzend zu Renew verwenden lässt, um an bestehenden Renew-Projekten zu arbeiten.

Die Entwicklung einer eigenständigen Anwendung minimiert die nötige Koordination mit dem existierenden Renew-Softwareprojekt. Die Anbindung an das bestehende Renew-Ökosystem mittels gemeinsamer Datenformate soll mehrere Vorteile bieten. Bestehende Renew-Benutzer:innen können bestehende Renew-Dokumente im neuen Editor bearbeiten, um die Vorteile der neuen Benutzeroberfläche zu erleben. Im neuen Editor erzeugte Dokumente können im altbewährten Renew weiterverarbeitet werden, falls sich der neue Editor nicht langfristig durchsetzt. Erkenntnisse über eine Software-Architektur, die sich bei der Entwicklung des Editors in dieser Arbeit bewährt, können in die Entwicklung von Renew zurückfließen.

Zusammenfassend lässt sich die Forschungsfrage dieser Arbeit also in zwei Teilen stellen. Wie lässt sich ein kollaborativer Editor mit grafischer Benutzeroberfläche für das bestehende Renew-Softwareprojekt entwickeln? Welche Erkenntnisse aus dieser Entwicklung lassen sich für die Weiterentwicklung von Renew nutzen?

1.4. Aufbau der Arbeit

Diese Arbeit besteht aus sieben Teilen. In Teil 1 werden zunächst die Motivation, Zielsetzung und Grundlagen für die Entwicklung des Webeditors vorgestellt. In Teil 2 werden die genauen Anforderungen an den Editor erarbeitet und eine Strategie für die Umsetzung vorgestellt. Der in dieser Arbeit entwickelte Editor lässt sich als Zusammenspiel aus den drei folgenden Komponenten verstehen. Die Umsetzung der Server-Komponente wird in Teil 3 behandelt. Die Umsetzung der Client-Komponente wird in Teil 4 behandelt. Teil 5 behandelt die Umsetzung der Simulator-Komponente. In Teil 6 wird diskutiert, inwieweit die in Teil 2 aufgestellten Anforderungen mittels der gewählten Strategie erfüllt werden konnten. Außerdem wird in Teil 6 untersucht, welche Erkenntnisse für Renew im Verlauf dieser Arbeit gesammelt werden konnten. In Teil 7 wird abschließend noch einmal zusammengefasst, was mit dieser Arbeit erreicht wurde und welcher Ausblick den entwickelten Editor für die Zukunft erwartet.

2. Grundlagen

In diesem Kapitel werden die Grundlagen präsentiert, die nötig sind, um den Inhalt dieser Arbeit zu verstehen und einordnen zu können. Für die Motivation dieser Arbeit sind besonders die Kenntnisse über Petrinetze, Referenznetze und über das Renew-Modellierungswerkzeug von Bedeutung.

Für die technische Umsetzung dieser Arbeit kommen einige Programmiersprachen und Programmierschnittstellen als Werkzeuge zum Einsatz. Dazu gehören die Schnittstellen zum Programmieren von Webanwendungen, die Programmiersprachen JavaScript, Erlang und Elixir, die Konzepte der funktionalen, reaktiven und relationalen Programmierung und die Programmbibliotheken Phoenix, Svelte, SQLite und Ecto.

In Kapitel 5 „Strategie zur Umsetzung“ wird die Verwendung dieser Programmiersprachen und Schnittstellen im Rahmen dieser Arbeit motiviert.

Im Folgenden wird ein kurzer Überblick über Petrinetze, Renew und die verwendeten Programmiersprachen und Schnittstellen gegeben.

2.1. Petrinetze

Petrinetze sind eine Klasse von Graph-basierten Formalismen zur Modellierung von nebenläufigen Prozessen. Sie wurden von Carl Adam Petri in den 1960er-Jahren entwickelt [Pet77c, Pet77b, Pet77a]. Wolfgang Reisig bietet eine umfassende Übersicht über Petrinetze [RR19].

Petrinetze bestehen aus einem gerichteten und gewichteten bipartiten Graphen als Netzstruktur und einer zugehörigen Semantik. Die Netzstruktur besteht aus zwei Arten von Knoten, den so genannten Plätzen und Transitionen. Die Plätze, Transitionen und Kanten können mit Anschriften versehen sein, die durch eine zugrundeliegende Semantik interpretiert werden können.

Im Allgemeinen repräsentieren die Platz-Knoten passive Elemente, wie Zustände oder Ressourcen. Die Transitions-Knoten hingegen stehen für aktive Elemente wie etwa Aktio-

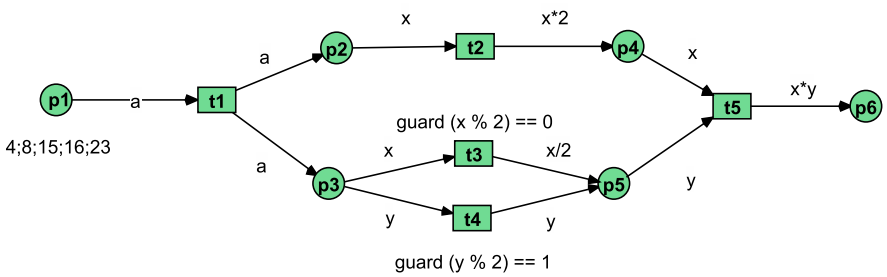


Abbildung 2.1.: Ein Petrinetz mit Ganzzahligen Marken

nen oder Ereignisse, die sich auf die Plätze auswirken. Die Kanten setzen die Plätze und Transitionen in eine Ursache-Wirkung-Beziehung beziehungsweise Vorbedingung-Nachbedingung-Beziehung zueinander. Ein Platz kann eine Menge von Marken enthalten. Eine Transition kann in einem sogenannten Schaltvorgang Marken von einem benachbarten Platz konsumieren oder neue Marken auf einem benachbarten Platz produzieren. Je nach Semantik können als Marken unterschiedliche Objekte zum Einsatz kommen.

In Abb. 2.1 ist ein einfaches Netz dargestellt. Es besteht aus sechs kreisförmigen Plätzen und fünf rechteckigen Transitionen. Auf dem Platz p_1 ganz links befinden sich als initiale Marken die fünf Ganzzahlen 4, 8, 15, 16 und 23. Die Kanten und Transitionen sind jeweils mit Textanschriften versehen. Dadurch wird ausgedrückt, welche Marken beim Schalten einer Transition von den benachbarten Plätzen konsumiert und produziert werden. Die Transition t_1 konsumiert eine beliebige Marke a von dem Platz p_1 und produziert auf den Plätzen p_2 und p_3 jeweils eine Marke mit dem selben Wert a. Die Transition t_3 konsumiert eine gerade Zahl x (`guard` $x \mod 2 \equiv 0$) von Platz p_3 und produziert auf Platz p_5 eine Marke, deren Wert sich durch $\frac{x}{2}$ ergibt.

Eine Netzstruktur zusammen mit der zugehörigen Semantik kann verwendet werden, um eine Folge von sequenziellen oder nebenläufigen Schaltvorgängen zu simulieren. Somit können Petrinetze verwendet werden, um komplexe Systeme zu modellieren und auf Korrektheitseigenschaften zu untersuchen [RR19].

2.2. Referenznetze

Olaf Kummer erweiterte die Petrinetze um eine ausdrucksstarke Semantik zu sogenannten Referenznetzen [KW99]. In Referenznetzen können dynamische Netzinstanzen aus einer statischen Netzvorlage erzeugt werden. Vergleichbar mit Objekten in der objektorientierten Programmierung, die basierend auf einer Klassendefinition instanziiert werden können. Diese Netzinstanzen können dann wiederum als Marken auf den Plätzen einer anderen Netzinstanz platziert werden. Somit gehören die Referenznetze zu der Klasse der Netze höherer Ordnung. Außerdem können auch beliebige Java-Objekte als Marken auf die Plätze einer Netzinstanz gelegt werden. Transitionen in Referenznetzen können während eines Schaltvorgangs auch beliebigen Java-Code ausführen. Verschiedene Transitionen können durch sogenannte synchrone Kanäle auch netzübergreifend miteinander gekoppelt werden. Dadurch können Transitionen dazu gebracht werden, gemeinsam schalten zu müssen.

Abb. 2.2 stellt ein Netzsystem aus drei Referenznetzen dar. Die drei Netze Renew als Beispielnetze in Renew enthalten. Das Netzsystem modelliert einen Löschvorgang, bei dem Feuerwehrleute (Fireman) sich mit wassergefüllten Eimern (Bucket) einem Feuer nähern, um dieses zu löschen. Die Referenznetzsemantik spiegelt sich darin wider, dass die Netze sich gegenseitig referenzieren und Exemplare eines Netzes jeweils als Marke auf den Plätzen eines anderen Netzes eingesetzt werden. Beispielsweise produziert

die Transition ganz oben links im Fireman-Netz (Abb. 2.2(b)) ein Exemplar des Bucket-Netzes (Abb. 2.2(c)) und legt dieses auf den angrenzenden Platz. Die Transitionen im Fireman-Netz sind über die synchronen Kanäle, die unter anderem als `fill`, `refill` und `exchange` benannt wurden, an die zugehörigen Transitionen in den anderen beiden Netzen gekoppelt.

2.3. Renew

Renew (Reference-Net-Workshop) ist eine auf Java basierende Entwicklungsumgebung zur Modellierung, Simulation und Verifikation von Petrinetzen und insbesondere der namensgebenden Referenznetze. Es wurde 1999 erstmals veröffentlicht und seither stetig weiterentwickelt [KW99]. Die zum Zeitpunkt dieser Arbeit aktuellste Version 4.1 wurde 2023 veröffentlicht [KWD+23].

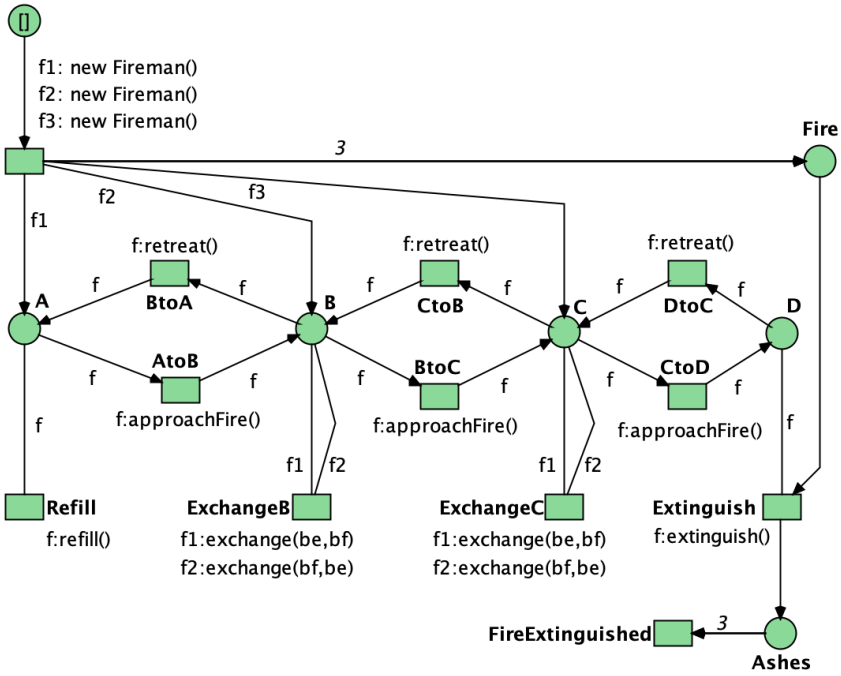
Dank der von Schumacher 2004 eingeführten Plugin-Architektur lässt sich Renew auch vielseitig erweitern [Sch03]. Es unterstützt inzwischen neben der Modellierung von Referenznetzen auch Formalismen zur Modellierung von endlichen Automaten, Agent-Interaction-Protocol (AIP)-Diagrammen und Multi-Agenten-Systemen [Möl16, MHMS17, Möl18, Cab10].

Zusammengefasst besteht Renew aus einer grafischen Benutzeroberfläche, mittels derer auf einer Zeichenfläche grafische Modelle gezeichnet werden können und aus einer Simulator-Engine, die ein gezeichnetes Modell mit einer gewählten Semantik interpretieren und simulieren kann.

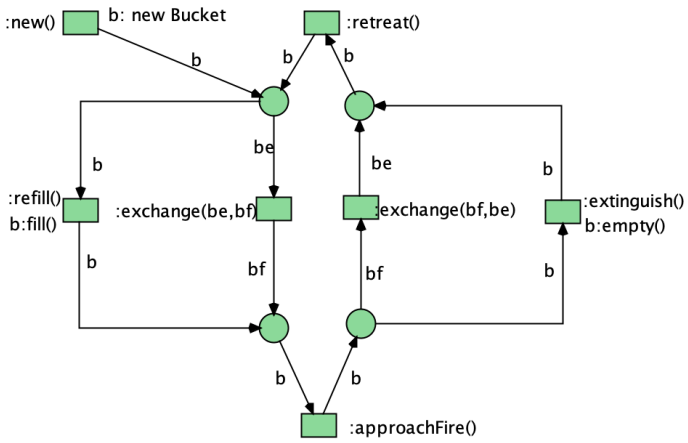
In Abb. 2.3 ist die Benutzeroberfläche von Renew dargestellt. Auf der linken Seite sind die Schaltflächen der Werkzeuge zu sehen, mittels derer sich grafische Elemente auf der Zeichenfläche erstellen lassen, um Netzgraphen und andere Modelle zu zeichnen. Auf der rechten Seite ist eine Netzstruktur auf zwei getrennten Zeichenflächen zu sehen. Die obere Zeichenfläche mit weißem Hintergrund ermöglicht das Bearbeiten der einzelnen Elemente. Die untere, blau eingefärbte Fläche stellt die Simulation einer Instanz des Netzes dar. Auf dieser lässt sich das dynamische Verhalten des Netzes beobachten. Per Mausklick lassen sich Transitionen schalten, um die Marken auf den Plätzen durch das Netz zu bewegen. In der Simulationsansicht lässt sich die Struktur des Netzes aber nicht mehr bearbeiten.

Darüber hinaus bietet Renew weitere Werkzeuge, um mit den gezeichneten Modellen zu arbeiten. Viele der Funktionen lassen sich auch ohne grafische Benutzeroberfläche per Befehlszeile ansteuern.

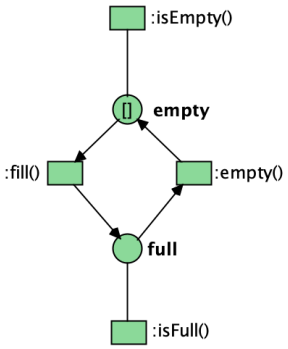
Seit 2019 wird in der Entwicklung von Renew auf das Java Platform Module System (JPMS) gesetzt, um das Gesamtprojekt in kleinere, klar getrennte Komponenten zu zerlegen und diese entsprechend ihrer Zuständigkeit besser zu strukturieren [CMH+22, Ehl23]. Die grafische Benutzeroberfläche stellt in diesem Prozess der Umstrukturierung aber eine besondere Herausforderung dar, weil die Benutzeroberfläche stark mit allen anderen Funktionen von Renew verwoben ist [WMH18].



(a) FireExtinction



(b) Fireman



(c) Bucket

Abbildung 2.2.: Drei Referenznetze, die zusammen als Netzsystem einen Löschvorgang modellieren

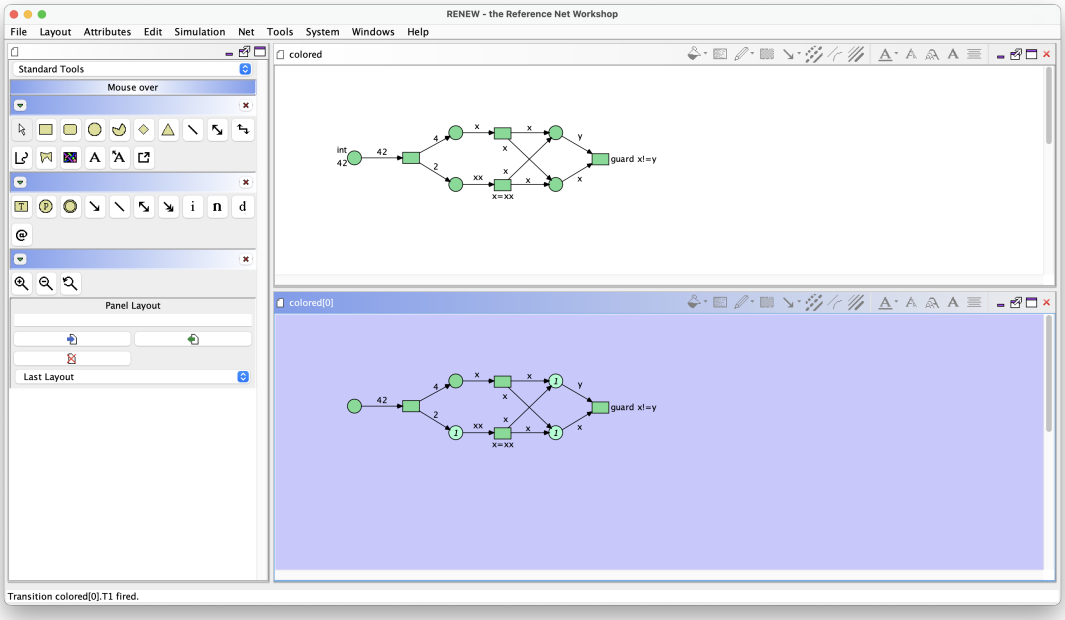


Abbildung 2.3.: Screenshot der RENEW Benutzeroberfläche

2.4. Webanwendungen

Ursprünglich wurde das World Wide Web (WWW) entwickelt, um sich gegenseitig referenzierende Textdokumente, wie etwa wissenschaftliche Papiere, im Internet bereitstellen und mittels einer dafür vorgesehenen Anwendung, dem Web-Browser, abrufen und anzeigen zu können [BL89].

Mit der Zeit wurde der Funktionsumfang von Web-Browsern aber so stark erweitert, dass sich heutzutage komplexe Softwareanwendungen, sogenannte Webanwendungen, im Webbrowser aufrufen und ausführen lassen.

Im Folgenden wird ein Überblick über die Vorteile von Webanwendungen und über die wichtigsten zugehörigen Programmierschnittstellen gegeben.

2.4.1. Vorteil gegenüber nativen Anwendungen

Für Softwareentwickler:innen bieten Webanwendungen den Vorteil gegenüber nativen Anwendungen, dass nicht speziell für ein jeweiliges Betriebssystem programmiert werden muss, sondern ein Web-Browser als Zielplattform auf einer Vielzahl von Geräten zur Verfügung steht.

Für eine:n Benutzer:in bieten Webanwendungen den Vorteil gegenüber nativen Anwendungen, dass eine zu verwendende Anwendung nicht auf dem eigenen Gerät installiert und mittels Softwareupdates gewartet werden muss, sondern die Anwendung direkt im Web-Browser aufgerufen und ausgeführt werden kann.

2.4.2. HTML5

Ähnliche Vorteile haben auch Java Applets, Adobe Flash und Microsoft Silverlight versprochen, konnten sich aus verschiedenen Gründen allerdings nicht durchsetzen [Ope16, Moz17, Mic21]. In diesem Kontext wird daher unter einer Webanwendung eine Softwareanwendung verstanden, die auf den standardisierten Programmierschnittstellen eines Web-Browsers aufgebaut ist. Der Überbegriff dieser Schnittstellen lautet HTML5 [WHA25].

Im folgenden werden einige dieser Programmierschnittstellen kurz vorgestellt. Webanwendungen sind typischerweise Client-Server-Anwendungen in denen ein Client (der Web-Browser) über eine Netzwerkverbindung mit einem Server kommuniziert [TS95]. Wie diese Kommunikation im Speziellen aussieht, hängt von der Anwendung ab.

Die hier vorgestellten Schnittstellen beziehen sich auf die vom Client (Web-Browser) bereitgestellte Funktionalität. Für die Programmierung der Server-Komponente einer Webanwendung gibt es im Allgemeinen keine Einschränkungen, auf die geachtet werden muss, da der Server von den Entwicklern und Entwicklerinnen der Anwendung bereitgestellt wird. Die Programmierung des Clients hingegen muss kompatibel mit dem vom Nutzer verwendeten Web-Browser sein.

2.4.3. Hypertext Transfer Protocol (HTTP)

Das Hypertext Transfer Protocol (HTTP) ist ein Transmission Control Protocol (TCP) basiertes Netzwerkprotokoll, welches ursprünglich dafür entwickelt wurde, um Textdokumente von einem entfernten Rechner anfordern und herunterladen zu können [Fie99, Pos81]. Im Kontext von Webanwendungen kann es weiterhin dafür benutzt werden, um beliebige Ressourcen bereitzustellen. Per HTTP ist keine bidirektionale Kommunikation möglich [Fie99]. Ein Server kann also Ressourcen für einen Client bereitstellen, allerdings kann sein Server einen Client nicht proaktiv über Änderungen informieren.

Representational State Transfer (REST) ist ein Paradigma, nach dem Programmierschnittstellen einer Webanwendung auf Basis von HTTP entwickelt werden [Fie00].

2.4.4. HTML und CSS

Hyptertext Markup Language (HTML) ist eine Extensible Markup Language (XML)-ähnliche Auszeichnungssprache, die ursprünglich dafür entwickelt wurde, um formatierte Textdokumente zu erstellen [WHA25, BPSMM98]. In Kombination mit Cascading Style Sheets (CSS) können vielseitige und komplexe Dokumente gestaltet werden [Wor96].

Im Kontext von Webanwendungen können anstelle von Textdokumenten auch grafische Benutzeroberflächen für interaktive Softwareanwendungen gestaltet werden, denn HTML bringt neben Text- und Tabellenformatierung auch Bausteine für interaktive Elemente wie Texteingabefelder, Auswahlboxen und Schaltflächen mit sich [WHA25].

2.4.5. JavaScript

JavaScript (JS) bzw. JavaScript (ECMAScript) ist eine objektorientierte Programmiersprache, die in HTML-Dokumente eingebettet werden kann und von Webbrowsern zur Laufzeit interpretiert wird [ECM15]. Ursprünglich war JavaScript dafür konzipiert, statische HTML-Dokumente um dynamische Elemente wie Animationen zu erweitern oder auf Benutzerinteraktionen reagieren zu lassen [WG99]. Im Kontext von Webanwendungen stellt der JavaScript-Code aber das Herzstück einer Anwendung dar, an dem die gesamte Ein- und Ausgabe koordiniert wird. Mittels der Document Object Model (DOM) Schnittstelle kann ein JS Prozess auf den Inhalt eines HTML Dokuments zugreifen und dieses in Reaktion auf Benutzereingaben manipulieren [con25b].

2.4.6. JavaScript Objekt Notation (JSON)

JavaScript Objekt Notation (JSON) ist eine Teilmenge der Syntax von JavaScript [Bra17]. Sie ist darauf beschränkt, einfache Datenstrukturen, die in JavaScript als Literale angegeben werden können, in Form der Literalsyntax zu serialisieren. Mittels JSON können Zahlen, Strings, Wahrheitswerte und geschachtelte Listen und endliche Abbildungen serialisiert werden. JSON kann als Datenformat zur Übertragung von Daten per HTTP verwendet werden.

2.4.7. Canvas

Die Canvas-Schnittstelle stellt eine Rastergrafik-Zeichenfläche bereit, die in ein HTML-Dokument eingebunden werden kann [Moz25]. Mittels JS oder WebGL-Shader können prozedural zur Laufzeit beliebige Grafiken generiert werden, die sich mittels HTML und CSS Schnittstelle nicht erzeugen ließen [Khr18].

2.4.8. Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) ist ein XML-basiertes Datenformat zur Beschreibung von Vektor-Grafiken [Wor11b]. Es lässt sich direkt in HTML-Dokumente einbetten und mittels JavaScript manipulieren. Es kann im Kontext von Webanwendungen verwendet werden, um Vektorgrafiken zur Laufzeit zu erzeugen und anzuzeigen. Mittels CSS können Darstellungseigenschaften von Grafikelementen interaktiv modifiziert werden [Wor11a].

2.4.9. WebSocket

WebSocket ist ein Netzwerkprotokoll, welches die bidirektionale Kommunikation zwischen einem Server und einem Client ermöglicht [Int11]. Im Gegensatz zu HTTP ist es in einer WebSocket-Verbindung für den Server möglich, eigenständig Nachrichten an einen Client zu senden [Doc25]. Im Rahmen von Webanwendungen kann das WebSocket-Protokoll verwendet werden, um eine bidirektionale Kommunikation zwischen einem zentralen Server und einem oder mehreren Clients zu implementieren.

2.4.10. Server Sent Events (SSE)

SSE sind eine Erweiterung des HTTP-Protokolls, damit ein Client mehrere vom Server gesendete Nachrichten empfangen kann [Wor12]. Es kann in Kombination mit dem original HTTP-Protokoll ähnlich wie WebSocket verwendet werden, um eine bidirektionale Kommunikation zwischen Client und Server aufzubauen.

2.4.11. Web Real Time Connection (WebRTC)

WebRTC ist ein weiteres Netzwerkprotokoll, welches es mehreren Clients (Webbrowsern) ermöglicht, eine peer-to-peer (P2P) Verbindung miteinander aufzubauen, ohne mit einem zentralen Server zu kommunizieren [Wor18].

Zusammenfassend lässt sich feststellen, dass für die Programmierung einer Webanwendung eine Vielzahl an Schnittstellen zur Netzwerkkommunikation und Darstellung grafischer Elemente bereitsteht.

2.5. Funktionale Programmierung

Funktionale Programmierung ist eine bestimmte Weise, den Kontrollfluss und Datenfluss in der Programmierung eines Systems zu strukturieren [Tho10]. Gegenüber der in Java und Renew verwendeten objektorientierten Programmierung zeichnet sich die funktionale Programmierung dadurch aus, dass anstelle von Klassen, Objekten und Methoden nur Funktionen zur Programmierung des Kontrollflusses einer Anwendung verwendet werden [Pau10].

Im Folgenden wird anhand eines kurzen Beispiels die funktionale Programmierung der objektorientierten Programmierung gegenübergestellt. Danach werden die reaktive Programmierung und das Optik-Entwurfsmuster als zwei Techniken aus der funktionalen Programmierung vorgestellt.

2.5.1. Vergleich zu objektorientierter Programmierung

Eine Funktion ist eine eindeutige Abbildung von Eingabe- auf Ausgabewerte [Pau10]. Eine Funktion kann, anders als eine objektorientierte Methode, keine Änderungen an einem globalen Zustand oder an einer existierenden Datenstruktur vornehmen [Tho10]. Änderungen von Zuständen können ausschließlich durch die Erzeugung eines neuen Zustands als expliziten Rückgabewert einer Funktion erreicht werden.

Diese starke Einschränkung erlaubt das logische Schließen über Veränderungen von Datenstrukturen anhand des lokalen Kontrollflusses, der sich aus dem programmierten Quelltext ergibt [Pau10].

In Listing 2.1 ist ein Beispiel gegeben. Im objektorientierten Paradigma können wir, basierend auf dem gezeigten Ausschnitt, nichts darüber wissen, in welchem Zustand sich das in Zeile 1 erzeugte Objekt befindet. Sowohl die Operation `setupOperation` in

Zeile 2, als auch die Operation `otherOperation` in Zeile 3 haben potenziell eine Änderung an dem in `someInitial` enthaltenen Zustand vornehmen können. Basierend auf einer funktionalen Semantik können wir aber garantieren, dass im Laufe der Operationen der Zustand `someInitial` nicht verändert worden sein kann. Die einzige Auswirkung von `setupOperation` kann die Berechnung des sich nun in `someState` befindlichen Wertes gewesen sein. Genau so kann nur `otherOperation` weder am Zustand von `initialState` noch an `someState` etwas verändert haben.

In der objektorientierten und prozeduralen Programmierung können Änderungen an einer Datenstruktur auftreten, die im lokalen Kontext einer Methode oder einer Prozedur nicht absehbar sind. Dadurch ist es sehr schwierig, potenzielle Fehler im Programmablauf zu finden und nachzuvollziehen.

```
1 const someInitial = new Object()
2 const someState = setupOperation(someInitial)
3 const result = otherOperation(someState)
```

Quelltext 2.1: Beispiel für Kontrollfluss in funktionaler Programmierung

2.5.2. Reaktive Programmierung

Die reaktive Programmierung ist die Übersetzung des Beobachtermusters aus der objektorientierten Programmierung in das Paradigma der deklarativen funktionalen Programmierung [Lam86, HK09, WH00].

In der objektorientierten Programmierung ist das Beobachtermuster ein verbreitetes Entwurfsmuster, welches dazu dient, Zustandsänderungen zwischen verschiedenen Komponenten eines Systems zu kommunizieren [GHJV94].

Eine Schwierigkeit beim Einsatz des Beobachtermusters in komplexen Systemen ist die Gewährleistung der Konsistenz zwischen allen Komponenten. Außerdem muss sichergestellt werden, dass in der Verkündung von Änderungen zwischen Komponenten keine endlosen Zyklen entstehen.

In der reaktiven Programmierung wird der Datenfluss zwischen Komponenten in Form von funktionalen Abhängigkeiten deklariert. Transitive Abhängigkeiten können automatisch erkannt und in topologisch korrekter Reihenfolge verarbeitet werden.

In der Entwicklung von interaktiven Benutzeroberflächen kann die reaktive Programmierung also eingesetzt werden, um einen stets konsistenten Zustand der Benutzeroberfläche zu erzielen [CC13].

Quellcode 2.2 zeigt ein einfaches Beispiel für die Anwendung des reaktiven Paradigmas in JavaScript. In Zeile 1 und 2 werden zunächst zwei reaktive Variablen initialisiert. Das Schlüsselwort `state` markiert sie als reaktiv. Sie enthalten ein Geburtsdatum und das aktuelle Jahr. In Zeile 3 wird eine Berechnung durchgeführt, um aus den beiden Variablen das aktuelle Alter zu bestimmen. Das Schlüsselwort `calculate` markiert diese Berechnung als reaktiv. Der Wert soll also immer automatisch neu berechnet werden,

sobald sich die Variablen verändern. In Zeile 4 wird das berechnete Alter mittels `print`-Anweisung ausgegeben. Das Schlüsselwort `effect` markiert auch diese Ausgabe als reaktiv, sodass sie automatisch erneut ausgeführt wird, sobald sich der ausgegebene Wert verändert. In Zeile 5 wird das aktuelle Jahr (`currentYear`) auf einen neuen Wert gesetzt. Die reaktive Programmierung führt dazu, dass hierdurch das Alter neu berechnet und erneut ausgegeben wird.

```
1 let yearOfBirth = state(1991)
2 let currentYear = state(2024)
3 let currentAge = calculate(currentYear - yearOfBirth)
4 effect(() => print(currentAge))  // => prints "33"
5 currentYear = 2025   // => prints "34"
```

Quelltext 2.2: Beispiel für reaktive Programmierung

2.5.3. Das Optik Entwurfsmuster

Optics ist ein Entwurfsmuster in der funktionalen Programmierung [MFP91, MW15]. Es kann vereinfacht gesagt als Analogie zum Facade-, Adapter- und Proxy-Pattern in der objektorientierten Programmierung betrachtet werden.

Vereinfacht ausgedrückt ist eine Optik ein Isomorphismus zwischen zwei Datenstrukturen. In der funktionalen Programmierung werden Optiken eingesetzt, um eine große Datenstruktur in kleinere Strukturen zu zerlegen und diese später wieder zu einer großen Struktur zusammenzusetzen.

Für die Entwicklung von Benutzeroberflächen können Optiken in Kombination mit reaktiver Programmierung verwendet werden, funktionale Abhängigkeiten zwischen verschiedenen Komponenten der Oberfläche zu definieren und alle einzelnen Komponenten zu einer einheitlichen Oberfläche zusammenzusetzen [CC13, Kar16a].

Quellcode 2.3 zeigt ein einfaches Beispiel für die Definition und Verwendung einer Optik. In Zeile 1 wird zunächst eine einfache Datenstruktur aus vier Feldern (`x`, `y`, `width` und `height`) zur Beschreibung eines Rechtecks definiert und mit Werten belegt. Zwischen Zeile 2 und 11 wird nun eine Optik (`bottomCorner`) definiert. Dafür wird ein Paar aus zwei Funktionen angegeben. Zwischen Zeile 3 und Zeile 5 wird eine lesende Funktion definiert. Diese bildet ein gegebenes Rechteck auf die Position der unteren rechten Ecke ab, indem die Koordinaten und Ausmaße entsprechend addiert werden. Zwischen Zeile 6 und Zeile 11 wird die zugehörige schreibende Funktion definiert. Diese verändert nicht wirklich die Belegung von Variablen, sondern bildet eine neue Position der rechten unteren Ecke (`newValue`) und ein gegebenes Rechteck (`oldRect`) auf ein neues Rechteck ab. Aus objektorientierter Programmierung lässt sich `bottomCorner` nun also als ein Objekt zur Modellierung eines Getter-Setter-Methodenpaares verstehen. In Zeile 13 wird die definierte Optik verwendet, um aus in Zeile 1 definierten Rechtecks die Position der unteren rechten Ecke auszulesen. In Zeile 15 wird ein neues Rechteck berechnet, indem eine neue Eckposition für das bestehende Rechteck gewählt wird.

```
let rectangle = {x:10, y:20, width: 50, height: 30}
let bottomCorner = Optik.define(
    (rect) => ({
        x: rect.x + rect.width, y: rect.y + rect.height
    }),
    (newValue, oldRect) => ({
        x: oldRect.x,
        y: oldRect.y
        width: newValue.x - oldRect.x,
        height: newValue.y - oldRect.y,
    }),
)
let currentCorner = Optik.read(bottomCorner, rectangle) // => {x: 60, y: 50}
let newRectangle = Optik.update(bottomCorner, {x: 30, y: 30}, rectangle)
// => newRectangle == {x:10, y:20, width: 20, height: 10}
```

Quelltext 2.3: Beispiel einer Optik-Definition in JavaScript

2.6. Svelte

Die vom Webbrowser bereitgestellten Schnittstellen zur Programmierung von interaktiven Benutzeroberflächen basieren auf prozeduralen und objektorientierten Paradigmen [con25b]. Schaltflächen und Eingabefelder werden als Objekte betrachtet, die sich mittels jeweiliger Methoden manipulieren lassen [con25b, ZCZ+14].

Für die Programmierung von Benutzeroberflächen im Webbrowser mittels funktionaler Programmierung ist eine Übersetzung der gegebenen Programmierschnittstellen in eine Schnittstelle mit funktionaler Semantik nötig [CN19, Gac15].

Svelte ist eine JavaScript-Bibliothek, die einen solchen Adapter mit funktionaler und reaktiver Semantik bereitstellt [con25a].

Quellcode 2.4 zeigt ein einfaches Beispiel einer in Svelte definierten Komponente. Zeile 1 bis Zeile 6 enthalten einen JavaScript-Block, durch den der Zustand der Komponente (`currentValue`) und mögliche Aktionen (`step`) definiert werden. Zeile 7 bis 14 definiert die HTML-Struktur, die von der Komponente erzeugt und ausgegeben werden soll. Diese kann sich auf den aktuellen Zustand beziehen, um diesen auszugeben. In Zeile 11 wird beispielsweise das `value`-Attribut des Eingabefeldes an den aktuellen Zustandswert der Komponente gebunden. Die in Zeile 13 und 14 definierten Schaltflächen sollen bei Mausklicks den aktuellen Wert inkrementieren oder dekrementieren. In den Zeilen 16 bis 18 wird noch ein Stylesheet für die Steuerung der Schriftfarbe definiert. Dieses wird in Zeile 9, bedingt durch den aktuellen Zustandswert, auf das Eingabefeld angewendet. Durch das Prinzip der reaktiven Programmierung ist es nur nötig, diese Datenabhängigkeiten deklarativ anzugeben. Svelte kümmert sich automatisch darum, dass bei Veränderung des Zustands alle Abhängigkeiten aktualisiert werden. Beispielsweise wird das Stylesheet automatisch aktiviert und die Schrift rot gefärbt, sobald der Eingabewert auf über Zehn erhöht wird.

```
<script>
    let currentValue = state(42)
    function step(delta) {
      currentValue = currentValue + delta;
    }
</script>

<input
    class:danger={currentValue > 10}
    type="number"
    bind:value={currentValue}   />

<button onclick={() => increment(+1)}>Increment</button>
<button onclick={() => increment(-1)}>Decrement</button>

<style>
    .danger { color: red; }
</style>
```

Quelltext 2.4: Beispiel einer Svelte-Komponente

2.7. Erlang

Erlang ist eine von Joe Armstrong bei Ericsson entwickelte funktionale Programmiersprache, die ursprünglich für die Programmierung von Telefonschaltanlagen konzipiert wurde [Arm10]. Sie ist stark durch die Logik-Programmiersprache Prolog inspiriert. Erlang wird auf der virtuellen Maschine Bogdan's Erlang Abstract Machine (BEAM) ausgeführt. Erlang verhält sich zur BEAM so wie Java zur Java Virtual Machine (JVM).

Ein besonderes Merkmal von Erlang ist der Umgang mit unerwarteten Programmabstürzen. Mittels des Konzepts der Supervisor-Bäume können in Erlang nebenläufige Prozesse sehr robust modelliert werden. Einzelne Komponenten einer Anwendung können abstürzen und neugestartet werden, ohne sich negativ auf den Rest der Anwendung auszuwirken [VWW96, Heb13].

Diese Art der Modellierung nebenläufiger Prozesse ist prinzipiell auch in Java möglich, beispielsweise seit Java 21 mittels Structured Concurrency [PB25]. Doch Erlang zeichnet sich dadurch aus, dass das gesamte Ökosystem aller Programmbibliotheken auf diese Art der Fehlerbehandlung ausgelegt ist. Somit ist es in Erlang besonders einfach, fehlertolerante verteilte Anwendungen zu entwickeln.

Quellcode 2.5 zeigt ein einfaches Beispiel zur Demonstration der Syntax von Erlang. In den ersten beiden Zeilen wird die Fakultätsfunktion rekursiv und mittels deklarativer Fallunterscheidung definiert. Danach wird 10! ausgerechnet. Abschließend wird die Fakultät für die ersten zehn natürlichen Zahlen berechnet: $\{(x, x!) \mid x \in \mathbb{N} \land 1 \leq x \leq 10\}$ als Liste mittels gesonderter Syntax zur Erzeugung von Listen berechnet.

```erlang
fact(0) -> 1;              %% Pattern matching for control-flow
fact(N) -> N * fact(N-1).  %% Recursion to create loops

example:fact(10).
3628800
[{I, example:fact(I)} || I <- lists:seq(1,10)].
%% => [{1, 1}, {2, 2}, {3, 6}, {4, 24}, {5, 120}, {6, 720},
%% {7, 5040}, {8, 40320}, {9, 362880}, {10, 3628800}]
```

Quelltext 2.5: Beispiel für die Erlang Syntax.

```elixir
defmodule Example do
  def fac(1), do: 1
  def fac(n), do
    n * fac(n-1)
  end
end

fac(10) #=> 3628800
(for x <- range(1, 10), do: {x, fac(n)})
# =>  [{1, 1}, {2, 2}, {3, 6}, {4, 24}, {5, 120}, {6, 720},
#      {7, 5040}, {8, 40320}, {9, 362880}, {10, 3628800}]
```

Quelltext 2.6: Beispiel für die Elixir Syntax.

2.8. Elixir

Elixir ist eine auf Erlang basierende funktionale Programmiersprache, die ebenfalls auf der BEAM ausgeführt werden kann [Jur24]. Elixir verhält sich zu Erlang so wie Kotlin zu Java [JI17]. Es ist also eine alternative Syntax, die zur Ausführung auf die gleiche virtuelle Maschine wie Erlang übersetzt wird. Zur Laufzeit unterscheidet sich die Semantik von Elixir nicht von Erlang.

Sie bietet hauptsächlich drei Vorteile gegenüber Erlang. Zum einen orientiert sich die Syntax von Elixir stärker an verbreiteteren Sprachen wie Python oder Ruby, wohingegen die Syntax von Erlang stark von Prolog beeinflusst ist und für viele Programmierer ungewohnt sein mag [Bra05, FM08, VR+07]. Außerdem bringt Elixir ein mächtiges, von Scheme und Lisp inspiriertes Makrosystem mit sich [McC78, Jur24]. Damit ist es möglich, umfangreiche Programmlogik in prägnanter Syntax zu formulieren. Zudem ermöglicht Letzteres auch die geschickte Verwendung und Komposition von bestehenden Bibliotheken mittels eingebetteter Domainspecific Language (domänenspezifische Sprache, DSL).

Mit dem Build-Werkzeug Mix und der Paketverwaltung Hex bringt Elixir ein großes Ökosystem an mächtigen Programmbibliotheken mit sich [Leo14, MJ17].

Quellcode 2.6 zeigt ein einfaches Beispiel zur Demonstration der Elixir-Syntax. Genau wie in dem Erlang-Beispiel aus Quellcode 2.5 wird die Fakultätsfunktion definiert. Die Syntax dürfte mit der Ähnlichkeit zu Python und Ruby aber einer größeren Menge an Menschen vertrauter erscheinen.

2.9. Phoenix Web Framework

Das Phoenix Web Framework ist eine Elixir-Bibliothek, die eine Vielzahl von Modulen für die serverseitige Programmierung einer Webanwendung bereitstellt [TMV19]. Phoenix kann verglichen werden mit dem Java Spring Framework, mit Rails für Ruby oder mit Django für Python [MOC13, HT09, FBC08].

Die Basis aus Erlang und Elixir bringt für das Phoenix Framework einige Vorteile mit sich. Die rein funktionale Semantik der BEAM erleichtert es besonders, den sonst oft komplexen Daten- und Kontrollfluss in einem so umfangreichen Framework nachzuvollziehen und Fehlerquellen zu identifizieren. Die von der BEAM vorgegebene Semantik für nebenläufige Prozesse und deren Fehlerbehandlung erlaubt es dem Phoenix Framework und den darauf basierenden Webanwendungen, besonders robust im Umgang mit Programmier- und Netzwerkfehlern zu sein. Die Syntax von Elixir und dem zugehörigen Makrosystem fördert das Schreiben von sehr leserlichem Programmcode, der sich auf die Modellierung der Geschäftsprozesse fokussiert.

2.10. SQLite

SQLite ist ein relationales Datenbanksystem, welches sich direkt als Bibliothek in eine Anwendung integrieren lässt [OA10]. SQLite wurde von Richard Hipp entwickelt und erstmals im Jahr 2000 veröffentlicht [GPB+22, Ric07a]. Es ist das verbreitetste und meistverwendete Datenbanksystem der Welt, da es sowohl in allen gängigen Webbrowsern (Firefox, Safari, Chrome) als auch in allen Betriebssystemen von Mobilgeräten (iOS, Android, Windows Phone) integriert ist [Ric07b].

Im Gegensatz zu anderen Datenbanksystemen ist es kein verteiltes Client-Server-System, bei welchem die Daten von einem externen Datenbankserver verwaltet werden [Ric07c]. Die von SQLite gespeicherten Daten liegen entweder direkt im Arbeitsspeicher des verwendenden Prozesses oder in einer Datei, auf die nur der jeweilige Prozess zugreift.

Insofern lässt sich SQLite für die Datenpersistenz eher als Alternative zu fopen verstehen. Zur Laufzeit stellt SQLite eine Alternative für die manuelle Speicherverwaltung oder die manuelle Verwaltung von Datenstrukturen dar.

Der Vorteil von SQLite gegenüber manueller Verwaltung und Serialisierung von Datenstrukturen ist also die Ausdrucksstärke des relationalen Datenmodells samt Mengenoperationen, Transaktionsemantik, Indizierung und zentraler Validierung von Wertebereichen [Ric07a, Ric07d].

2.11. Objekt-Relationale-Abbildung

Ein typischer Kritikpunkt an der Verwendung von relationalen Datenbanken und relationalen Modellen für die Anwendungsprogrammierung ist die Unstimmigkeit zwischen

der objektorientiert strukturierten Sicht und der relationalen Sicht auf Daten [Atwo6]. Um in einer Programmiersprache wie Java verarbeitet werden zu können, müssen Daten in Form von Objekten vorliegen. Objekte können sich gegenseitig mittels Speicheradresse referenzieren und erzeugen so einen gerichteten Graphen, den Referenzgraphen. Für die Speicherung in einem relationalen Schema müssen die Daten in eine Menge an Tupeln überführt werden [TGPM17].

Diese Transformation zwischen den beiden Sichten wird Objekt-Relationale-Abbildung (Object Relational Mapper (ORM)) genannt und ist in vielen Fällen sehr aufwändig und fehleranfällig. Die Schwierigkeit liegt darin, dass sich die Semantik einer objektorientierten Programmiersprache nicht eindeutig auf die Semantik einer relationalen Datenbank abbilden lässt.

2.12. Ecto

Ecto ist ein Datenbankadapter für die Verwendung von relationalen Datenbanken in Elixir und Erlang [WMJ19]. Im Gegensatz zu Objekt-Relationalen-Mappern in objektorientierten Sprachen wie Java oder Python ist es Ecto gelungen, eine sehr klare und gut strukturierte Abbildung des relationalen Modells auf die Elixir- und Erlang-eigenen Datentypen zu definieren.

Ecto nutzt hierbei sowohl die rein funktionale Semantik von Erlang als auch die mächtigen Syntaxmakros von Elixir, um eine ausdrucksstarke Schnittstelle zwischen Elixir und einer relationalen Datenbank wie SQLite, aber wahlweise auch MySQL oder PostgreSQL bereitzustellen.

Quellcode 2.7 zeigt exemplarisch die Definition eines Datenbankschemas in Ecto zusammen mit einer zugehörigen Abfrage. Es werden die beiden Entitätstypen `User` und `Book` als Elixir-Modul definiert. Jeweils wird in Zeile 4 und 23 ein Schema definiert, um anzugeben, aus welchen Feldern die beiden Entitätstypen bestehen. Für die einzelnen Felder werden in der jeweiligen `changeset`-Definition in Zeile 10 und 29 noch zugehörige Gültigkeitsbedingungen vorgegeben, beispielsweise, dass jeder `email`-Wert eindeutig sein muss (Zeile 15). Außerdem wird mittels `has_many` (Zeile 7) und `belongs_to` (Zeile 26) Anweisungen eine $(1 : n)$-Beziehung zwischen Benutzerinnen und Büchern deklariert. Abschließend wird ab Zeile 38 in der von Ecto bereitgestellten Structured Query Language (Strukturierte Abfrage-Sprache, SQL)-basierten Syntax eine Abfrage formuliert und an das Repositorium gesendet, um zehn Bücher einer Benutzerin namens Frieda auszulesen.

```elixir
defmodule User do
  use Ecto.Schema

  schema "user" do
    field :name, :string
    field :email, :string
    has_many :books,  Book
  end

  def changeset(user, attrs) do
    user
    |> cast(attrs, [:name, :email])
    |> cast_assoc(:books)
    |> validate_required([:name])
    |> unique_constraint(:email)
  end
end

defmodule Book do
  use Ecto.Schema
  import Ecto.Changeset

  schema "book" do
    field :name, :string
    field :isbn, :string
    belong_to :user,  User
  end

  def changeset(book, attrs) do
    book
    |> cast(attrs, [:name, :isbn])
    |> validate_required([:name, :isbn])
    |> unique_constraint(:isbn)
  end
end

def friedas_books() do
    from(
        u in User,
        join: b in assoc(u, :books),
        where: user.name == "Frieda",
        select: u.books,
        limit: 10
    )
    |> Repository.all()
end
```

Quelltext 2.7: Beispiel für Ecto Schema Definition und Query

3. Verwandte Arbeiten und Stand der Forschung

Im Folgenden werden einige andere Arbeiten vorgestellt, die sich ebenfalls schon mit der Benutzeroberfläche von Renew oder vergleichbaren Modellierungswerkzeugen befasst haben.

3.1. Anbinden mehrerer Benutzeroberflächen an Renew

Marc Richter und Tim Kilian haben in ihren jeweiligen Bachelorarbeiten bereits 2019 versucht, einen Web-Editor für die Bearbeitung von Renew-Netzen zu erstellen [Ric19, Kil19]. In Tim Kilians Arbeit lag der Fokus aber auf der Erarbeitung einer zugehörigen Plugin-Architektur für Metamodellierung [Kil19]. Die Zielsetzung von Marc Richters Arbeit kommt dem Ziel dieser Arbeit vermutlich am nächsten. Allerdings haben sich sowohl Richter als auch Kilian dem begrenzten Zeitrahmen, der ihnen zur Verfügung stand, dafür entschieden, eine bestehende JavaScript-Bibliothek zum Zeichnen von Graphen zu verwenden und eine dazu passende Schnittstelle zu Renew zu entwickeln. Das Ergebnis von Richters und Kilians Arbeit konnte erfolgreich demonstrieren, wie sich Renew-Dokumente in einer Webanwendung bearbeiten und simulieren lassen. Jedoch konnte der von ihnen entwickelte Editor sich mangels Funktionsumfang nicht als Werkzeug im Produktiveinsatz etablieren und wurde auch später nicht weiterentwickelt. Es kann vermutet werden, dass die von Richter und Kilian gewählte objektorientierte Softwarearchitektur die Weiterentwicklung erschwert hat.

Leon Meyer hat in seiner Bachelorarbeit 2023 untersucht, wie sich unter Verwendung von Web-Technologien eine alternative Benutzeroberfläche an Renew anschließen lässt [Mey23]. Seine Experimente haben zwar demonstriert, wie dies prinzipiell umgesetzt werden kann, allerdings ist darüber hinaus kein verwendbares Softwareprodukt entstanden, welches von Endnutzerinnen und Endnutzern eingesetzt werden kann.

Lukas Voß hat in seiner Masterarbeit 2024 untersucht, wie sich Renew in ein System aus Microservices zerlegen lässt [Voß24]. Dafür hat er die Benutzeroberfläche in einen eigenen Service abgetrennt und für einen seiner Prototypen auch eine webbasierte Benutzeroberfläche entwickelt. Ihm ist es dabei gelungen, die zyklischen Abhängigkeiten zwischen der Renew-Graphical User Interface (grafische Benutzeroberfläche, GUI) aufzulösen und Kommunikationsprotokolle zwischen GUI und Simulator zu definieren. Er hat dafür einige Modellierungstechniken angewendet, die auch in dieser Arbeit später zum Einsatz kommen werden, wie etwa die Einführung von Universally Unique Identifier (UUID) oder die Delta-Kodierung zur Kommunikation von Änderungen zwischen Modulen. Sein Fokus lag aber, im Gegensatz zu dieser Arbeit, auf der Erkundung der

Microservice-Architektur für Renew und nicht zentral auf der Verbesserung der Benutzeroberfläche.

Martin Wincierz hat in seiner Masterarbeit 2014 die bestehende Benutzeroberfläche von Renew hinsichtlich verschiedener Aspekte überarbeitet und modernisiert [Win18, WMH18]. Durch seine Arbeit wurde die Zeichenfläche von Renew um eine Minimap und ein Zoomwerkzeug erweitert. Außerdem hat Wincierz die gesamte Fensterverwaltung von Renew überarbeitet, sodass sich verschiedene Dokumente und Werkzeugleisten effizienter anordnen lassen.

3.2. Entwicklung von Web-Anwendungen Mittels Renew

Eva Müller hat in ihrer Diplomarbeit 2016 ein JavaScript-Framework für die Entwicklung von agentenorientierten Webanwendungen erarbeitet [Mül16]. Das Framework von Eva Müller soll also die Entwicklung von Webanwendungen mittels Renew unterstützen. Wohingegen in dieser Arbeit eine Webanwendung zur Bearbeitung von Renew-Dokumenten entwickelt werden soll.

Auch Tobias Betz hat sich zusammen mit Lawrence Cabac und Michael Duvigneau damit befasst, Renew die agentenorientierte Modellierung von Webservices zu verwenden [Bet11, BCWE11, BCD+13, BCD+14].

Wenn Renew selbst dafür verwendet werden kann, Webanwendungen zu modellieren, könnte man versuchen, den in dieser Arbeit entwickelten Editor selbst mit Hilfe und auf Basis von Renew zu modellieren und zu entwickeln. Das wäre zwar ein interessantes Experiment, wird aber im Rahmen dieser Arbeit nicht weiter verfolgt.

3.3. Kollaboratives Arbeiten in Renew

Michael Haustermann und Julian Burkhart haben mit PetriPad bereits 2012 versucht Renew um kollaborative Werkzeuge zu erweitern [BH12]. Damit war es mehreren Benutzerinnen und Benutzern möglich, aus ihrer jeweiligen Renew-Installation heraus gemeinsam an einem Dokument zu arbeiten. Das kollaborative Arbeiten wurde damit zwar ermöglicht, allerdings ist die Bedienung der Kollaborationswerkzeuge in der Renew-Benutzeroberfläche verglichen mit typischen Web-Editoren eher umständlich und somit nur für Renew-Experten zu verwenden. Außerdem ist die Verwendung von Renew auf Mobilgeräten weiterhin nicht möglich.

3.4. Petrinetze auf Mobilgeräten

Dominic Dibbern hat 2012 eine Strategie zur Portierung von Renew auf Android entwickelt [Dib12a, Dib12b]. Dabei wurde aber die Benutzeroberfläche ausgelassen, da die

nötige Java-Programmbibliothek AWT, auf der die Renew Benutzeroberfläche basiert, unter Android grundsätzlich nicht zur Verfügung steht.

Thomas Irgang et al. haben 2012 mit Multitouch Petrinet Simulator (MuPSi) einen Petrinetz-Simulator entwickelt, der sich auf einem Touchscreen per Multitouch-Handgesten bedienen lässt [IHB12]. Dabei kam heraus, dass sich Multitouch-Fingergesten gut eignen, um manuelle und simultane Schaltvorgänge von Transitionen während einer Simulation auszulösen. Somit kann die Arbeit als zusätzliche Motivation betrachtet werden, um eine Multitouch-freundliche Benutzeroberfläche für Renew zu konzipieren.

3.5. Event Coordination Notation

Ekkart Kindler hat 2014 mit der Event Coordination Notation (ECNO) ein Metamodellierungswerkzeug entwickelt, mit dessen Hilfe sich unter anderem Petrinetz-Editoren und -Simulatoren erstellen lassen [Kin12a, Kin12c, Kin12b, Kin11]. ECNO basiert allerdings auf dem Eclipse Modelling Framework (EMF) und kann daher genau wie Renew nicht einfach auf Mobilgeräten ausgeführt werden [SBMP08, Bud04]. Inwieweit sich die mittels ECNO modellierten Systeme um kollaborative Werkzeuge erweitern ließen, wurde nicht weiter untersucht.

3.6. Weitere Editoren zur Prozessmodellierung

Online lassen sich eine Vielzahl an webbasierten Petrinetz-Editoren und Simulatoren finden [Bar24, Jag12, Kim17, Ans18, Unb24, SS18, BK22].

Sie bieten aber jeweils entweder keine Möglichkeit, kollaborativ zu arbeiten, sind nur sehr beschränkt in ihrem Funktionsumfang (keine Undo/Redo-Historie, kein Zoom) und bieten nur wenig Freiheit in der Modellierung von Netzen (nur Integer-Marken auf Plätzen). Keiner dieser Editoren bietet eine ausdrucksstarke Semantik, die mit Referenznetz-Simulator von Renew ansatzweise vergleichbar wäre.

Abgesehen von den webbasierten Editoren existieren neben Renew auch noch einige Desktopanwendungen als Entwicklungsumgebungen für Petrinetze. Zu ihnen gehören PIPE2, Wolfgang und die Colored Petri Nets (CPN)Tools sowie deren Nachfolger Projekt CPN Integrated Development Environment (IDE) [RWL+03, VF21, Lan16, DKS09, CK10, BLP+07]. Diese Projekte werben zwar teilweise damit, plattformunabhängig (PIPE2, Platform Independent Petri Net Editor 2, [RWL+03]) zu sein, doch in der Praxis unterliegen sie den gleichen Einschränkungen wie Renew. Als Java-Anwendung lassen sie sich theoretisch auf jedem System ausführen, auf dem eine JVM zur Verfügung steht. Auf Mobilgeräten lässt sich der jeweilige Editor aber nicht verwenden, weil Java auf iOS nicht unterstützt wird oder die Benutzeroberfläche gar nicht mit der Verwendung per Touchscreen kompatibel ist.

3.7. Quelloffene Bibliotheken

Neben den Petrinetz-Editoren fällt als webbasiertes Modellierungswerkzeug auch bpmn.io mit seiner modernen Benutzeroberfläche ins Auge [RFS+13e]. bpmn.io bietet Werkzeuge zur Bearbeitung von Process Model and Notation (BPMN), Decision Model and Notation (DMN), Case Management Model and Notation (CMMN) und Formularen. Die zugehörigen a JavaScript-Bibliotheken (bpmn-js [RFS+13a], dmn-js [RFS+13c], cmmn-js [RFS+13b] und form-js [RFS+13d]) stehen quelloffen unter der MIT-Lizenz zur Verfügung.

Auch `Joint.js` ist ein zumindest teilweise quelloffenes Framework zur Erstellung von webbasierten Graph-Editoren [MDW+24].

Mit so viel Auswahl an quelloffenen Bibliotheken liegt es nahe, sich für eine von ihnen zu entscheiden und diese per passender Schnittstelle als fertige Benutzeroberfläche an Renew anzubinden. Dann spart man sich die Arbeit, all die aufwändigen Details einer Benutzeroberfläche neu zu erfinden.

Diese Strategie ist es, die Marc Richter und Tim Kilian in deren zuvor schon erwähnten Bachelorarbeit verfolgt hatten [Ric19, Kil19]. Sie hatten sich beide für die `diagram-js` Bibliothek entschieden, um ihren Webeditor für Renew zu konstruieren. Diese liegt auch dem `bpmn.io` Werkzeug zugrunde. Zwar konnten die Beiden so einerseits einiges an Arbeit sparen, aber umso mehr Aufwand mussten sie in die Entwicklung der Schnittstelle zwischen den zwei vollkommen unterschiedlichen Datenmodellen von Renew und `diagram-js` stecken. In dieser Arbeit soll daher auf die Verwendung solcher fertigen Graph-Editor-Bibliotheken verzichtet werden. Stattdessen sollen Werkzeuge wie `bpmn.io` rein optisch als Inspiration fungieren.

3.8. Weitere Zeichenwerkzeuge und Grafikeditoren

Unter allen digitalen Zeichenwerkzeugen und Graph-Editoren und Anwendungen mit grafischer Benutzeroberfläche lassen sich viele weitere Projekte finden, die optisch, funktional oder auch hinsichtlich ihrer Software-Architektur als Inspiration für eine grafische Benutzeroberfläche für Renew dienen können.

Adobe Photoshop [Müh20], Adobe Illustrator [Müh20], Figma [Fig16] und Affinity Designer [Eur14] sind vier proprietäre Projekte und keine Modellierungswerkzeuge, sondern reine Grafikprogramme. Doch viele durch sie definierte Arbeitsabläufe wie das Hinein- oder Herauszoomen der Zeichenebene oder die Anordnung von Elementen in einer verschachtelten Ebenen-Hierarchie wurden durch die Verbreitung dieser Grafikprogramme so stark geprägt, dass es sich in der Gestaltung von Benutzerinteraktionen an ihnen zu orientieren gilt, um Benutzer:innen ein vertrautes Erlebnis zu bieten.

Auch verbreitete 3D-Modellierungsanwendungen wie Blender und Godot beinhalten Komponenten zum Editieren von gerichteten Graphen mit ausgefeilter Benutzeroberfläche [Fou94, Eng14]. Auch hier ergibt es durchaus Sinn, sich inspirieren zu lassen.

Teil II.

Konzeption

4. Anforderungen

In diesem Kapitel werden die genauen Anforderungen für den Web-Editor erarbeitet. Wie schon in der Einleitung und Motivation erklärt, soll eine Software-Anwendung entstehen, die sich in der Praxis ergänzend zu der existierenden Renew-Entwicklungsumgebung für das Bearbeiten und Simulieren von Referenznetzen verwenden lässt.

4.1. Renew Dateien lesen und schreiben

In Renew erstellte Netze lassen sich in verschiedene Dateiformate exportieren. Das Petri Net Markup Language (PNML) Format ist ein XML-basiertes Standardformat zur Speicherung und zum Austausch von Petrinetz-Dokumenten [HKPT10]. Renew besitzt aber auch ein eigenes Dateiformat, welches üblicherweise verwendet wird, um Arbeitsergebnisse in Renew abzuspeichern. Das Renew-eigene Dateiformat wird mit der Dateiendung .rnw gekennzeichnet.

Damit der Web-Editor vollwertig ergänzend zu Renew verwendet werden kann, muss er in der Lage sein, in Renew erzeugte .rnw-Dateien einzulesen. Außerdem muss der Web-Editor die erstellten und bearbeiteten Netze auch in Form einer .rnw-Datei exportieren können, damit in Renew-weitergearbeitet werden kann.

Wie in Kapitel 2 „Grundlagen“ bereits erklärt, unterstützt Renew nicht nur die Arbeit mit Referenznetzen, sondern auch viele weitere Formalismen. Der Web-Editor soll in der Lage sein, jegliche von Renew erstellte Dokumente einzulesen und darzustellen, um einen robusten Umgang mit .rnw-Dateien zu gewährleisten. Allerdings werden hinsichtlich der Werkzeuge zum Bearbeiten die Anforderungen auf das Bearbeiten von Referenznetzen beschränkt.

4.2. Benutzbarkeit auf Desktop- und auf Mobil-Geräten

In der Motivation für diese Arbeit wurde die fehlende Benutzbarkeit von Renew auf Mobilgeräten bereits bemängelt. Dass der nun entwickelte Web-Editor sich sowohl auf Desktop als auch auf Mobilgeräten wie Smartphone und Tablet verwenden lässt, gehört also zu den Kernanforderungen.

Besonders wichtig hierbei ist, dass der Editor sich nicht nur prinzipiell auf einem Mobilgerät ausführen lässt, sondern auch angenehm auf einem Touchscreen bedienen lässt.

4.3. Kollaboratives Arbeiten

Auch Werkzeuge zum gemeinsamen Arbeiten mehrerer Benutzer:innen an einem Renew-Dokument wurden bereits in der Motivation für diese Arbeit gefordert. Es soll also möglich sein, dass zwei oder auch mehr Benutzer:innen auf ihrem jeweiligen Arbeitsgerät gleichzeitig auf dasselbe Dokument zugreifen können.

Benutzer:innen sollen dabei Änderungen am Dokument vornehmen können und die jeweils von anderen Personen vorgenommenen Änderungen in Echtzeit beobachten können. Außerdem soll nachvollziehbar sein, wer zum jeweiligen Zeitpunkt in einem Dokument arbeitet und auf welchem Bereich des Dokuments der Fokus der jeweiligen Person liegt.

4.4. Bearbeitung hierarchischer Dokumente

Ein in Renew erzeugtes Dokument besteht nicht nur aus einer ungeordneten Menge an Elementen. Die Netzelemente, Plätze, Transitionen und Kanten sind logisch miteinander verknüpft und können auch hierarchisch in Gruppen zusammengefasst werden.

In anderen Petrinetz-Werkzeugen, die im Kapitel 3 „Verwandte Arbeiten und Stand der Forschung“ vorgestellt wurden, wird so eine hierarchische Organisation von Elementen teilweise nicht unterstützt. Für die Arbeit mit Renew-Dokumenten ist dies aber wichtig. Daher ist es eine Anforderung für den Web-Editor, dass hierarchische Dokumente eingelesen, erstellt und bearbeitet werden können.

4.5. Bearbeitung von Elementen

Ein Renew-Petrinetz-Dokument zu bearbeiten bedeutet, die sich darin befindenden Elemente zu bearbeiten. Referenznetze setzen sich aus Plätzen, Transitionen, gerichteten Kanten und Textanschriften zusammen. Jedes dieser Elemente kann mit visuellen Attributen versehen werden und logisch mit anderen Elementen verknüpft werden.

Der Web-Editor soll Werkzeuge anbieten, um diese Elemente zu erstellen und deren visuelle Attribute sowie logische Verknüpfungen zu bearbeiten.

4.6. Undo/Redo-Historie

Um in der Praxis effizient an einem Dokument arbeiten zu können, ist es wichtig, vorgenommene Änderungen auch wieder rückgängig machen zu können. Besonders beim kollaborativen Arbeiten kann es schnell passieren, dass ungewollte Änderungen versehentlich vorgenommen werden. Den Benutzer:innen muss also eine einfache Möglichkeit geboten werden, wie aus anderen Softwareanwendungen gewohnt, Änderungsoperationen rückgängig zu machen.

Die Erfüllung dieser Anforderung kann oftmals nur umständlich nachgerüstet werden, da sie sich auf viele andere Funktionen einer Software-Anwendung auswirkt. Viele der einfacheren Web-Editoren für Petrinetze, die in Kapitel 3 „Verwandte Arbeiten und Stand der Forschung“ vorgestellt wurden, bieten beispielsweise keine solche Arbeitshistorie. Für die Entwicklung eines in der Praxis einsetzbaren Editors muss diese Anforderung also von Beginn an mitbedacht werden.

4.7. Visuelle Navigation im Dokument

In der Motivation für diese Arbeit wurde schon darauf hingewiesen, dass die Benutzeroberfläche von Renew einige Eigenheiten mit sich bringt, die die Bedienung erschweren. Ein ebenfalls in der Motivation genanntes Beispiel ist die Möglichkeit, dass sich im Dokument befindliche Dokumente aus dem Sichtbereich der Benutzerin oder des Benutzers herausschieben lassen und dadurch dauerhaft unzugänglich werden.

Im Allgemeinen funktioniert in Renew die visuelle Navigation in einem geöffneten Dokument nicht so reibungslos, wie man es aus anderen Grafikanwendungen gewohnt sein könnte. Beim Herein- und Herauszoomen in einem Dokument verschiebt sich hin und wieder das Zentrum des sichtbaren Bereichs der Zeichenfläche auf unerwartete Weise.

Die Navigation des Sichtbereichs in einem Dokument mittels Fingergesten auf einem Touchscreen funktioniert auch in einigen anderen verbreiteten Grafikanwendungen oft nicht reibungslos.

Daher wird als weitere Anforderung aufgestellt, dass Benutzer:innen sowohl per Maus und Tastatur als auch per Fingergesten auf einem Touchscreen reibungslos in einem im Web-Editor dargestellten Dokument navigieren können.

4.8. Simulation von Referenznetzen

Ein besonderes Merkmal von Renew gegenüber anderen Petrinetz-Werkzeugen ist der Simulator mit der zugehörigen Referenznetzsemantik. Ohne den Simulator wäre Renew nur ein einfacher Grafikeditor. Entsprechend wäre auch der Web-Editor nur ein einfacher Grafikeditor, wenn er nicht an den Simulator von Renew angebunden wird.

Damit der Web-Editor als vollwertiges Werkzeug zur Modellierung von Referenznetzen eingesetzt werden kann, müssen sich die gezeichneten Netze auch direkt aus dem Editor heraus simulieren lassen.

Einige der in Kapitel 2 „Grundlagen“ vorgestellten webbasierten Petrinetz-Editoren unterstützen zwar die Simulation einfacher Petrinetze im Webbrowser, doch keiner unterstützt die vollwertige Referenznetzsemantik von Renew.

Eine Anforderung an den Web-Editor ist es also, die gezeichneten Netze im Browser mittels vollwertiger Referenznetzsemantik simulieren zu können, ohne dass für eine:n Benutzer:in eine lokale Installation von Renew nötig ist.

4.9. Eingrenzung des Funktionsumfangs

Um in dem Zeitrahmen dieser Arbeit zu einem sinnvollen Ergebnis gelangen zu können, werden einige Funktionsmerkmale, die potenziell für einen Web-Editor denkbar wären, im Vorfeld ausgeschlossen. Damit soll es möglich sein, den Arbeitsaufwand zielgerichtet zu fokussieren.

4.9.1. Offline Arbeiten

Ein zentrales Funktionsmerkmal des Web-Editors sollen die Werkzeuge zum kollaborativen Arbeiten sein. Das kollaborative Arbeiten ist nur möglich, wenn die Geräte der arbeitenden Person in einem Netzwerk, etwa dem Internet, miteinander verbunden sind. Renew lässt sich grundsätzlich ohne nötige Internetverbindung verwenden.

Für den Web-Editor stellt sich also die Frage, ob dieser sich auch ohne Internetverbindung verwenden lassen soll und eine Internetverbindung nur für die kollaborative Arbeit nötig ist, oder ob für die Verwendung des Web-Editor grundsätzlich eine Netzwerkverbindung vorausgesetzt wird, selbst wenn ein:e Benutzer:in alleine an einem Projekt arbeitet.

Im Rahmen dieser Arbeit wird der Fokus auf einen reinen Online-Editor gelegt. Es wird also nicht möglich sein, den Web-Editor ohne entsprechende Netzwerkverbindung zu verwenden.

4.9.2. Mit Renew inkompatible Dokumente

Der Web-Editor soll nicht um Werkzeuge erweitert werden, mittels derer sich Dokumente erstellen lassen, die sich in Renew zum Zeitpunkt der Entwicklung auch nicht erstellen lassen.

In Renew ist es beispielsweise nicht möglich, den Plot eines Funktionsgraphen in ein Dokument einzufügen. Die neue Benutzeroberfläche des Web-Editors könnte es nun ermöglichen, die Erstellung solcher Elemente, die sich in Renew bisher nicht einfach umsetzen ließen, direkt in Web-Editor zu integrieren. Im Rahmen dieser Arbeit soll der Fokus aber nur darauf liegen, die von Renew bereits unterstützte Struktur an Dokumenten zu bearbeiten.

4.9.3. Detaillierte Fehlerbehandlung

Bei der Modellierung von Petrinetzen in Renew können eine Vielzahl an Fehlern auftreten, über welche die Anwender:innen informiert werden und darauf reagieren müssen. Beispielsweise kann die Textanschrift an einem Netzelement syntaktisch fehlerhaft sein. Eine ausgewählte Operation kann aus verschiedenen Gründen fehlschlagen. Ein Programmierfehler kann dazu führen, dass die Anwendung abstürzt.

Idealerweise werden Benutzer:innen detailliert über den jeweils aufgetretenen Fehlerfall informiert, um angemessen darauf reagieren zu können und in der Lage zu sein, den Fehler gegebenenfalls selbst zu beheben.

Die Behandlung aller möglichen Fehlerfälle in jedem Anwendungsszenario und die dazu passende verständliche Ausformulierung einer Meldung ist mit hohem Aufwand verbunden. Daher wird im begrenzten Zeitrahmen dieser Arbeit auf die ausführliche Behandlung von Fehlerfällen und auf hilfreiche Fehlermeldungen verzichtet.

5. Strategie zur Umsetzung

Nachdem im vorherigen Kapitel ein Katalog an Anforderungen für den Web-Editor aufgestellt wurde, wird im Folgenden eine Strategie für deren Erfüllung erarbeitet. Die Strategie zielt darauf ab, die Anforderungen mit jeweils möglichst geringem Arbeitsaufwand zu erfüllen. Außerdem wird versucht, so vorzugehen, dass sich Teilergebnisse dieser Arbeit auch dann sinnvoll weiterverwenden lassen, falls das letztendliche Ziel, einen Web-Editor zu entwickeln, nicht erreicht wird.

5.1. Keine Änderungen an Java Renew

Der Web-Editor soll als eigenständige Anwendung unabhängig vom Entwicklungsprozess von Renew entwickelt werden. Zum einen soll durch dieses Vorgehen die notwendige Koordination mit dem Renew-Entwicklungsprozess minimiert werden. Zum anderen soll verhindert werden, dass Renew zugunsten der Kompatibilität an Anforderungen des Web-Editors angepasst wird. Probleme bei der Interaktion zwischen Web-Editor und Renew werden zwar in dieser Arbeit dokumentiert, aber in Renew nicht behoben.

5.2. Import und Export von rnw-Dateien

Die Kompatibilität zwischen dem Web-Editor und Renew wurde bereits als eine zentrale Anforderung für diese Arbeit benannt. Auch unabhängig davon, ob der Web-Editor erfolgreich umgesetzt werden kann, ist es ein sinnvoller Meilenstein, das Renew-Dateiformat auch außerhalb von Renew verarbeiten zu können. Daher wird als ein Grundbaustein für den Web-Editor ein eigenständiges Modul zum Lesen und Schreiben von rnw-Dateien entwickelt.

5.3. Anbindung an den Renew Simulator

Die Anbindung des Web-Editors an den Renew-Simulator wurde als weitere wichtige Anforderung genannt, um den Web-Editor von anderen webbasierten Grafikanwendungen zu differenzieren. Ohne Renew selbst zu verändern oder um zusätzliche Schnittstellen zu erweitern, wird die Art und Weise der Simulatoranbindung durch die von Renew bereits angebotenen Schnittstellen bestimmt sein. Eine dieser Schnittstellen ist die Steuerung des Simulators per Textbefehlszeile. Wie vielseitig diese Schnittstelle ist, wird zu untersuchen sein.

5.4. Webanwendung mittels Client-Server-Architektur

Der Web-Editor soll als Client-Server-Anwendung entwickelt werden. Zwar verhindert dieser Ansatz die Verwendung des Editors ohne eine verfügbare Netzwerkverbindung, allerdings bringt die Client-Server-Architektur auch einige Vorteile mit sich. Der Server kann als zentraler Kommunikationsknoten sicherstellen, dass alle Änderungen an einem Dokument einheitlich und korrekt durchgeführt werden, ohne dass unerwartete Konflikte entstehen.

Außerdem können im Rahmen dieser Arbeit die Server- und die Client-Komponente jeweils getrennt voneinander entwickelt werden. Das hält die Option offen, eine der beiden Komponenten später noch auszutauschen.

In PetriPad wurde ja schon gezeigt, wie sich Renew per Client-Server-Architektur in eine kollaborative Arbeitsumgebung erweitern lässt. Wenn sich der Editor erfolgreich umsetzen lässt, wäre es beispielsweise zukünftig möglich, eine alternative Server-Komponente mit entsprechender Schnittstelle direkt in Renew als Plugin zu integrieren. Alternativ könnte Renew auch um ein Plugin erweitert werden, sich als Client zu dem in dieser Arbeit entwickelten Editor-Server zu verbinden.

Eine zur Client-Server-Architektur alternative Strategie wäre eine P2P=Architektur, in der kein zentraler Server-Knoten existiert, sondern mehrere Instanzen von Editor-Anwendungen direkt miteinander kommunizieren, um kollaborativ an einem Dokument zu arbeiten. Dies wäre beispielsweise auf Basis des WebRTC-Protokolls zusammen mit Conflict-free replicated data type (CRDT)- oder Operational Transforms (OT)-Datenstrukturen umsetzbar. Dafür müsste eine passende verteilte Datenstruktur zur Repräsentation von Renew-Dokumenten entworfen werden. In Anbetracht des gesetzten Ziels, einen in der Praxis vollwertig einsetzbaren Editor zu entwickeln, wird dies im begrenzten Zeitrahmen dieser Arbeit als zu aufwändig und riskant angesehen.

5.5. Relationale Modellierung

Ein besonderer Fokus dieser Arbeit wird auf die Modellierung des Datenmodells zur Repräsentation der im Editor bearbeitbaren Dokumente gelegt. Das Datenmodell gibt vor, welche Dokumente und Zustände vom Editor überhaupt repräsentiert werden können. Außerdem hat es auch einen Einfluss darauf, wie einfach fehlerhafte Zustände erkannt und korrigiert werden können.

Im Allgemeinen ist ein Referenznetz in Renew ein gerichteter Graph mit Platz- und Kantenanschriften. Ein Renew-Dokument kann aber auch weitere Elemente wie Freitext, Vektor-Symbole oder eingefügte Grafiken enthalten. Jedes fehlerfreie in Renew präsentierbare Dokument muss auch in dem Datenmodell des Web-Editors darstellbar sein.

Ein Beispiel für ein Dokument in einem fehlerhaften Zustand ist ein Dokument, welches eine Kante enthält, die zwei Knoten miteinander verbindet, wobei die beiden Knoten aber selbst nicht im Dokument enthalten sind. Eine mögliche Korrektur wäre das Entfernen

dieser Kante. Eine alternative Korrektur wäre das Hinzufügen der fehlenden Knoten in das Dokument. Im Idealfall ist ein solch fehlerhaftes Dokument in dem Datenmodell des Editors gar nicht darstellbar.

In Renew werden Dokumente als Graph aus sich gegenseitig referenzierenden Java-Objekte repräsentiert. Auf diese Weise sind vielerlei ungültige Zustände darstellbar. Diese können nur mittels vieler manuell überprüfter Vor- und Nachbedingungen von Änderungsoperationen vermieden werden. Für ein gegebenes Dokument zu entscheiden, ob es sich in einem global gültigen Zustand befindet, ist sehr aufwändig. Dies wird in Renew zum Zeitpunkt dieser Arbeit auch nicht überprüft. Dokumente in einem fehlerhaften Zustand führen einfach zu nicht definiertem Verhalten, wenn mit ihnen weitergearbeitet wird.

Für den Web-Editor ist so ein fragiles Datenmodell nicht einsatztauglich. Besonders im kollaborativen Arbeitsprozess, in dem mehrere Änderungsoperationen möglicherweise miteinander konkurrieren, muss garantiert sein, dass sich ein Dokument immer in einem gültigen Zustand befindet oder einfach in einen solchen überführt werden kann.

Das relationale Datenmodell hat sich in der Entwicklung von Datenbanken bewährt. Es ermöglicht das Formulieren von Invarianten und Operationen basierend auf einem Modell von Mengen und logischen Ausdrücken. Es wird stark von der konkreten Ausführungsreihenfolge einzelner Prozessschritte abstrahiert. Dies ist für Datenbanken als verteiltes System mit nebenläufigen Transaktionen essenziell.

Daher wird auch für die Entwicklung des Web-Editors in dieser Arbeit ein relationales Modell zur Repräsentation von Netzen, Dokumenten und Simulationen gewählt. Unter der Verwendung eines relationalen Datenmodells kann eine existierende Datenbank eingesetzt werden, um das Datenmodell zu implementieren. Der Vorteil dieser Strategie ist, dass ein Großteil aller Konsistenzbedingungen automatisch überprüft und garantiert werden kann.

5.6. Funktionale Programmierung

In dieser Arbeit wird das Paradigma der funktionalen Programmierung verwendet. Besonders in nebenläufigen Prozessen und verteilten Systemen, wie dem in dieser Arbeit entwickelten Editor, ist es besonders wichtig, die Abläufe im System einfach nachvollziehen zu können. Daher wird in dieser Arbeit auf das funktionale Paradigma und Programmiersprachen mit streng funktionaler Semantik gesetzt. Zwar kann auch in anderen Programmiersprachen wie Java in einem funktionalen Stil programmiert werden, allerdings behindert die Semantik einer nicht rein funktionalen Programmiersprache die konsequente und verlässliche Umsetzung.

5.7. Verwendete Werkzeuge

In diesem Abschnitt wird die Auswahl an Werkzeugen, Programmiersprachen und Programmbibliotheken vorgestellt, die auf Basis der gesetzten Anforderungen und Strategien für diese Arbeit verwendet werden. Neben der gewählten Strategie hat auch die persönliche Erfahrung im Umgang mit den jeweiligen Werkzeugen deren Auswahl beeinflusst.

5.7.1. SQLite

Für die Umsetzung des relationalen Datenmodells wird SQLite gewählt. Es ist eine sehr robuste und weitverbreitete Implementation einer relationalen Datenbank. SQLite ist als Bibliothek sehr einfach in eine Anwendung einzubinden und benötigt keinen eigenständigen Datenbankserver. Sollte sich das im Rahmen dieser Arbeit entwickelte Datenmodell bewähren, könnte es auf Basis von SQLite auch in Renew eingesetzt werden.

5.7.2. Elixir

Elixir wird als Programmiersprache für die Entwicklung der Server-Komponente verwendet. Als funktionale Programmiersprache bietet sie sich besonders für die Anwendung des funktionalen Programmierparadigmas. Basierend auf Erlang eignet sich Elixir besonders für die Entwicklung einer robusten Server-Anwendung, die stabil und langlebig laufen soll. Außerdem bietet Elixir Zugang zu dem Phoenix Web Framework.

5.7.3. Phoenix Web Framework

Das Phoenix Web Framework ist eine Sammlung an Programmbibliotheken für die Komposition von Webanwendungen. Phoenix beinhaltet unter anderem den Datenbankadapter Ecto, der eine ausdrucksstarke domänenspezifische Sprache zur Interaktion mit SQL-Datenbanken bereitstellt. Außerdem enthält Phoenix einen HTTP-Router und einen Service zur Verwaltung von WebSocket-Verbindungen. Damit bietet Phoenix genau die Werkzeuge, die nötig sind, um eine interaktive Webanwendung auf Basis von SQLite zu entwickeln und sich dabei auf die Programmierung der Geschäftslogik zu fokussieren.

5.7.4. JavaScript

JavaScript ist die einzige Programmiersprache, mittels derer auf alle Programmierschnittstellen eines Webbrowsers zugegriffen werden kann. Damit stellt sie gezwungenermaßen die einzige Möglichkeit für die Entwicklung der Client-Komponente des Editors dar. Zwar wäre es möglich, Teile des Clients in einer anderen Sprache zu programmieren und dann in JavaScript zu übersetzen, doch im Rahmen dieser Arbeit soll versucht werden, diese Indirektion zu vermeiden, um das Gesamtsystem einfach zu halten.

5.7.5. Svelte Bibliothek

Svelte ist eine von vielen verfügbaren JavaScript-Bibliotheken, die die Schnittstelle zur funktionalen und reaktiven Programmierung von Benutzeroberflächen im Webbrowser anbieten [con25a]. Neben Svelte sind ReactJS, SolidJS und VueJS sehr verbreitete Alternativen [Wal13, Car21, You14]. In dieser Arbeit wird Svelte aus persönlicher Präferenz gewählt.

5.7.6. `partial.lenses` Bibliothek

In den Grundlagen dieser Arbeit wurde bereits das Optik-Entwurfsmuster aus der Funktionalen Programmierung vorgestellt. Die `partial.lenses`-Bibliothek ist eine umfangreiche Sammlung an JavaScript-Funktionen, die das Optik-Entwurfsmuster in JavaScript implementieren [Kar16b].

In dieser Arbeit wird `partial.lenses` in Kombination mit Svelte verwendet, um die Elemente der Benutzeroberfläche in der Client-Komponente zu programmieren und einfach zu kompositionieren. Dafür werden einige Ideen der CalmmJS-Architektur umgesetzt [Kar16a].

Teil III.

Umsetzung des Editor-Servers

6. Import von rnw-Dateien

Um die Anforderung der Kompatibilität mit Renew zu erfüllen, muss der Web-Editor in der Lage sein Renew Dateien einzulesen. In diesem Kapitel wird die dafür zuständige Komponente entwickelt. Dafür wird zuerst das rnw-Dateiformat vorgestellt. Danach wird eine Grammatik in Elixir definiert, um die von Renew serialisierten Java-Objekte samt zyklischer Referenzen einlesen zu können. Die dabei gewonnenen Erkenntnisse werden später als Orientierung für die Entwicklung der weiteren Komponenten verwendet.

6.1. Das rnw-Dateiformat

Um rnw-Dateien einlesen zu können, muss zunächst verstanden werden, wie diese aufgebaut sind. Es handelt sich um Textdateien, die eine Menge an serialisierten Java-Objekten enthalten.

Eine rnwD̄atei enthält zu Beginn eine Versionsnummer. Darauf folgt die serialisierte Wurzel eines Tiefensuchbaumes, der durch das Traversieren eines Java-Objektgraph entsteht. Abschließend kann eine rnwD̄atei noch einige Metainformationen über die Position des Anwendungszustands zum Zeitpunkt des Speichervorgangs enthalten.

Zur Serialisierung in das Textformat einer rnw-Datei wird der Objektgraph in Hauptreihenfolge traversiert. Für jedes Objekt wird der Klassenname gefolgt von jedem einzelnen Attributwert ausgegeben. Als Trennzeichen werden ein oder mehrere Leerzeichen verwendet. Es werden keine gesonderten Trennzeichen verwendet, um Objekte oder Sequenzen von Attributen zu terminieren.

Java-Objekte können sich gegenseitig referenzieren und dabei Referenzzyklen bilden. Zyklische Referenzen bilden Rückkanten beim Durchlaufen des Graphen. Sie werden während der Serialisierung erkannt und mittels gesondertem Schlüsselwort (`REF`) gefolgt von einem Zielindex serialisiert. Der Zielindex bezieht sich dabei auf einen ganzzahligen Zähler, der für jedes serialisierte Objekt erhöht wird. Die Ausgabe `REF 7` beispielsweise steht für eine Referenz auf das Objekt, welches zuvor während des Traversierungsprozesses als siebtes besucht und ausgegeben wurde.

Listing 6.1 zeigt den Inhalt einer einfachen rnw-Datei. In Zeile 1 ist die Versionsnummer (11) des Dateiformats angegeben. In Zeile 2 startet das Wurzelelement des Dokuments (`CPNDrawing`) und gibt an, 3 Kindelemente zu erwarten. Das erste Kindelement (`PlaceFigure`) erstreckt sich von Zeile 3 bis Zeile 9. Das zweite Kindelement (`TransitionFigure`) erstreckt sich von Zeile 10 bis Zeile 16. Das dritte Kindelement (`ArcConnection`) erstreckt sich von Zeile 17 bis Zeile 25. Das Ende der jeweiligen Elemente lässt sich syntaktisch in der Datei nicht erkennen, sondern nur unter Zuhilfenahme der jeweiligen Java-Klassen-Definitionen. In Zeile 23 und Zeile 24 ist die `REF`-Syntax zu erkennen, mittels der sich das `ArcConnection`-Elemente auf die beiden vorherigen `PlaceFigure`- und `TransitionFigure`-Elemente bezieht.

```
11
de.renew.gui.CPNDrawing 3
de.renew.gui.PlaceFigure
"attributes" "attributes" 3
"FillColor" "Color" 112 219 147 255
"FrameColor" "Color" 255 199 158 0
"LineWidth" "Int" 1
20 20 50 50
NULL
de.renew.gui.TransitionFigure
"attributes" "attributes" 3
"FillColor" "Color" 112 219 147 255
"FrameColor" "Color" 255 199 158 0
"LineWidth" "Int" 1
192 21 50 50
NULL
de.renew.gui.ArcConnection
"attributes" "attributes" 1
"Test" "Color" 100 100 200 50
3 69 40 132 26 192 40
NULL
CH.ifa.draw.figures.ArrowTip 0.4 8.0 8.0 1 ""
CH.ifa.draw.figures.ChopEllipseConnector REF 1
CH.ifa.draw.standard.ChopBoxConnector REF 2
NULL
```

Quelltext 6.1: Beispiel einer einfachen rnw-Datei

Für Renew ist dieser Prozess zur Serialisierung für jede Java-Klasse eigenständig in einer jeweiligen Methode implementiert. Einzelne Klassen können also auch von dem hier beschriebenen Format abweichen. Die `AttributeFigure`-Klasse ist ein Beispiel für eine Java-Klasse, die in ihrer Serialisierung von dem hier beschriebenen Prinzip abweicht.

Die Klassendefinitionen in Renew machen starken Gebrauch der objektorientierten Vererbung in Java. Das Datenformat zur Serialisierung eines Objekts hängt also nicht nur von der Definition der zugehörigen Klasse, sondern auch von der gesamten Sequenz aller Elternklassen.

Abhängig von der verwendeten Renew-Version, die zu Beginn einer rnw-Datei vermerkt wird, kann das Format leicht variieren. Der Webeditor soll in der Lage sein, mit all den auftretenden Variationen umgehen zu können.

Jede Klasse, deren Objekte sich als Teil einer rnw-Datei serialisiert abspeichern lassen, implementiert das `Storable`-Interface. Insgesamt wird dieses Interface zum Zeitpunkt dieser Arbeit auf dem `main`-Branch des Renew-Git-Repositoriums von über 200 Klassen implementiert.

Für das Dateiformat von Renew ist also bisher keine explizite Grammatik definiert, auf die sich bezogen werden kann, um rnw-Dateien außerhalb von Renew zu verarbeiten. Stattdessen ergibt sich aus den über 200 Klassen, die das `Storable`-Interface implementieren, implizit die Grammatik des serialisierten rnw-Formats.

Für die Referenznetze, die der Web-Editor bearbeiten können soll, werden nicht alle diese 200 Klassen aus Renew benötigt. Einige der Klassen werden auch für das Zeichnen von endlichen Automaten oder BPMN-Diagrammen verwendet. Allerdings können in einem Renew-Dokumente grundsätzlich beliebige Figurelemente eingefügt und vermischt werden. Somit kann eine rnw-Datei potenziell ein Exemplar jeder dieser 200 Klassen enthalten.

Aufgrund der Präfixkodierung ohne Trennzeichen können beim Einlesen von rnw-Dateien auch keine unbekannten Objekte übersprungen werden. Aus vorangestellten Klassennamen ergibt sich, anhand der Java-Klassendefinition, wie viele darauffolgende Attributwerte zu lesen sind.

6.2. Definition einer Grammatik

Da der Web-Editor nicht in Java entwickelt wird, kann nicht auf die in Renew enthaltenen Java-Klassendefinitionen zurückgegriffen werden. Stattdessen muss eine neue Grammatik definiert werden. Diese muss der durch Renew implizit vorgebenden Struktur entsprechen. Außerdem soll diese Grammatik nicht nur dafür verwendet werden können, um Dateien einzulesen, sondern auch, um Dokumente wieder in eine rnwDatei serialisieren zu können.

Da sich das Renew-Dateiformat zukünftig auch ändern kann, soll auch die in dieser Arbeit definierte Grammatik sich leicht anpassen lassen. Im Gegensatz zu Renew wurde die Grammatik als Liste von Produktionsregeln in einer Datenstruktur definiert. Die Struktur der erarbeiteten Produktionsregeln ist absichtlich direkt an die Vererbungshierarchie der Java-Klassen angelehnt. Listing 6.2 zeigt exemplarisch die Deklaration von vier solcher Regeln.

Die Liste aller nötigen Produktionsregeln der Grammatik wurde manuell aus den über 200 Klassendefinitionen, über verschiedene Renew-Versionen hinweg, herausgearbeitet.

6.3. Auflösung zyklischer Referenzen

In Renew wird aus der rnw-Datei beim Einlesen wieder ein Java-Objektgraph erstellt. Dieser enthält dann auch wieder alle zyklischen Referenzen, da diese anhand der `REF`-Markierungen rekonstruiert werden können.

Der Web-Editor in dieser Arbeit und die in diesem Kapitel entwickelten Grammatik werden aber in Elixir programmiert. In dem funktionalen Programmierparadigma können diese zyklischen Strukturen aber gar nicht direkt repräsentiert werden.

Stattdessen werden die `REF`-Markierungen beim Einlesen gar nicht automatisch aufgelöst, sondern als Referenz stehen gelassen. Das Ergebnis des Einlesevorgangs ist entsprechend auch nicht ein Wurzelknoten, von welchem aus sich die restlichen Objekte transitiv erreichen lassen. Vielmehr wird beim Einlesen eine vollständige Liste aller gelesenen Objekte erzeugt und bereitgestellt. Anhand dieser Liste können einzelne `REF`-Markierungen bei Bedarf manuell aufgelöst werden.

6.4. Test an bestehenden rnw-Dateien

Um sicherzustellen, dass die Grammatik vollständig ist und für alle bestehenden rnw-Dateien verlässlich funktioniert, wurde eine Testsuite aus über 1000 rnw-Dateien zusammengestellt. Diese Dateien wurden aus allen Beispieldateien von verschiedenen Renew-Plugins zusammengestellt und manuell um einige spezifische Beispiele erweitert.

Es wird sowohl das Lesen als auch das Erzeugen von rnw-Dateien getestet. Zusätzlich wird auch das Verhalten im Umgang mit ungültigen und beschädigten Dateien getestet.

6.5. Bereitstellung als eigenständiges Modul

Die Grammatik hat sich in den Tests als sehr robust und einfach erweiterbar erwiesen. Damit stellt sie auch unabhängig vom Web-Editor ein erfolgreiches Arbeitsergebnis dar. Sie wurde im Zuge dieser Arbeit als eigenes Elixir-Modul in der Hex-Paketverwaltung für Elixir und Erlang veröffentlicht.

Zusätzlich zu der Umsetzung in Elixir wurde die Grammatik als Experiment auch in JavaScript implementiert. Auch hier wurde sie als eigenständiges Paket in der JS Paketverwaltung Node Package Manager (NPM) veröffentlicht [Kor25c].

```
%Renewex.Grammar{
  version: 11,
  hierarchy: %{
    "CH.ifa.draw.standard.AbstractFigure" => %{
      # This class has no super class in Java
      super: nil,
      # This class implements two interfaces in Java
      interfaces:
        ["CH.ifa.draw.framework.Figure", "CH.ifa.draw.framework.ParentFigure"]
    },
    "CH.ifa.draw.standard.CompositeFigure" => %{
      # This java class is defined as sub class of another class
      # (`CH.ifa.draw.standard.AbstractFigure`)
      # When reading a serialized object of this class
      # the parser for the parent class is applied first.
      # This mirrors the `super.read()` Java code in the
      # original Renew implementation.
      super: "CH.ifa.draw.standard.AbstractFigure",
      interfaces: ["CH.ifa.draw.framework.Figure"],
      # This class has one field (`figures`) that is serialized.
      # The field is a list of serialized objects
      # (each of type `CH.ifa.draw.framework.Figure`)
      fields:
        [figures: {:list, {:storable, "CH.ifa.draw.framework.Figure"}}]
    },
    "CH.ifa.draw.figures.AttributeFigure" => %{
      super: "CH.ifa.draw.standard.AbstractFigure",
      interfaces: []
    },
    "CH.ifa.draw.figures.RectangleFigure" => %{
      super: "CH.ifa.draw.figures.AttributeFigure",
      interfaces: [],
      # This class containes 4 primitive fields of type integer
      fields: [x: :int, y: :int, w: :int, h: :int]
    },
    # ... 200 more rules ...
}
```

Quelltext 6.2: Ausschnitt aus der Grammatikdefinition zum Lesen von rnw-Dateien

7. Parametrische Grafiksymbole

In diesem Kapitel wird die Repräsentation von grafischen Symbolen, wie Kreisen, Dreiecken oder Kreuzen, im Web-Editor erarbeitet. Dafür wird zunächst untersucht, welche Symbole in Renew zur Verfügung stehen und wie diese in Renew umgesetzt wurden. Danach wird ein Datenformat entwickelt, um die gesamte Menge an Symbolen einheitlich zu beschreiben und in dem Web-Editor zu übertragen.

7.1. Beispiel für Symbole

In Renew lassen sich eine Vielzahl an grafischen Symbolen in einem Dokument platzieren. Transitionen und Plätze eines Petrinetzes werden als Rechtecke und Ellipsen dargestellt. Aber darüber hinaus existieren viele weitere `Figure`-Elemente, die zum Einsatz kommen können.

Abb. 7.1 zeigt einige Beispiele an `Figure`-Element in Renew. Einige von ihnen sind in Renew mit einer semantischen Bedeutung verknüpft, beispielsweise die weiße, doppelt umkreiste Ellipse steht für den Endzustand eines endlichen Automaten. Aber einige sind auch nur dafür gedacht, visuell etwas im Dokument hervorzuheben, beispielsweise ein Dreieck oder ein Kreisbogen. Die einzelnen Symbole lassen sich in Renew entlang horizontaler und vertikaler Richtung skalieren. Die Skalierung wirkt sich dabei auf die verschiedenen Symbole individuell aus, sodass deren markanten Merkmale erhalten bleiben. Beispielsweise die umgeklappte Ecke des Papiersymbols unten links in Abb. 7.1 verläuft stets im 45° Winkel, egal wie weit das Papier horizontal oder vertikal gestreckt oder gestaucht ist. Auch die sechs schwarzen AIP-Symbole rechts unten in Abb. 7.1 lassen sich strecken und stauchen. Auch dabei bleiben die Proportionen der kleinen jeweils mittigen Raute aber unbeeinflusst.

Um eine eingelesene rnw-Datei im Web-Editor grafisch darzustellen, müssen alle zugehörigen Symbole korrekt positioniert ausgegeben werden.

7.2. Figure-Elemente in Java Renew

Jedes in Renew verwendbare Symbol ist in einer zugehörigen Java-Klasse definiert. Es sind die gleichen Java-Klassen, welche auch für die Serialisierung eines Dokuments in eine rnw-Datei verwendet werden.

Das Aussehen eines Symbols ist prozedural in der `draw(Graphics2D)`-Methode der jeweiligen `Figure`-Klasse definiert. Die `draw(Graphics2D)`-Methode erhält als Parameter den Grafikkontext, in den das Symbol gezeichnet werden soll. Quellcode 7.1 zeigt ein vereinfachtes Beispiel einer `draw`-Methode aus der Renew `EllipseFigure` Klasse.

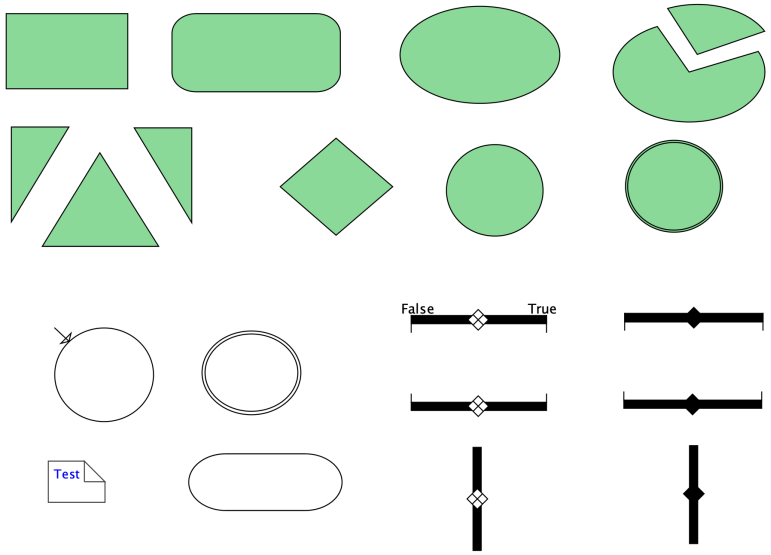


Abbildung 7.1.: Auf der Zeichenfläche von Renew platzierte Symbole

```
1  import java.awt.Graphics2D;
2  public class EllipseFigure extends AttributeFigure {
3      private Rectangle fDisplayBox;
4      // ...
5      public void draw(Graphics g) {
6          Shape s = new Ellipse2D.Double(r.x, r.y, r.width, r.height);
7          g.setColor(...);
8          g.fill(s);
9          g.setColor(...);
10         g.draw(s);
11     }
12     // ...
13 }
```

Quelltext 7.1: Die vereinfachte `draw`-Method der `EllipseFigure`-Klasse in Renew

Das Rechteck `fDisplayBox` in Zeile 3 gibt an, in welcher Größe und an welcher Position ein Symbol gezeichnet werden soll. Für jeden neuen Symboltyp, der in Renew hinzugefügt wird, muss eine neue solche Klasse mit passender `draw(Graphics2D)`-Methode definiert werden. Um diese Klasse muss auch das Serialisierungsformat erweitert werden.

Für die Darstellung von rnw-Dateien im Webeditor dieser Arbeit muss also jedem der 200 einlesbaren Figurelemente eine Darstellungsform zugeordnet werden, die der Darstellung in Renew entspricht. Da die Darstellung in Renew als Java-Methode kodiert ist, können die Symbole nicht automatisch übernommen werden. Stattdessen müssen passende Symbole von Grund auf neu definiert werden.

Listing 7.2 zeigt eine weitere `draw(Graphics2D)`-Method aus Renew. Diese ist etwas umfangreicher als die aus Listing 7.1. Mittels einer gesonderten `outline`-Methode wird die rautenförmige Silhouette erzeugt, die in Abb. 7.1 unten rechts zu sehen ist.

```
1  public class SplitDecoration {
2      public Shape outline(Rectangle r) {
3          GeneralPath p = new GeneralPath();
4
5          p.moveTo(r.x, r.y + r.height / 2);
6          p.lineTo(r.x + r.width / 2, r.y);
7          p.lineTo(r.x + r.width, r.y + r.height / 2);
8          p.lineTo(r.x + r.width / 2, r.y + r.height);
9          p.closePath();
10         return p;
11     }
12
13     public void draw(Graphics g) {
14         Shape shape = outline(
15             new Rectangle(x - _halfSize, y - _halfSize, _size, _size)
16         );
17
18         g2.setColor(lineColor);
19         g2.draw(shape);
20     }
21 }
```

Quelltext 7.2: Die vereinfachten `draw`-Method der `SplitDecoration` Klasse in Renew

7.3. Vereinheitlichung bestehender Symbole

Zunächst wurde die Menge und Variation der bestehenden Symbole in Renew ermittelt. Anstatt, wie in Renew, die Symbole erneut prozedural direkt in den Programmcode des Web-Editors zu kodieren, soll versucht werden, die Symbole in einem einheitlichen Format deklarativ zu definieren. Dieses Format soll dann unabhängig von einer bestimmten Programmiersprache verarbeitet und dargestellt werden können. Ein typisches Format zur Speicherung von geometrischen Formen ist das SVG-Format. Als Vektorformat ist es von der Auflösung des Ausgabemediums unabhängig.

Bei der Untersuchung bestehender Renew-Symbole ergibt sich aber, dass einige dieser Symbole nicht vollständig als SVG-Grafik repräsentiert werden können. In einigen Figur-Elementen wird von der Möglichkeit Gebrauch gemacht, während des Zeichenvorgangs geometrische Berechnungen in Java durchzuführen, um gewünschte Abstände und Proportionen in dem jeweiligen Symbol zu berechnen. Die dynamische Ausdrucksstärke in der in Java basierten Symboldefinition lässt sich nicht in eine statische SVG-Grafik übertragen. Daher kommt das SVG-Format zur Umsetzung der Symbole nicht in Frage.

Es lässt sich aber eine Gemeinsamkeit aller in Renew-Figuren vorkommenden Berechnungen feststellen. Alle diese Berechnungen werden durchgeführt, um geometrische Formen mit festen Seitenverhältnissen in einen Zielbereich mit variablen Seitenverhältnissen zu skalieren. Die dafür nötigen Rechenterme lassen sich über alle Symbole hinweg vereinheitlichen.

7.4. Domänenspezifische Sprache zur Definition von Symbolen

Aufbauend auf allen in Renew-Figuren durchgeführten Berechnungen lässt sich ein neues Datenformat zur Beschreibung von geometrischen Formen definieren. Dieses Format basiert auf den Vektor-Pfad-Befehlen in SVG und erweitert diese um einige arithmetische Operationen.

Das SVG-Format definiert Vektorpfadbefehle zum Zeichnen von Linien, Kreisbögen und Bézierkurven. Abb. 7.2 zeigt die Grammatik eines SVG-Pfadbefehls. Die Koordinaten eines Pfades müssen jeweils als konstante Gleitkommazahlen angegeben werden. Zwar lässt sich eine Grafik nachträglich linear transformieren, aber die einzelnen Koordinaten können nicht individuell in Abhängigkeit zur Ausgabegröße definiert werden.

Das für den Web-Editor entwickelte Vektorgrafik-Format baut auf den von SVG definierten Pfadbefehlen auf, erlaubt aber als Koordinatenparameter neben konstanten Werten auch arithmetische Ausdrücke. Die Erweiterung der Grammatik ist in Abb. 7.3 dargestellt. Die Produktionsregel ⟨num⟩ wurde ersetzt. Die ⟨unit⟩ Regel erlaubt die Verwendung einiger freier Variablen anstelle von konstanten Zahlenwerten. Dadurch hat ein Ausdruck in diesem Format keinen festen Wert, sondern ist nur ein Schema, welches durch die Belegung der Variablen `width` und `height` zu einem Pfad ausgewertet werden kann.

Dieses Format erhöht die Ausdrucksstärke gegenüber dem SVG-Pfadformat. Die erhöhte Ausdrucksstärke erlaubt die Beschreibung aller in Renew darstellbaren Grafiksymbole in einem einheitlichen Datenformat. Dieses Format kann von einer beliebigen Programmiersprache interpretiert werden, um für ein gegebenes Rechteck einen proportional passend skalierten SVG-Pfad zu erzeugen.

7.5. Definition der einzelnen Symbole

Um die Funktionstüchtigkeit zu demonstrieren, wurden alle in Renew einprogrammierten Symbole in das vorgeschlagene deklarative Datenformats übersetzt. Dadurch ist der Web-Editor in der Lage, jedem aus einer rnw-Datei gelesenen Element ein zugehöriges Symbol zuzuweisen und dieses auch darzustellen.

Abb. 7.4 zeigt die Darstellung eines parametrisch definierten Symbols in drei unterschiedlichen proportionalen Skalierungen. Zu sehen ist, dass sich einige Maße des Symbols an das Seitenverhältnis der rechteckigen Begrenzung (grün, gestrichelt) anpassen. Andere Teile des Symbols behalten ihre Proportionen unabhängig von dem Seitenverhältnis der Begrenzung bei. Der schwarze Balken im Hintergrund streckt sich horizontal vollständig bis an beide Enden. In der Vertikalen hat der schwarze Balken aber eine feste Größe und bleibt von der Skalierung unbeeinflusst. Der rautenförmige Ausschnitt in der Mitte ist gänzlich unbeeinflusst von jeglicher Skalierung. In Listing 7.3 ist die zugehörige Definition des Pfadschemas dargestellt. In den ersten fünf Zeilen wird das schwarze Rechteck in vier Schritten gezeichnet. In den nächsten fünf Zeilen wird die Raute gezeichnet.

⟨path⟩ ⊨ ⟨segment⟩ | ⟨segment⟩ ⟨path⟩
⟨segment⟩ ⊨ ⟨move⟩ ⟨step⟩
⟨step⟩ ⊨ ⟨command⟩ | ⟨command⟩ ⟨step⟩
⟨command⟩ ⊨ ⟨line⟩ | ⟨arc⟩ | ⟨quadratic⟩ | ⟨cubic⟩ | ⟨close⟩
⟨move⟩ ⊨ ⟨move relative⟩ | ⟨move absolute⟩
⟨move relative⟩ ⊨ `m` ⟨num⟩ ⟨num⟩
⟨move absolute⟩ ⊨ `M` ⟨num⟩ ⟨num⟩
⟨line⟩ ⊨ ⟨line relative⟩ | ⟨line absolute⟩
⟨line relative⟩ ⊨ `l` ⟨num⟩ ⟨num⟩
⟨line absolute⟩ ⊨ `L` ⟨num⟩ ⟨num⟩
⟨arc⟩ ⊨ ⟨arc relative⟩ | ⟨arc absolute⟩
⟨arc relative⟩ ⊨ `a` ⟨num⟩ ⟨num⟩ ⟨bool⟩ ⟨bool⟩ ⟨bool⟩ ⟨num⟩ ⟨num⟩
⟨arc absolute⟩ ⊨ `A` ⟨num⟩ ⟨num⟩ ⟨bool⟩ ⟨bool⟩ ⟨bool⟩ ⟨num⟩ ⟨num⟩
⟨quadratic⟩ ⊨ ⟨quadratic relative⟩ | ⟨quadratic absolute⟩
⟨quadratic relative⟩ ⊨ `q` ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩
⟨quadratic absolute⟩ ⊨ `Q` ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩
⟨cubic⟩ ⊨ ⟨cubic relative⟩ | ⟨cubic absolute⟩
⟨cubic relative⟩ ⊨ `c` ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩
⟨cubic absolute⟩ ⊨ `C` ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩ ⟨num⟩
⟨close⟩ ⊨ `z` | `Z`
⟨num⟩ ⊨ ⟨literal⟩
⟨literal⟩ ⊨ *Numeric Literal*
⟨boolean⟩ ⊨ `1` | `0`

Abbildung 7.2.: Klassische Grammatik zur Beschreibung von Vektorpfaden SVG

⟨num⟩ ⊨ ⟨expression⟩ | ⟨literal⟩
⟨expression⟩ ⊨ ⟨dim⟩ + ⟨func⟩ (⟨literal⟩ , ⟨dim⟩)
⟨dim⟩ ⊨ ⟨literal⟩ · ⟨unit⟩
⟨unit⟩ ⊨ `width` | `height` | `maxsize` | `minsize`
⟨func⟩ ⊨ `min` | `max` | `sum`
⟨literal⟩ ⊨ *Numeric Literal*

Abbildung 7.3.: Erweiterung der SVG-Pfad-Grammatik um einfache arithmetische Ausdrücke

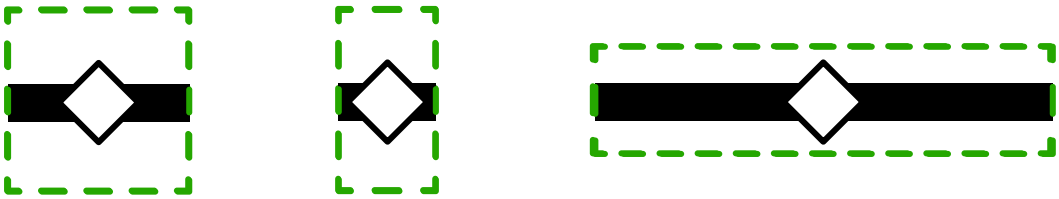


Abbildung 7.4.: Beispiel der Darstellung eines parametrisch definierten Symbols in verschiedenen Skalierungen

```
M 0.0 width  + min(0, 0 * minsize) 0.5 height + max(-10, -0.5 * height)
H 1.0 width  + min(0, 0 * minsize)
v 0.0 height + min(20, 1 * height)
H 0.0 width  + min(0, 0 * minsize)
Z
M 0.5 width + sum(  0, 0 * minsize) 0.5 height  + sum(-20, 0 * minsize)
l 0.0 width + sum( 20, 0 * minsize) 0   height  + sum( 20, 0 * minsize)
l 0.0 width + sum(-20, 0 * minsize) 0   height  + sum( 20, 0 * minsize)
l 0.0 width + sum(-20, 0 * minsize) 0   height  + sum(-20, 0 * minsize)
Z
```

Quelltext 7.3: Beispiel eine parametrischen Symboldefinition

7.6. Bereitstellung als eigenständiges Modul

Im Rahmen dieser Arbeit wurde eine Prozedur zur Auswertung des erweiterten Vektorpfadformats sowohl in JavaScript als auch in Elixir implementiert. Das Ergebnis wurde jeweils als eigenständiges Modul in dem NPM und Hex-Paketregister veröffentlicht [Kor25d].

8. Relationales Datenschema

Mittels der Ergebnisse aus den beiden vorherigen Kapiteln können rnw-Dateien eingelesen und prinzipiell auch grafisch dargestellt werden. In diesem Kapitel wird eine Datenstruktur erarbeitet, die sich auch dafür eignet, Änderungen an Dokumenten vorzunehmen. Zunächst werden einige Schwachstellen des in Renew verwendeten objektorientierten Modells aufgezeigt, um dadurch die Verwendung eines relationalen Datenmodells zu motivieren. Anschließend wird die große Menge an Datentypen aus Renew in eine kleinere Menge von Entitätstypen normalisiert. Auch die Beschreibung der grafischen Darstellung der verschiedenen Figurelemente wird in das relationale Modell übertragen.

8.1. Probleme des objektorientierten Modells

In Kapitel 5 „Strategie zur Umsetzung“ wurde bereits die Verwendung eines relationalen Datenmodells zur Repräsentation von Dokumenten motiviert. Im Folgenden wird das in Renew verwendete objektorientierte Modell untersucht, um einige Schwachstellen aufzudecken und den Einsatz des relationalen Modells für den Web-Editor ausführlicher zu motivieren.

8.1.1. Renew-Dokumente als gerichteter Graph

In Renew stellt ein `Drawing` Objekt den Wurzelknoten des Objekt-Graphen dar, der ein zu bearbeitendes Dokument repräsentiert. Die restlichen Knoten im Graphen sind die Elemente, aus denen sich der Inhalt des Dokuments zusammensetzt. Im Fall eines Petrinetzes also die Plätze, Transitionen und Textanschriften, aus denen das Netz besteht.

Aus Sicht dieser Wurzel kann der Graph zwar vereinfacht als Baumstruktur betrachtet werden, allerdings werden Elemente nicht eindeutig von nur einem Elternknoten referenziert. Außerdem können Knoten im Objektgraph sich auch gegenseitig referenzieren und Referenzzyklen bilden. Daher ist ein Dokument in Renew im Allgemeinen nur als zyklischer gerichteter Graph zu betrachten.

Die Struktur eines Petrinetzes ist zwar selbst ein gerichteter zyklischer Graph, allerdings ist zwischen der zu modellierenden Topologie und der Topologie der dafür verwendeten Datenstruktur zu unterscheiden. Ein gerichteter Graph muss nicht als Graph aus Objektreferenzen repräsentiert werden. Ein Beispiel für eine kompakte Repräsentation von Graphen ist eine Adjazenzmatrix.

Die Repräsentation eines Dokuments mittels gerichtetem Graph aus Objektreferenzen bringt einige Herausforderungen mit sich. In Renew sind einige dieser Herausforderungen nicht vollständig gelöst und lassen sich in unerwartetem Verhalten in der Benutzung wiedererkennen.

Ein Dokument kann Kanten enthalten, deren referenzierten Start- und Zielknoten nicht selbst als Elemente im Dokument enthalten sind. Ein Element kann mehrfach als Kindknoten im Dokument referenziert werden. Die resultierende Reihenfolge der Elemente hängt dann von der gewählten Traversionsstrategie ab. Die gesamte Menge aller Elemente eines Dokuments kann nur mittels linearer Suche ermittelt werden. Globale Eigenschaften des Dokuments können nur aufwändig geprüft und aufrechterhalten werden.

In Renew kann daraus resultierendes Fehlverhalten in verschiedenen Situationen beobachtet werden. Es kommt vor, dass sich ein Element per Mausklick nicht auswählen lässt, weil es während der Breiten- oder Tiefensuche nicht erreicht wird, obwohl es im Dokument vorhanden ist. Es kann auch vorkommen, dass Elemente, die per Mausbewegung auf der Zeichenfläche verschoben werden, sich schneller bewegen als der Mauszeiger, weil durch Zyklen im Referenzgraphen Rückkopplungen entstehen.

In Programmiersprachen ohne automatische Speicherverwaltung können unbeabsichtigte Zyklen oder Mehrfachkanten in Referenzgraphen zu Systemabstürzen führen. Dieses Thema wird in der Literatur auch unter dem Konzept der Eigentumssemantik diskutiert. Eine typische Lösung ist das strikte Erzwingen einer Baumtopologie im Referenzgraphen, um Objekte in eine eindeutige Eltern-Kind- beziehungsweise Eigentümer-Eigentum-Relation zueinander zu setzen.

8.1.2. Relationales Modell als Abstraktion

Für die Modellierung von Petrinetzen im Web-Editor die Verwendung eines relationalen Datenmodells Abhilfe schaffen. In der relationalen Modellierung wird von der konkreten Repräsentation von Daten im physikalischen Speichermedium des Computers abstrahiert. In diesem Modell werden alle Strukturen als Relationen, also als Mengen von Tupeln, betrachtet. Die Mengen können mittels relationalen und Mengenoperationen bearbeitet werden. Wie genau diese Mengen im Speicher des Computers abgebildet und persistiert werden, spielt auf dieser Abstraktionsebene keine Rolle. Wenn die Struktur von Renew-Dokumenten sich in Form eines relationalen Datenmodells ausdrücken lässt, können die zuvor genannten Herausforderungen des objektorientierten Modells umgangen werden.

Der entscheidende Unterschied zwischen der in Renew verwendeten Modellierung und des für den Web-Editor in dieser Arbeit vorgeschlagenen Modells liegt in der Entkopplung der verwendeten Datenstruktur von der Struktur der zu repräsentierenden Daten. Sowohl in Renew als auch im Web-Editor sollen interaktiv bearbeitbare Petrinetze modelliert werden.

In Renew entspricht die aus Java-Objekten bestehende Datenstruktur genau der Topologie des jeweils dargestellten Petrinetzgraphen. Ein fehlerhaftes Petrinetz wird durch eine fehlerhafte Datenstruktur repräsentiert. Auf Basis einer fehlerhaften Datenstruktur können aber keine aussagekräftigen weiteren Berechnungen durchgeführt werden.

Im Webeditor werden die beiden Strukturen entkoppelt. Im relationalen Modell wird ein fehlerhaftes Petrinetz von einer fehlerfreien und konsistenten Datenstruktur reprä-

sentiert. Änderungsoperationen, Datenabfragen und Konsistenzprüfungen lassen sich auf Basis eines relationalen Modells sehr prägnant und robust ausdrücken. Um eine Liste aller Transitionen eines Petrinetzes zu ermitteln, muss nicht prozedural das gesamte Netz durchlaufen werden. Stattdessen kann direkt auf eine vom Modell definierte Menge zugegriffen werden.

8.1.3. Gegenüberstellung von objektorientierten und relationalen Modell

Um den Unterschied zwischen dem in Renew verwendeten objektorientierten Modell und dem in dieser Arbeit verwendeten relationalen Modell zu verdeutlichen, wird ein konkretes Beispiel diskutiert.

Eine typische Operation beim Bearbeiten von Petrinetzen ist der Zugriff auf die Menge aller Kanten, die von Knoten des Typs Transition ausgehen. In der von Renew verwendeten objektorientierten Modellierung ergibt sich diese Menge nur implizit aus mehreren Implementationsdetails. Zunächst muss die Menge aller Knoten per Tiefensuche aus dem Dokument ermittelt werden. Ob Knoten dabei mehrfach auftreten können und wie damit umgegangen wird, ergibt sich aus der Definition der Objektgleichheit der jeweiligen Knotenobjekte. Als nächstes muss die Menge *aller* Kanten per Tiefensuche durch das Dokument ermittelt werden. Wie mit mehrfach vorkommenden Kanten umgegangen wird, ergibt sich jeweils wieder aus der konkreten Implementierung. Abschließend müssen die Mengen der Knoten und Kanten miteinander verknüpft werden, um die Teilmenge der gesuchten Kanten zu ermitteln. Auch hierfür ist ein Freiraum zur Umsetzung gegeben, der auch Raum für Programmierfehler bietet.

Im Relationalen Modell hingegen lässt sich eine solche typische Anfrage deklarativ sehr knapp und bündig in relationaler Algebra ausdrücken [Elm02]:

$$\text{Edges} \bowtie_{\text{source}=\textit{node\_id}} \left(\sigma_{\text{type}=\text{'transition'}}(\text{Nodes})\right))$$

Natürlichsprachlich ausgedrückt entspricht das der Verknüpfung aller Elemente der Menge *Nodes*, für die gilt, dass sie vom Typ *transition* sind, mit allen Elementen der Menge *Edges*, unter der Bedingung, dass das *source* Attribut des Elements aus *Edge* mit dem *node_id* Attribut des jeweiligen Elements aus *Node* übereinstimmt.

Die Bedeutung eines so bündigen deklarativen Ausdrucks ist einfacher nachzuvollziehen als die Auswirkung einer Prozedur aus mehreren Schleifen und Abbruchbedingungen. Dadurch ist ein korrektes Verhalten des Gesamtsystems einfacher zu erreichen und zu verifizieren.

8.1.4. SQLite als Implementation des Relationalen Modells

Letztendlich müssen die Datenstrukturen und Operationen eines relationalen Modells natürlich trotzdem auf einem Computer implementiert werden. Hierfür kann aber ein bestehendes relationales Datenbanksystem verwendet werden. Relationale Datenbank-

systeme sind darauf ausgelegt, die Konsistenz von großen Datenmodellen zu garantieren und nebenläufige Änderungsoperationen auf ihnen auszuführen. Damit erfüllen sie genau die Anforderungen des in dieser Arbeit entwickelten Web-Editors.

Die meisten relationalen Datenbanksysteme sind als Client-Server-Anwendung strukturiert. Der Datenbankserver wird unabhängig von einer Client-Anwendung ausgeführt und muss auch eigenständig gewartet werden.

SQLite hingegen ist ein relationales Datenbanksystem, welches sich als Programmbibliothek vollständig in eine Anwendung einbinden lässt. In diesem Kontext lässt sich SQLite als Werkzeug zur Definition und Manipulation von relationalen Datenstrukturen verstehen. Mit SQL als DSL lassen sich komplexe relationale Operationen und Abfragen präzise ausdrücken.

Im Rahmen dieser Arbeit wird SQLite in Kombination mit der Elixir-Datenbankschnittstelle Ecto verwendet. Somit können alle Datenstrukturen zum Speichern von Dokumenten und den enthaltenen Petrinetzen mittels relationaler Semantik in Elixir definiert werden. Nebenläufige Änderungsoperationen werden von SQLite koordiniert und müssen nicht manuell implementiert werden.

Das relationale Modell ist nicht an die Implementierung in Elixir gekoppelt, sondern kann ohne große Anpassungen in anderen Programmierumgebungen übernommen werden. Da SQL eine standardisierte Sprache für die Definition von relationalen Modellen ist, kann auch der Einsatz von SQLite durch eine alternative relationale Datenbank wie PostgreSQL oder MySQL ersetzt werden.

8.2. Eindeutige Identifizierung von Entitäten

Um das kollaborative Arbeiten in einem Dokument zu ermöglichen, müssen Änderungsoperationen eindeutig beschrieben und kommuniziert werden können. In Renew können Objekte direkt über eine Referenz auf ihre lokale Speicheradresse manipuliert werden. Im Web-Editor ist dies nicht möglich. Zum einen, weil die konkreten Speicheradressen hinter der Abstraktion des relationalen Modells verborgen bleiben. Zum anderen aber auch, weil eine lokale Speicheradresse in einem verteilten System gar keine sinnvolle Bedeutung hat.

Stattdessen muss eine neue Identifikationsmöglichkeit für bearbeitbare Elemente geschaffen werden. In der relationalen Modellierung wird ein Attribut, welches ein Element eindeutig identifiziert, Primärschlüssel genannt. Jedes Dokument und jedes darin enthaltene Element muss also per Definition mit einem Primärschlüssel versehen werden.

8.2.1. UUID

UUIDs sind eine Möglichkeit zur Generierung von synthetischen eindeutigen Primärschlüsseln. Sie können nebenläufig und unabhängig von den identifizierenden Elementen generiert werden. Damit eignen sie sich besonders zum Einsatz in verteilten Systemen.

Wie schon in Kapitel 2 „Grundlagen“ erwähnt, hat auch Lukas Voß in seiner Masterarbeit die Verwendung von UUID zur Identifikation von Objekten in Renew vorgeschlagen.

In dieser Arbeit werden UUIDs als Primärschlüssel für alle Entitäten verwendet. Anhand des Primärschlüssels lässt sich für zwei gegebene Entitäten auch eindeutig entscheiden, ob es sich um dieselbe oder um zwei unterschiedliche Entitäten handelt.

8.3. Normalisierung

Um die aus Renew eingelesenen Daten im Web-Editor bearbeiten zu können, müssen sie in ein passendes relationales Modell übersetzt werden. Das relationale Modell muss also so gestaltet werden, dass eine passende bidirektionale Abbildung zwischen Renews Datenmodell und dem des Web-Editors möglich ist.

In Kapitel 6 „Import von rnw-Dateien“ wurde schon ermittelt, dass das objektorientierte Modell, was in Renew die Struktur von Dokumenteninhalten vorgibt, aus über 200 Java-Klassen besteht. Diese 200 Klassendefinitionen in den Web-Editor zu übertragen, wäre im Rahmen dieser Arbeit zu aufwändig.

Stattdessen soll die Gelegenheit genutzt werden, diese Komplexität durch eine Normalisierung des Datenmodells zu reduzieren. Das Ziel der Normalisierung ist das Entfernen von Redundanzen in einem Datenschema.

Viele dieser Klassen in Renew unterscheiden sich nur schwach. Teilweise ist der einzige Unterschied die Definition der grafischen Darstellung. Die verschiedenen Klassen lassen sich in vier Gruppen einteilen, die sich natürlicherweise daraus ergeben, dass alle in Renew erstellten Dokumente Zeichnungen von gerichteten Graphen enthalten.

8.3.1. Renew Kästen

Kästen sind Elemente, die in einer bestimmten Größe an einer bestimmten Position auf der Zeichenfläche platziert und dargestellt werden können. Ein Beispiel hierfür sind Knoten eines Graphen, wie etwa Plätze oder Transitionen eines Netzes. Abb. 8.1 zeigt, wie verschiedene Figurelemente in Renew mittels eines rechteckigen Rahmens positioniert werden. Die Eckpunkte der Positionierung sind als weiße Kästchen zu erkennen.

Die Art der Darstellung dieser Elemente kann durch zusätzliche Attribute gesteuert werden. Beispielsweise kann die Füllfarbe, die Strichstärke oder die Deckkraft, mit der ein Element dargestellt wird, verändert werden.

Elemente können sich auch in ihrer geometrischen Form unterscheiden. Statt als Rechteck können Elemente auch in Form von Ellipsen, Dreiecken oder aufwändig gestalteten Symbolen dargestellt werden. Die Position und Größe eines Elements wird aber in jedem Fall von einem rechteckigen Rahmen vorgegeben.

Die Java-Klasse `CH.ifa.draw.figures.RectangleFigure` dient in Renew als Elternklasse für die Definition aller anderen kastenförmigen Figuren.

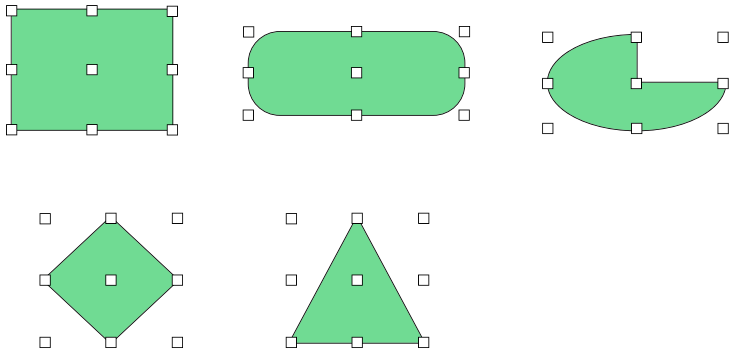


Abbildung 8.1.: Figuren in Renew mit den Eckpunkten der jeweils rechteckigen Umrahmung

8.3.2. Renew Schriftzüge

In Renew kann Freitext in ein Dokument eingefügt werden. Dieser muss an einer bestimmten Position auf der Zeichenfläche platziert werden. Mittels zusätzlicher Attribute kann das Aussehen wie Schriftfarbe, Schriftart und Schriftgröße gesteuert werden.

Außerdem kann ein Textelement auch mit einer Hintergrundfarbe und einer rechteckigen Umrandung versehen werden. Die Größe des Rechtecks ergibt sich automatisch aus dem Textinhalt und kann nicht manuell bestimmt werden. Abb. 8.2 zeigt verschiedene formatierte Text-Elemente in Renew.

8.3.3. Renew Polygonzüge

Polygonzüge sind Elemente, die durch eine Liste von (x, y)-Koordinaten beschrieben werden. Sie werden grafisch als Linienzug auf der Zeichenfläche dargestellt. Mittels zusätzlicher Attribute kann die Darstellungsform eines Linienzugs bestimmt werden. Beispielsweise kann die Strichstärke verändert oder zwischen linear und quadratisch interpolierter Darstellung gewählt werden. Abb. 8.3 zeigt Polygonzüge in Renew in verschiedenen Darstellungsformen.

Die Java-Klasse `CH.ifa.draw.contrib.PolygonFigure` dient in Renew als Elternklasse für die Definition aller anderen Figuren, die Linien oder Pfeile repräsentieren.

8.3.4. Renew Pfeilspitzen

Verschiedene Formen von Pfeilspitzen sind in Renew jeweils als eigene Java-Klasse definiert und können als Dekoration an den Anfang und das Ende von Polygonzügen angefügt werden. Abb. 8.4 zeigt verschiedene Pfeilspitzen an den Endpunkten von Polygonzügen in Renew. Durch Plugins kann Renew auch um weitere Pfeilspitzen erweitert werden.

In Renew implementieren alle Pfeilspitzen das `CH.ifa.draw.figures.LineDecoration`-Interface.

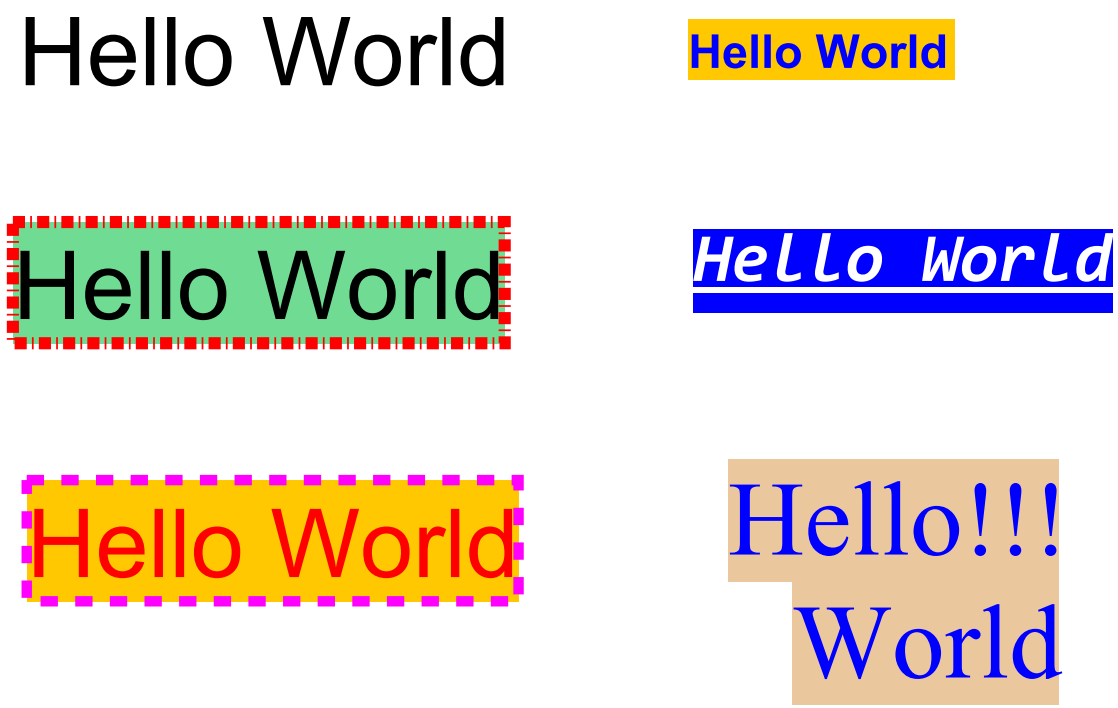


Abbildung 8.2.: Formatierter Text in RENEW in unterschiedlichen Farben, Größen und Schriftarten

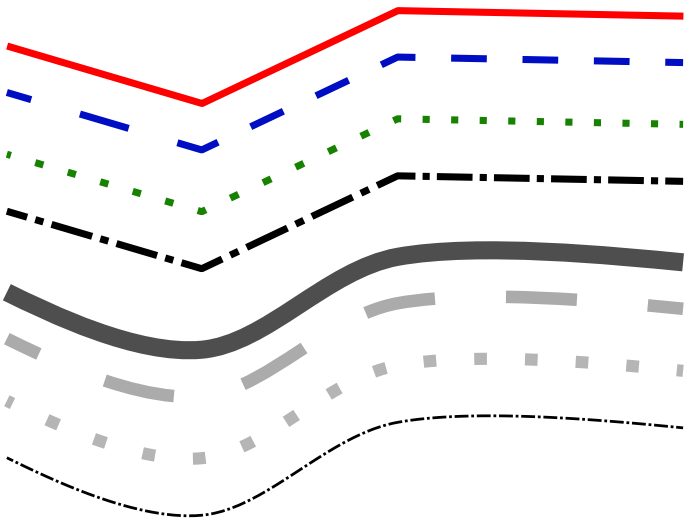


Abbildung 8.3.: Verschiedene Strichstärken, Farben und Interpolationen von Polygonzüge in RENEW

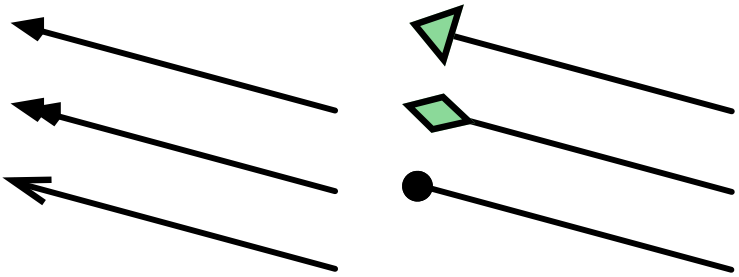


Abbildung 8.4.: Verschiedene Pfeilspitzen am Ende von Polygonzügen in Renew

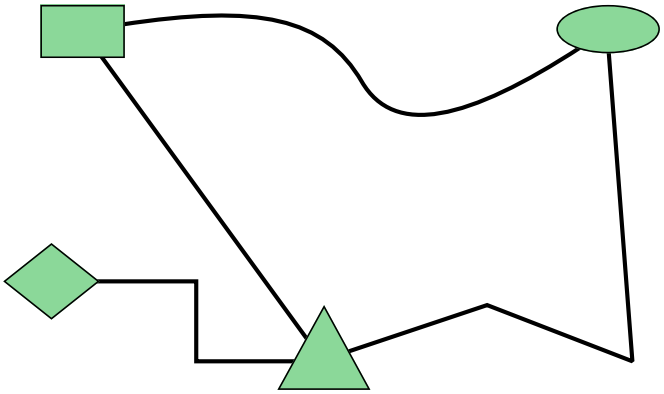


Abbildung 8.5.: Durch Polygonzüge verbundene Figurelemente in Renew

8.3.5. Renew Konnektoren

Konnektoren beschreiben eine vorgegebene geometrische Beziehung zwischen anderen Objekten in einem Dokument. Beispielsweise werden Konnektoren verwendet, um den Endpunkt eines Polygonzugs mit dem Mittelpunkt einer Ellipse zu verbinden. Durch diese Verbindung wird dann ausgedrückt, dass, wann immer die Position der Ellipse verändert wird, auch die Position des Endpunktes angepasst werden muss. Abb. 8.5 zeigt verschiedene Kombinationen aus Elementen, die durch Linienzüge miteinander verbunden sind. Abb. 8.6 zeigt die gleichen Verbindungslinien mit Pfeilspitzen. Daran ist zu erkennen, dass die Linienendpunkte nicht bis zur Mitte der Figurelemente ragen und durch diese verdeckt werden, sondern tatsächlich auf der Kontur des jeweiligen Elements platziert sind.

Renew erhält die durch Konnektoren beschriebenen Beziehungen eigenständig aufrecht. Die Konnektoren werden aber nur benutzt, um die Positionen von Elementen infolge einer durchgeführten Änderung zu positionieren. Durch Konnektoren verbundene Elemente speichern ihre jeweils aktuelle Position. Dadurch ist es möglich, Elemente korrekt positioniert darzustellen, ohne die von den Konnektoren beschriebene Positionsberechnung durchzuführen.

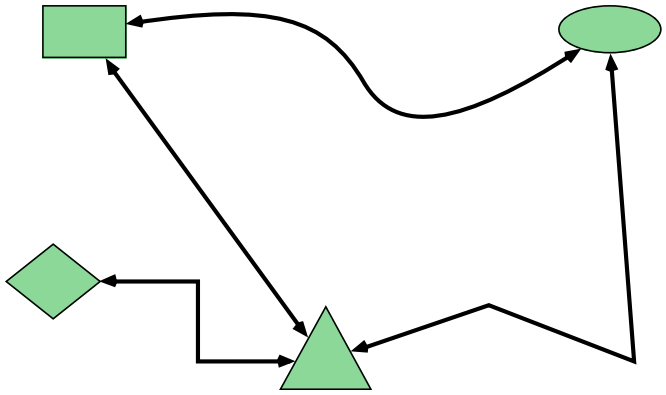


Abbildung 8.6.: Durch Polygonzüge mit Pfeilspitzen verbundene Figurelemente in Renew

Konnektoren in Renew implementieren das `CH.ifa.draw.framework.Connector`-Interface. Renew bietet verschiedene Arten von Konnektoren an, um unterschiedlich komplexe geometrische Beziehungen auszudrücken.

8.4. Das relationale Schema

Für die Definition des relationalen Datenmodells des Web-Editors werden nun die zu Renew passenden Relationen beschrieben. Die über 200 Java-Klassen können so auf wenige Entitätstypen reduziert werden.

8.4.1. Figurtypen

Zunächst können aus der vorangegangenen Analyse die Entitätstypen namens Box (Kasten), Edge (Kante) und Text (Text) abgeleitet werden. Abb. 8.7 stellt diese in einem Entity-Relationship-Diagramm dar. Hierfür wird die Chen-Notation verwendet [Che76]. Kästen bestehen aus einer (x, y)-Position und einer Breite und Höhe. Kanten besitzen einen Start- und einen Endpunkt und beliebig viele dazwischen liegende Wegpunkte. Wegpunkte wiederum besitzen jeweils selbst eine (x, y)-Position und sind in einer festen Reihenfolge angeordnet. Texte bestehen aus einer (x, y)-Position und einem Textinhalt. Alle Entitäten sind über eine eindeutige UUID identifizierbar.

8.4.2. Reihenfolge im Dokument

Die festgelegte Reihenfolge von Elementen im Dokument hat eine Auswirkung auf die Reihenfolge, in der sie dargestellt werden, und damit darauf, wie sie sich grafisch überlappen und verdecken können. In anderen Grafikprogrammen wie Adobe Photoshop oder Illustrator wird als Metapher für eine geordnete Liste von Elementen in einem Dokument der Begriff der *Ebene*, auf englisch *Layer*, verwendet.

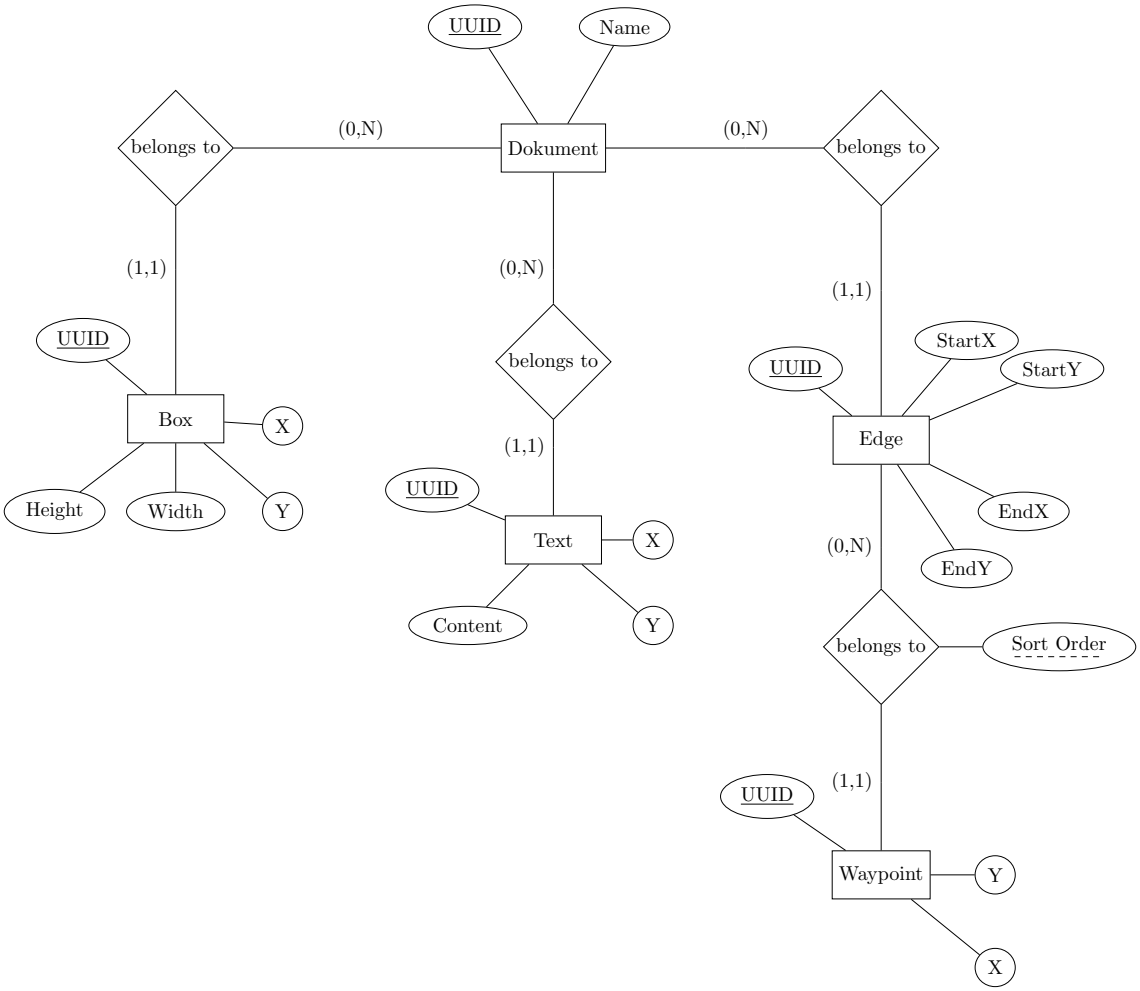


Abbildung 8.7.: Entity-Relationship (ER)-Diagramm zur Beschreibung der vereinfachten Entitätstypen von Renew-Dokumenten

Diese Reihenfolge, in der Elemente dargestellt werden sollen, muss explizit im Datenmodell gespeichert werden. Abb. 8.8 zeigt das entsprechend erweiterte Entity-Relationship (ER)-Diagramm. Das Datenmodell wurde um einen Entitätstyp namens `Layer` erweitert, der die verschiedenen Elemente zusammenfasst. Ein Layer erhält ein Attribut `Z-Index` um die Sortierung in Richtung Z-Achse zu erlauben. Außerdem wird für den Entitätstypen Layer das Attribut `Semantic Tag` definiert. Darüber kann jedem Layer der Name einer Renew-Java-Klasse zugeordnet werden. Das ist nötig, um verschiedene Arten von Knoten und Kanten aus Renew verlustfrei importieren zu können.

8.4.3. Hierarchie im Dokument

In Kapitel 4 „Anforderungen“ wurde bereits als Anforderung aufgestellt, dass auch die hierarchische Beziehung zwischen Elementen im Web-Editor modelliert werden muss. Die Ebenen in einem Dokument können in einer Eltern-Kind-Beziehung zueinander stehen. Der Inhalt eines Dokuments kann entsprechend eine beliebige Baumstruktur widerspiegeln.

In der relationalen Modellierung und in klassischen SQL gibt es keine natürlichen Ausdrucksmittel zum Beschreiben von rekursiv hierarchischen Baumstrukturen. Allerdings existieren verschiedene Methoden, um solche Baumstrukturen indirekt zu modellieren.

Eine Variante zur Modellierung von Baumstrukturen mittels Relationen ist die Speicherung der transitiven Hülle der Eltern-Kind-Relation eines Baumes. Wird die vollständige transitive Hülle eines Baumes gespeichert, lassen sich Eigenschaften wie Zyklenfreiheit sehr einfach überprüfen.

Zunächst lässt sich die Prüfung auf Vollständigkeit der transitiven Hülle sehr einfach in SQL ausdrücken. Sollte die transitive Hülle der Relation nicht vollständig sein, lässt sich dies ebenfalls einfach korrigieren.

Nachdem die transitive Hülle verifiziert wurde, lässt sich die Zyklenfreiheit sehr elegant überprüfen. Die transitive Hülle einer zyklischen Relation ist nicht antisymmetrisch. Die Prüfung auf Antisymmetrie kann also verwendet werden, um Zyklenfreiheit zu garantieren.

Die Baumstruktur eines Dokuments wird daher im Web-Editor als transitive Hülle einer Eltern-Kind-Relation gespeichert. Abb. 8.9 zeigt das entsprechend erweiterte ER-Diagramm. Zu erkennen ist, dass rechts über der `Layer`-Definition eine reflexive Beziehung `nested in` hinzugefügt wurde. Darüber werden jedem Layer alle seine transitiven Kindknoten (Nachfahren, Descendants) in der Baumstruktur des Elements zugeordnet. Diese Beziehung hat ein zusätzliches Attribut `Depth` (Tiefe), welches den Abstand zwischen zwei Ebenen in der Hierarchie erfasst. Die Kardinalität der Beziehung ist in beide Richtungen mit $(1, N)$ angegeben. Da jede Ebene zu sich selbst in einer Eltern-Kind-Beziehung der Tiefe 0 steht, muss jede Ebene mindestens einen Vorfahren und einen Nachfahren haben.

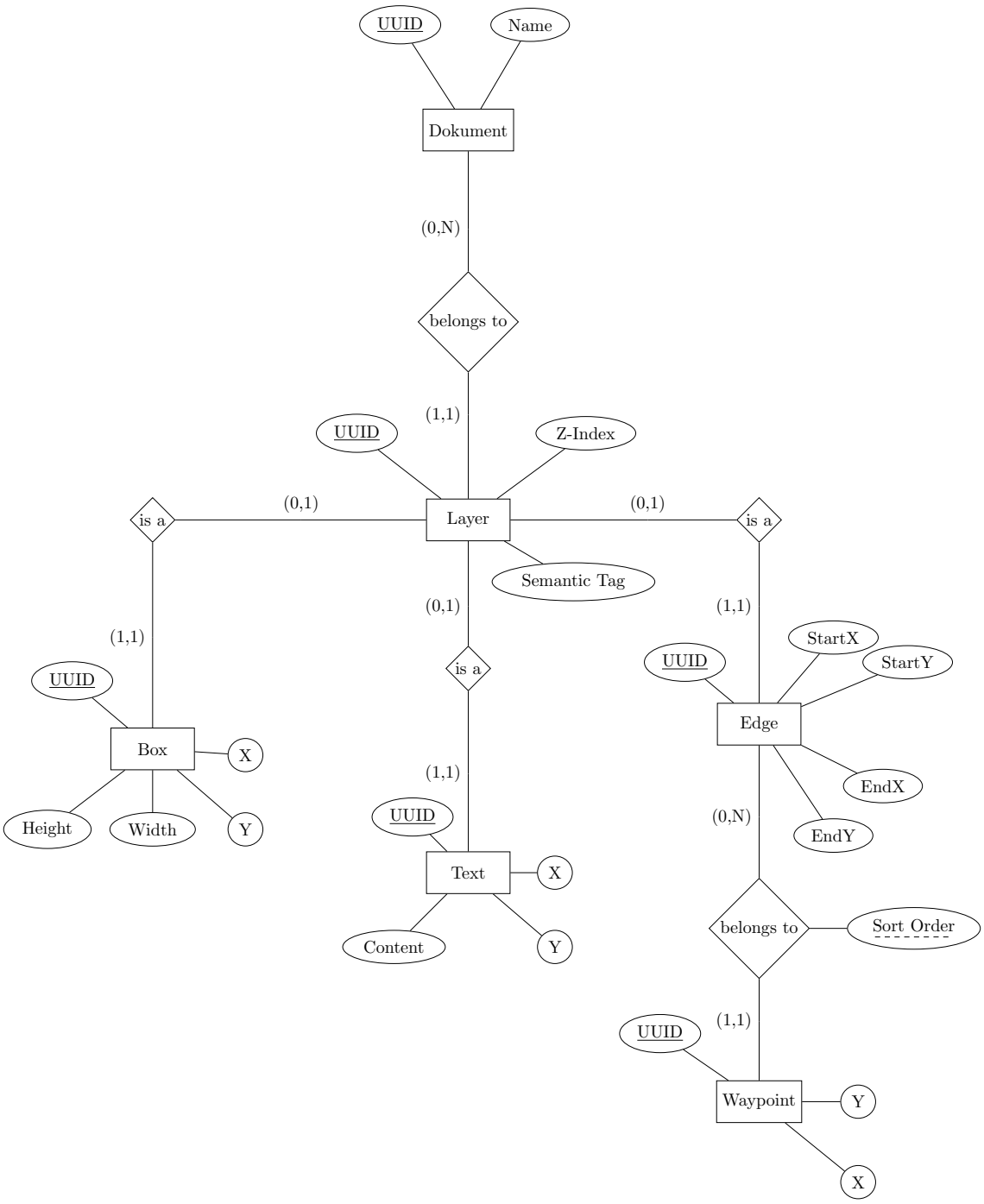


Abbildung 8.8.: ER-Diagramm zur Modellierung von Ebenen zur sortierten der Elemente in einem Dokument

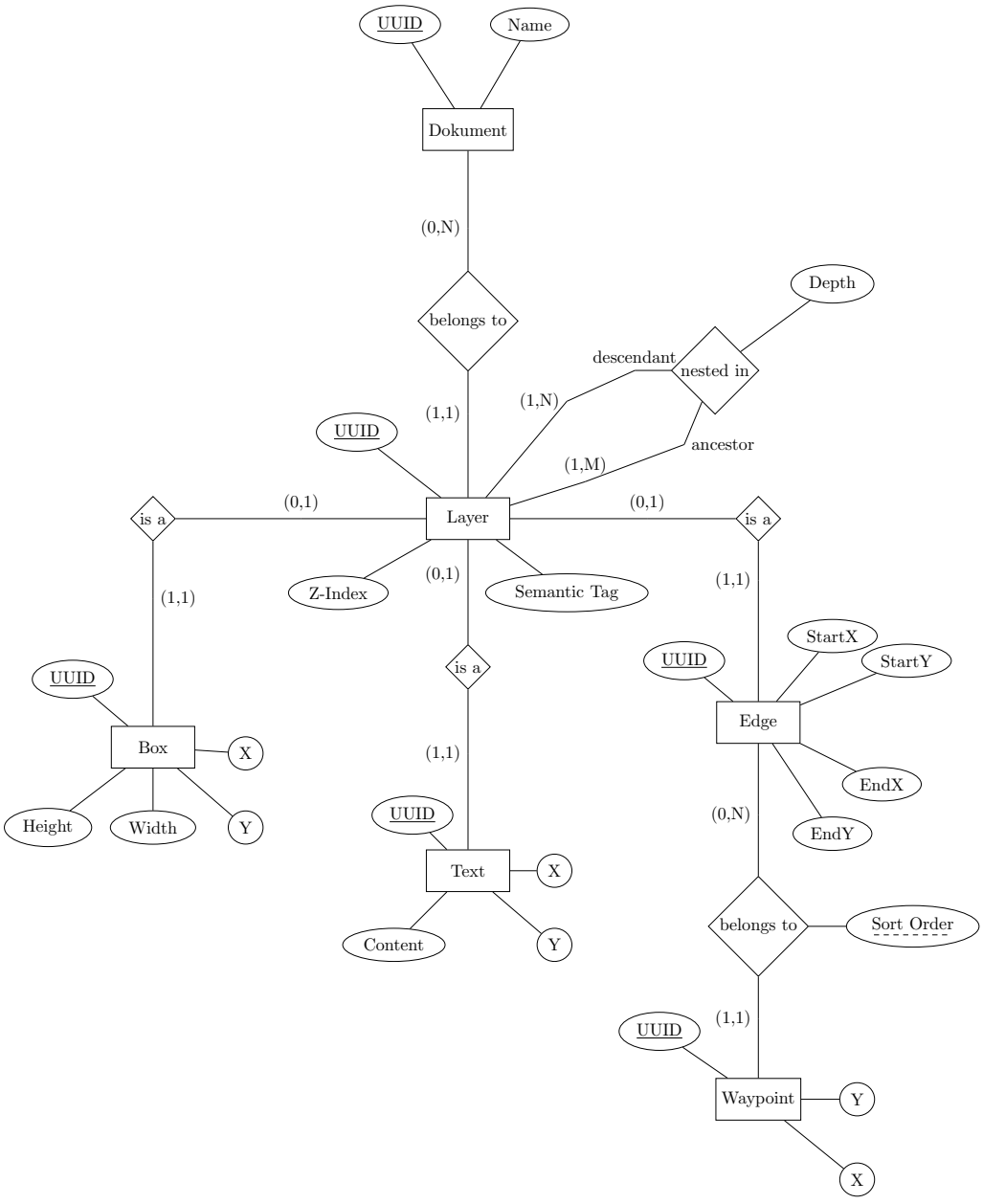


Abbildung 8.9.: ER-Diagramm zur Modellierung von hierarchischen Anordnung von Ebenen in einem Dokument

8.4.4. Darstellungsattribute

In Renew lässt sich die grafische Darstellung von Objekten durch zusätzliche Attribute steuern. Die soll auch im Web-Editor möglich sein. Die Menge und Art der Attribute unterscheidet sich aber je nach Figurelement. Im relationalen Modell kann dies durch das Hinzufügen weiterer Entitätstypen bzw. Relationen erreicht werden.

In Renew sind die Attribute eines Elements eine offene Menge, die beliebig erweitert werden kann. Elemente können also auch Attribute besitzen, die für die Darstellung gar nicht verwendet werden. Für die Darstellungsattribute im Web-Editor wurde eine endliche Menge an Darstellungsattributen definiert. Es ist also nicht möglich, beliebige Attribute zu Elementen hinzuzufügen, ohne das Datenmodell zu verändern. Das führt zwar theoretisch dazu, dass Renew Dokumente mit solchen zusätzlichen Attributen nicht verlustfrei importiert werden können, allerdings konnte im Rahmen dieser Arbeit nicht festgestellt werden, dass solche Dokumente überhaupt existieren. Gegebenenfalls kann das Datenmodell des Web-Editors aber um eine zusätzliche offene Menge an Attributen erweitert werden.

Abb. 8.10 zeigt die Erweiterung des Entity-Relationship-Diagramms um Darstellungsattribute. Ausgehend von Abb. 8.8 wurden die drei Entitätstypen `Layer Style`, `Edge Style` und `Text Style` hinzugefügt und über eine optionale Beziehung mit den `Layer`, `Edge` und `Text` Entitätstypen verknüpft. `Text Style` setzt sich aus den Attributen zur Beschreibung der Textdarstellung zusammen. `Edge Style` besteht aus den Attributen zur Beschreibung der Kantendarstellung. `Layer Style` besteht aus Attributen, die sowohl auf Textelemente als auch auf Boxelemente anwendbar sind.

8.4.5. Verbindungen zwischen Elementen

Die in Abschnitt 8.3.5 beschriebene Konnektoren müssen ebenfalls in das relationale Modell übertragen werden. Hierfür wird ähnlich vorgegangen wie für die Definition der geometrischen Symbole in Kapitel 7 „Parametrische Grafiksymbole“.

In Renew sind verschiedene `Connector`-Klassen definiert, die sich beliebig kombinieren lassen, um komplexe geometrische Beziehungen zwischen Objekten zu definieren. Jede einzelne `Connector`-Klasse könnte zudem beliebig aufwändige Berechnungen in Java durchführen, um ein spezifisches Ergebnis zu erzielen.

In Praxis kommen in Renew-Dokumenten aber nur eine sehr begrenzte Kombination an Konnektoren vor. In den meisten Fällen werden sie benutzt, um den Endpunkt eines Linienzuges auf dem Mittelpunkt einer Ellipse oder auf der Kontur eines Rechtecks zu verankern.

Für den Web-Editor wird in dieser Arbeit daher eine Reduktion der Ausdrucksstärke der Konnektoren vorgeschlagen. Ähnlich wie für die Definition der Grafiksymbole wird eine vordefinierte Menge an möglichen geometrischen Beziehungen definiert, die sich je nach Anwendungsfall parametrisieren lassen. Es können nur die Endpunkte von Linien-

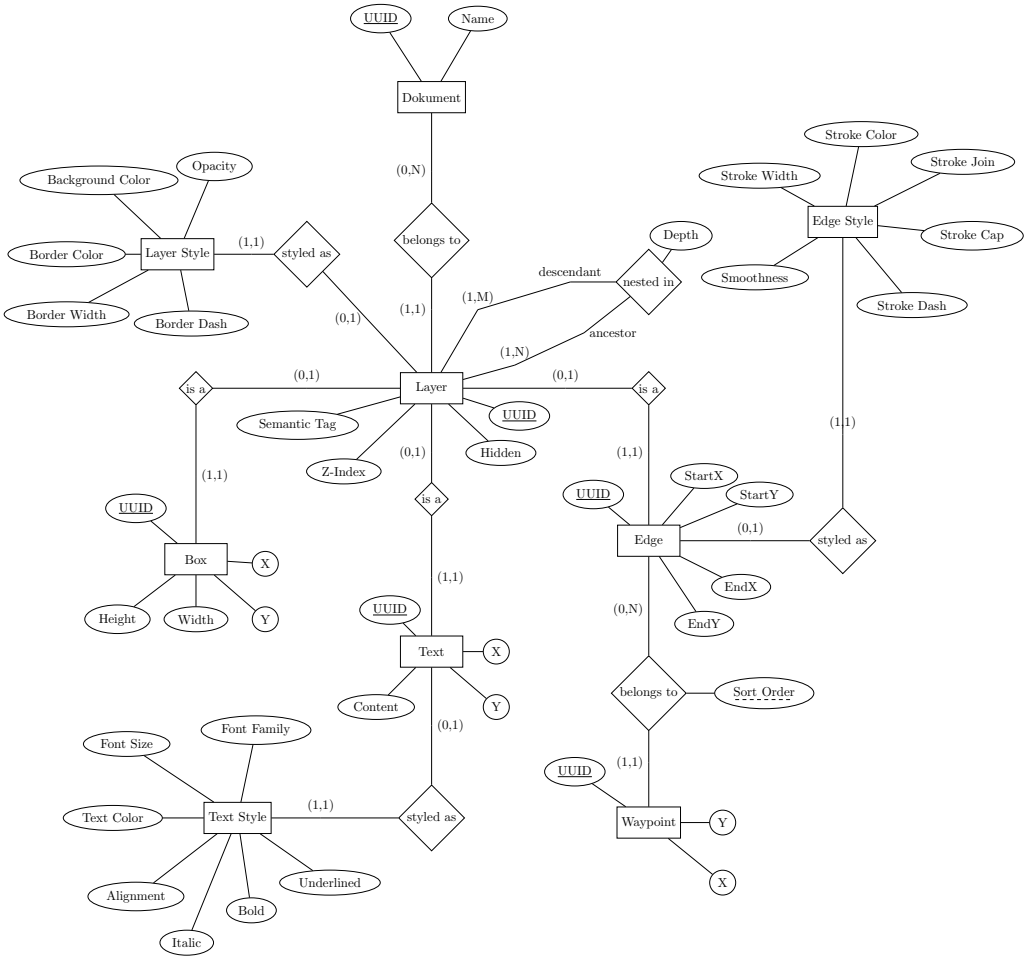


Abbildung 8.10.: Entity-Relationship-Diagramm zur Beschreibung von Darstellungsattributen in Renew-Dokumenten

zügen geometrisch an Positionen relativ zu anderen Elementen angeheftet werden. Diese Einschränkung zielt vor allem darauf ab, zyklische Abhängigkeiten zwischen Elementpositionen grundsätzlich auszuschließen. Bei Bedarf kann das Modell später erweitert werden.

Zusätzlich zur geometrischen Verknüpfung von Elementen und orthogonal zur Baumstruktur des Dokuments lassen sich Ebenen auch in eine zusätzliche semantische Beziehung zueinander setzen. Beispielsweise können Textelemente als Anschrift mit einer Kante oder einem Knoten eines Petrinetzes verknüpft werden. In Renew entspricht das dem `parent`-Feld der `TextFigure`-Klasse.

Außerdem können in Renew auch virtuelle Platzknoten und virtuelle Transitionen in einem Dokument platziert werden. Das sind grafisch duplizierte Darstellungen von Plätzen und Transitionen, die verwendet werden können, um die geometrische Darstellung eines Netzes von der topologischen Struktur zu entkoppeln und übersichtlicher zu gestalten.

Für sowohl die Verknüpfung von Textanschriften mit Kanten als auch die Verknüpfung von virtuellen Elementen wird im Datenmodell des Web-Editors eine einheitliche zusätzliche Relation namens `Hyperlink` definiert. Über diese lassen sich zwei beliebige Ebenen eines Dokuments in eine zweistellige Beziehung zueinander stellen.

Abb. 8.11 zeigt das um Verbindungen erweiterte ER-Diagramm. Zu erkennen ist eine zusätzliche reflexive Beziehung namens Hyperlink am Layer-Entitätstypen. Außerdem wurde eine Dreifachbeziehung (`attached to`) zwischen `Edge`, `Layer` und `Socket` hinzugefügt. Ein `Socket` entspricht hierbei dem Konzept eines Konrektors aus Renew und steht somit für die relativ zu einem Element bzw. zu einer Ebene verortete Position. Durch die Dreifachbeziehung wird also einem Endpunkt einer Kante eine Position relativ zu einem anderen Layer zugeordnet.

8.5. Parametrische Grafiksymbole im relationalen Schema

In Kapitel 7 „Parametrische Grafiksymbole“ wurde ein Datenformat zur Beschreibung der grafischen Figurelemente entwickelt. Dieses Format lässt sich ebenfalls im relationalen Schema abbilden. Durch die Übertragung der Grafiksymboldefinitionen in das relationale Datenmodell lassen sich die einzelnen Symbole auch als Datensätze zur Laufzeit bearbeiten und erweitern. Außerdem lassen sie sich einem Box-Element als Silhouette oder einer Kante als Pfeilspitze zuordnen.

Abb. 8.12 zeigt einen Ausschnitt aus dem entsprechend erweiterten ER-Diagramm. Es wurde ein neuer Entitätstyp `Symbol` definiert. Dieser ist über die Beziehung `shaped as` mit dem Entitätstyp `Box` verbunden und über die beiden Beziehungen mit `Source Tip` und `Target Tip` mit Edge Style. Außerdem lässt sich unterhalb des Entitätentyps `Symbol` erkennen, wie die in Kapitel 7 „Parametrische Grafiksymbole“ beschriebenen Grammatikregeln in Form von Entitätstypen nachgebildet wurden. Ein Symbol besteht aus einem oder mehreren sortierten Pfaden (Path). Ein Pfad besteht aus mehreren sor-

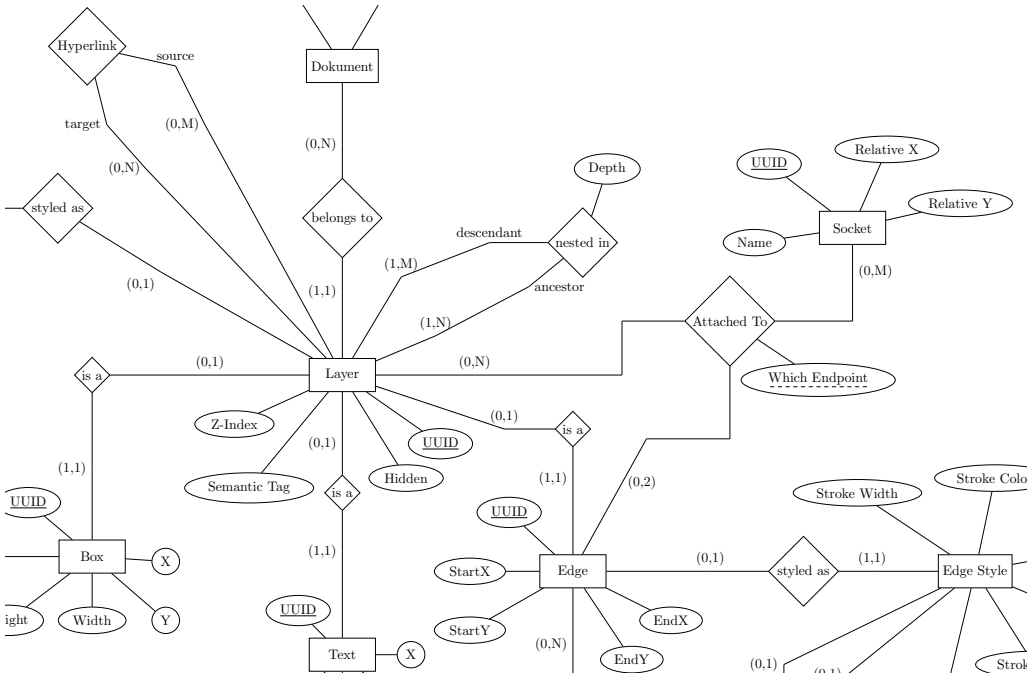


Abbildung 8.11.: Ausschnitt des ER-Diagramm zur Modellierung geometrischen Beziehung zwischen Elementen

tierten Segmenten (`Segment`). Ein Segment hat eine Startposition, die entweder absolut oder relativ zum Vorgänger positioniert ist. Ein Segment besteht aus mehreren sortierten Schritten (`Step`). Diese können sich wahlweise in horizontaler und vertikaler Richtung bewegen und entlang eines Kreisbogens (`Arc`) oder einer Bézierkurve gekrümmt sein (`Curvature`). Aus den vier $(0,1)$-Beziehungen der Step-Entität ergeben sich $4^2 = 8$ mögliche Kombinationen für Bewegungsrichtungen der Schritte Pfades.

8.6. Konvertierung von rnw-Dokumenten

Das vollständige relationale Schema wurde im Rahmen dieser Arbeit mittels Ecto in Elixir und SQLite implementiert.

Quellcode 8.1 zeigt beispielhaft einen Ausschnitt aus der Definition des Schemas in Elixir. Es werden die Entitätstypen `Document`, `Layer` und `Box` in Form von Elixir-Modulen definiert. Jeweils werden in einem `schema`-Deklarationsblock die zugehörigen Attribute und Beziehungen zu anderen Entitätstypen deklariert.

Um die Funktionstüchtigkeit des Entwurfs zu validieren, wurde auch ein Konvertierungsprozess entwickelt, um eine wie in Kapitel 6 „Import von rnw-Dateien“ eingelesene rnw-Datei in das relationale Schema zu importieren. Damit ist es im Web-Editor grundsätzlich möglich, jede beliebige existierende Renew Datei einzulesen.

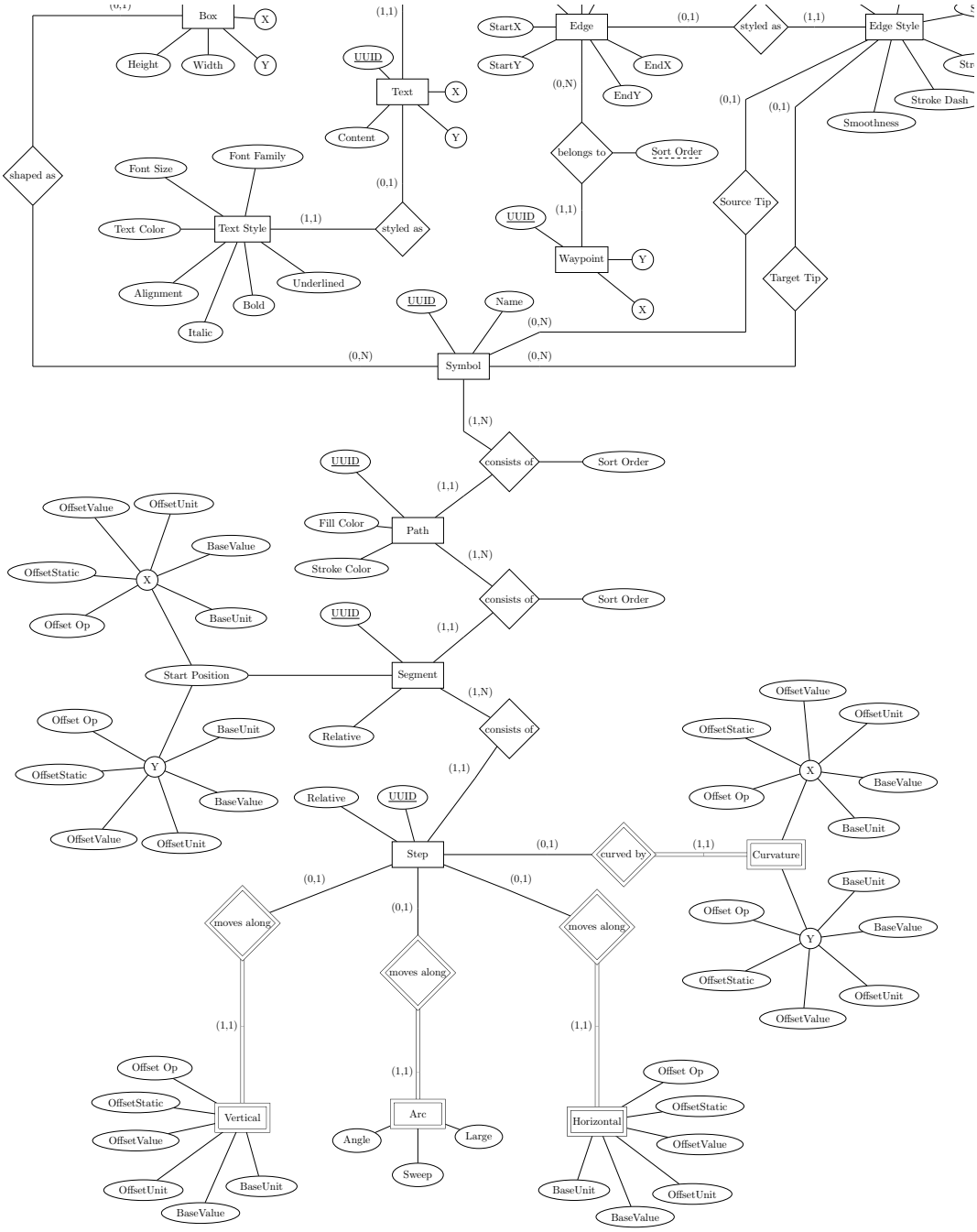


Abbildung 8.12.: ER-Diagramm zur Modellierung der parametrisches Grafiksymbole im relationalen Schema

```elixir
defmodule RenewCollab.Document.Document do
  @primary_key {:id, :binary_id, autogenerate: true}
  schema "document" do
    field :name, :string
    field :kind, :string

    has_many :layers, RenewCollab.Hierarchy.Layer,
      on_delete: :delete_all,
      preload_order: [asc: :z_index]

    has_many :snapshots, RenewCollab.Versioning.Snapshot

    has_one :latest_snapshot_marker, RenewCollab.Versioning.LatestSnapshot
    has_one :current_snaptshot, through: [:latest_snapshot_marker, :snapshot]

    timestamps(type: :utc_datetime)
  end
end

defmodule RenewCollab.Hierarchy.Layer do
  @primary_key {:id, :binary_id, autogenerate: true}
  schema "layer" do
    field :z_index, :integer
    field :semantic_tag, :string
    field :hidden, :boolean, default: false

    belongs_to :document, RenewCollab.Document.Document

    has_one :box, RenewCollab.Element.Box, on_delete: :delete_all
    has_one :text, RenewCollab.Element.Text, on_delete: :delete_all
    has_one :edge, RenewCollab.Element.Edge, on_delete: :delete_all
    has_one :style, RenewCollab.Style.LayerStyle, on_delete: :delete_all
  end
end

defmodule RenewCollab.Element.Box do
  @primary_key {:id, :binary_id, autogenerate: true}
  schema "element_box" do
    field :position_x, :float
    field :position_y, :float
    field :width, :float
    field :height, :float

    belongs_to :symbol_shape, RenewCollab.Symbol.Shape
    belongs_to :layer, RenewCollab.Hierarchy.Layer
  end
end
```

Quelltext 8.1: Ausschnitt der Ecto-Schemadefinition zur Umsetzung des relationalen Datenmodells

So wie die parametrischen Grafiksymbole und die Grammatik zum Lesen von rnw-Dateien wurde auch der Konvertierungsprozess als eigenständiges Modul entwickelt, welches sich unabhängig vom Web-Editor verwenden lässt [Kor25b]. Der aktuell veröffentlichte Stand im Hex-Paketregister entspricht aber zum aktuellen Zeitpunkt nicht dem neuesten im Web-Editor enthaltenen Entwicklungsstand.

9. Automatisches Layout von Kanten

Im vorherigen Kapitel wurde schon beschrieben, dass sich Elemente in Renew miteinander verknüpfen lassen, um das Aufrechterhalten von geometrischen Beziehungen zu erzwingen. In diesem Kapitel wird erklärt, wie die automatische Positionierung von Kanten im Web-Editor umgesetzt wird. Dafür wird zunächst untersucht, welche Arten der Positionierung in Renew zur Verfügung stehen. Als nächstes wird gezeigt, wie sich die Berechnungen in vereinfachter Form in den Web-Editor übertragen lassen. Außerdem wird erklärt, wie sich die Schnittpunktberechnungen zwischen Figuren und Kanten mittels Signed Distance Function (SDF) nachbilden lassen.

Abb. 9.1 zeigt ein Beispiel, in dem die Endpunkte zweier Kanten mit einem Start- und einem Zielknoten verknüpft sind. Die Pfeilspitzen zeigen genau auf die Kontur der verknüpften Ellipse. Das andere Ende der einen Linie zeigt genau auf die Mitte des verknüpften Rechtecks. Das Ende der anderen Linie zeigt auf die rechte Seitenkante des Rechtecks. Wenn nun das Rechteck oder die Ellipse bewegt werden, passt Renew die Position der Pfeilspitze an, damit diese weiterhin auf den Zielknoten zeigt. Diese Funktionalität soll genauso auch im Web-Editor enthalten sein.

9.1. Kantenkonnektoren in Renew

Zunächst muss genauer untersucht werden, wie die Berechnungen zur Positionierung in Renew vorgenommen werden.

Renew definiert die folgenden Java-Klassen, um eine feste Position relativ zu einem bestimmten Element zu beschreiben: `ShortestDistanceConnector`, `VSplitCenterConnector`, `HSplitCenterConnector`, `VerticalConnector`, `HorizontalConnector`, `TopConnector`, `RightConnector`, `BottomConnector`, `LeftConnector`, `HTopConnector` und `HBottomConnector`.

Die Klassen `ChopPolygonConnector`, `ChopBoxConnector`, `ChopRoundRectangleConnector`, `ChopEllipseConnector` und `ChopPieConnector` projizieren eine errechnete Koordinate auf die Kontur einer geometrischen Form.

Diese Projektion wird in Renew verwendet, damit Pfeilspitzen von Kanten sich stets außerhalb von den Knoten befinden, auf die sie zeigen. Außerdem können mittels der Klassen `OffsetLocator` und `RelativeLocator` absolute oder relative Feinjustierungen der Position entlang der X- oder Y-Achse vorgenommen werden.

9.2. Vereinfachung der Berechnung

Abgesehen von der Projektion auf die Kontur lassen die unterschiedlichen `Connector`-Klassen zu einer gemeinsamen Berechnung zusammenführen. Es ist die gleiche Berech-

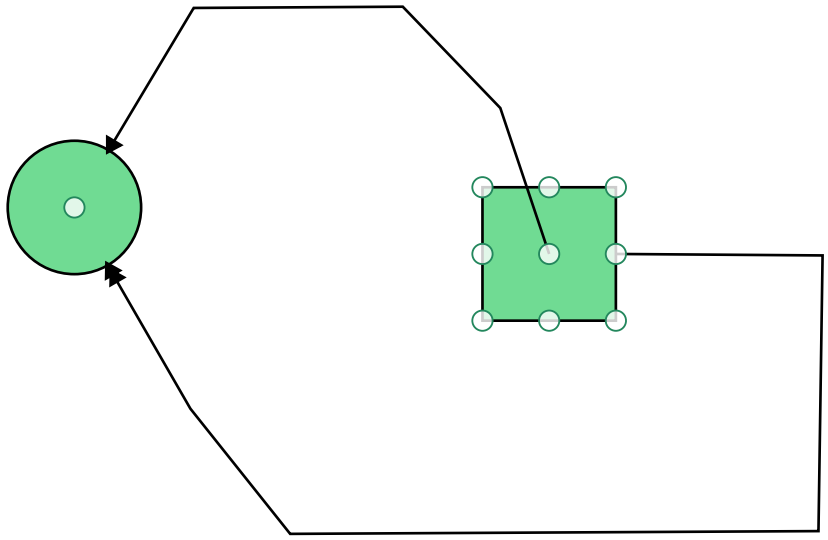


Abbildung 9.1.: Beispiel für zwei Kanten, deren Endpunkte jeweils an einen Knoten angeheftet sind.

nung, die auch für die Definition der geometrischen Symbole verwendet wurde. Abb. 9.2 zeigt eine Grammatik, die jede Berechnung, die in den `Connector`-Klassen durchgeführt wird, ausdrücken kann.

Somit sind für den Web-Editor nicht 18 verschiedene Datentypen zur Berechnung von Kantenpositionen nötig. Stattdessen kann das relationale Schema um einen Entitätstyp `Socket` erweitert werden. Abb. 9.3 zeigt das zugehörige ER-Diagramm. Ein Socket-Element entspricht einer Connector-Klasse in Renew. Ein Socket-Schema fasst mehrere Socket-Definitionen zu einer Gruppe zusammen. Ein Interface verknüpft ein Layer, also ein Element in einem Dokument, mit einem Socket-Schema. Es legt also fest, welche Sockets für die Verknüpfung mit einem Element zur Auswahl stehen. Eine Edge kann sowohl ihre Start- als auch ihre Zielposition mit dem Socket eines anderen Elements verknüpfen. Die X- und Y-Positionen eines Sockets sind durch jeweils sechs Parameter beschrieben. Durch diese wird die arithmetische Berechnung ausgedrückt, die in Quelltext9.2 bereits als Grammatik beschrieben wurde.

9.3. Raytracing

In Renew wird für die Projektion auf die Kontur einer geometrischen Form ein entsprechendes Gleichungssystem gelöst. Das Gleichungssystem hängt von der jeweiligen geometrischen Form ab. Entsprechend sind in Renew die zuvor genannten `Chop`-Klassen individuell für verschiedene geometrische Formen implementiert.

Für den Web-Editor wurde auf eine exakte analytische Lösung verzichtet. Stattdessen wird die Projektion mittels Raytracing angenähert [Gla89]. Raytracing ist ein Algorithmus,

⟨XY-position⟩ ⊨ ⟨expression⟩ , ⟨expression⟩
⟨expression⟩ ⊨ ⟨dim⟩ + ⟨func⟩ (⟨literal⟩ , ⟨dim⟩)
⟨dim⟩ ⊨ ⟨literal⟩ · ⟨unit⟩
⟨unit⟩ ⊨ `width` | `height` | `maxsize` | `minsize`
⟨func⟩ ⊨ `min` | `max` | `sum`
⟨literal⟩ ⊨ *Numeric Literal*

Abbildung 9.2.: Grammatik für einen Ausdruck zur verallgemeinerten Berechnung einer Position

der den Schnittpunkt zwischen einer Geraden und einer beliebigen geometrischen Form ermittelt, indem ein Punkt auf der Geraden iterativ vorwärtsbewegt wird, bis der Abstand zwischen Punkt und Zielgeometrie klein genug ist. Für einfache geometrische Formen wie Rechtecke und Ellipsen genügt eine begrenzte Anzahl von fünf Schritten, um ein visuell zufriedenstellendes Ergebnis zu erzielen. Quellcode 9.1 zeigt die Implementierung des Raytracers in Elixir.

9.4. Signed Distance Function

Damit der Raytracer die gesuchte Projektion annähern kann, muss ein Maß für den Abstand zwischen der aktuellen Position der Zielgeometrie, zum Beispiel einer Ellipse, angegeben werden. Eine einfache Möglichkeit, diesen Abstand anzugeben, ist eine SDF [LB09, Pat24].

Eine SDF ist eine Abbildung von einem Punkt in $\mathbb{R}^d$ auf einen Abstand $\mathbb{R}$. Für eine Vielzahl geometrischer Formen lassen sich bereits vordefinierte SDF finden [Qui22, LB09]. Ein Vorteil von SDF gegenüber einigen alternativen Repräsentationen ist es, dass einfache Formen algebraisch zu komplexeren Formen zusammensetzen lassen, ohne den Rechenaufwand oder den Programmieraufwand zu erhöhen. Quellcode 9.2 zeigt die Implementierung der SDF für ein Rechteck und eine Ellipse in Elixir.

Für den Web-Editor wurden nur die Ellipse und das Rechteck exemplarisch implementiert, aber alle Renew `Chop`-Klassen lassen sich prinzipiell als SDF ausdrücken. Somit lässt sich die Projektionsberechnung zur automatischen Positionierung durch Einsatz des Raytracers generisch umsetzen.

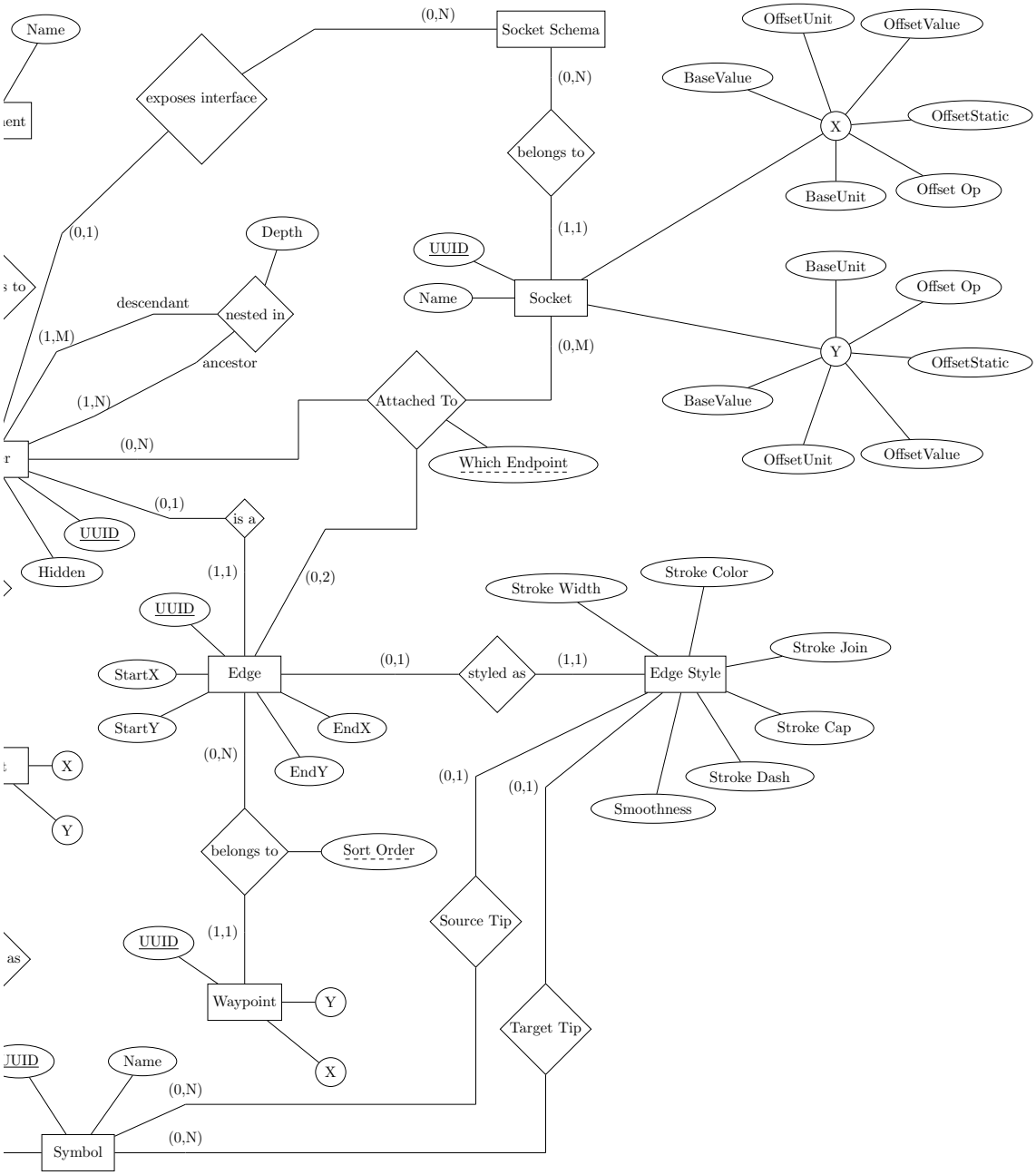


Abbildung 9.3.: ER-Diagramm zur Modellierung relativen Positionen als Sockets

```elixir
defmodule RenewexRouting.RayTracer do
    def nearest(stencil, box, {t_x, t_y} = target, {w_x, w_y} = waypoint) do
        dir_x = w_x - t_x
        dir_y = w_y - t_y
        dir_len = hypot(dir_x, dir_y)
        dir_x_norm = if(dir_len > 0, do: dir_x / dir_len, else: 0)
        dir_y_norm = if(dir_len > 0, do: dir_y / dir_len, else: 0)

        dist = Stream.unfold({t_x, t_y}, fn
          {x, y} ->
            d = RenewexRouting.Sdf.distance(stencil, box, {x, y})

            if d < 0 do
               { x + d * dir_x_norm, y + d * dir_y_norm} |> then(&{d, &1})
            end
        end)
        |> Stream.take(5)
        |> Enum.sum()

        %{
            position_x: t_x - dir_x_norm * dist,
            position_y: t_y - dir_y_norm * dist
        }
    end
end
```

Quelltext 9.1: Implementation eines einfachen Raytracers in Elixir

```elixir
defmodule RenewexRouting.Sdf do
    def distance(:rect, box, {x, y}) do
        box_center_x = box.position_x + box.width / 2
        box_center_y = box.position_y + box.height / 2
        {rel_x, rel_y} = {x - box_center_x, y - box_center_y}
        dist_x = abs(rel_x) - box.width / 2
        dist_y = abs(rel_y) - box.height / 2

        outside_distance = hypot(max(dist_x, 0), max(dist_y, 0))
        inside_distance = min(max(dist_x, dist_y), 0)

        outside_distance + inside_distance
    end

    @epsilon 0.00001
    def distance(:ellipse, box, {x, y}) do
        {rx, ry} = {box.width / 2.0, box.height / 2.0}
        box_center_x = box.position_x + rx
        box_center_y = box.position_y + ry
        {rel_x, rel_y} ={ x - box_center_x, y - box_center_y}

        k1 = max(hypot(rel_x / rx, rel_y / ry), @epsilon)
        k2 = max(hypot(rel_x / (rx * rx), rel_y / (ry * ry)), @epsilon)

        k1 * (k1 - 1.0) / k2
    end
end
```

Quelltext 9.2: Implementation einer Signed Distance Function in Elixir

10. Operationen zur Bearbeitung von Dokumenten

Nachdem nun das Datenmodell zur Repräsentation von Dokumenten vollständig definiert ist, können die einzelnen Operationen zum Bearbeiten von Dokumenten entworfen werden. In diesem Kapitel wird die Umsetzung dieser Bearbeitungsoperationen vorgestellt.

Dafür wird vom Kommando-Entwurfsmuster ausgegangen und gezeigt, wie sich durch die Trennung in schreibende Kommandos und in lesende Abfragen eine Menge kompositionierbarer Operationen definieren lässt, die sich atomar innerhalb einer Transaktion ausführen lassen.

Insgesamt wurden im Rahmen dieser Arbeit 40 Bearbeitungsoperationen implementiert. Eine dieser Operationen wird exemplarisch dargestellt.

10.1. Das Kommando-Entwurfsmuster

Das Kommando-Entwurfsmuster ist ein aus der objektorientierten Programmierung bekanntes Verhaltensmuster [GHJV95]. Es wird verwendet, um Befehle, die ausgeführt werden sollen, zuvor als Datenstruktur zu repräsentieren. Das erlaubt es, diese Befehle unabhängig von ihrer Ausführung durch das System zu reichen. Sie können beispielsweise in eine Warteschlange eingereiht oder in einem Protokoll vermerkt werden.

Im Web-Editor wird das Kommando Entwurfsmuster aus mehreren Gründen verwendet. Die Beschreibung jeder einzelnen Änderungsoperation als eigenständige Datenstruktur bildet eine sehr klare Schnittstelle. Die Menge der vom Editor erlaubten Bearbeitungsoperationen lässt sich exakt mittels der Menge an verfügbaren Kommandos angeben. Außerdem kann die einheitliche zentrale Verarbeitung aller Kommandos erzwungen werden. Dies ist im Kontext der funktionalen Programmierung wichtig, weil Änderungseffekte durch den Kontrollfluss des Systems propagiert werden müssen.

10.2. Trennung zwischen Zuständigkeit von Kommando und Abfrage

Das Command-Query-Responsibility-Segregation (CQRS)-Prinzip ist ebenfalls ein Entwurfsmuster für den Entwurf von Softwareservices [You10, Fow11]. Es verlangt, dass die Schnittstelle zum Auslesen eines aktuellen Zustands von der Schnittstelle zum Verändern des Zustands getrennt wird.

In der objektorientierten Programmierung definiert eine Klasse typischerweise ein Paar aus Methoden zum Lesen und Ändern eines Attributs. Dies stellt eine Verletzung des

CQRS Musters dar, weil schreibender und lesender Zugriff in einer gemeinsamen Schnittstelle einer Klasse vermischt werden.

In der Entwicklung des Web-Editors wurde das CQRS Muster eingesetzt, um die für Dokumente definierten Bearbeitungsoperationen vollkommen unabhängig von der späteren Darstellung von Dokumenten entwickeln zu können.

Der Einsatz des CQRS Musters führt im Web-Editor zu einem stark einseitig gerichteten Kontrollfluss. Dieser lässt sich während der Entwicklung und während der Fehlersuche leicht nachvollziehen.

10.3. Transaktionen als atomare Operationen

Ein weiterer Vorteil der strikten Verwendung der Kommando- und CQRS-Entwurfsmuster ist die einfache Umsetzung mittels relationaler Datenbank. Jedes Kommando kann in genau eine Datenbanktransaktion übersetzt werden.

Dadurch kann für jede durchgeführte Änderung garantiert werden, dass diese vollständig erfolgreich umgesetzt wurde. Außerdem können seitens der Datenbank auch alle aufgestellten Konsistenzbedingungen überprüft werden. Wird beispielsweise ein Knoten aus einem Netz gelöscht, kann garantiert werden, dass auch alle anliegenden Kanten gelöscht werden und die gesamte Operation ansonsten fehlschlägt.

10.4. Komposition von Operationen

Die Elixir-Datenbankschnittstelle Ecto erlaubt die Definition von Datenbanktransaktionen mittels DSL [ME11]. Die so definierten Transaktionen lassen sich außerdem zu zusammengesetzten Transaktionen komponieren. Somit konnten alle im Web-Editor definierten Änderungsoperationen sehr prägnant implementiert werden.

10.5. Beispiel einer Änderungsoperation

Eine der vom Web-Editor definierten Änderungsoperationen ist das Anheften eines Kantenendpunktes an den Fangpunkt (`socket`) eines anderen Elements. Quellcode 10.1 zeigt die Implementierung dieser Operation als Ecto-Datenbanktransaktion. In Zeile 2 werden die Parameter der Änderungsoperationen definiert, also in welchem Dokument welche Kante an welchen Socket von welcher anderen Ebene (`layer`) geknüpft werden soll. Von Zeile 11 bis 30 wird mittels Ecto DSL die Datenbanktransaktion aus einzelnen SQL-Operationen zusammengesetzt. In Zeile 31 wird die gesamte Operation mit einer weiteren Operation (`Bonding.reposition_multi`) konkateniert.

Insgesamt wurden für den Web-Editor über 40 verschiedene Bearbeitungsoperationen definiert. Tabelle 10.1 zeigt eine Übersicht über alle Operationen gruppiert nach ihrer Auswirkung.

Operation	Beschreibung
`assign_layer_socket_schema`	
`create_document`	
`create_edge_bond`	
`create_layer_edge_waypoint`	
`create_layer`	
`create_parent_layer`	
`create_snapshot_label`	
`create_snapshot`	
`delete_bond`	
`delete_document`	
`delete_layer_edge_waypoint`	
`delete_layer`	
`update_document_meta`	
`update_box_shape`	
`update_box_size`	
`update_edge_attributes`	
`update_edge_position`	
`update_edge_style`	
`update_edge_waypoint_position`	
`update_semantic_tag`	
`update_style`	
`update_text_body`	
`update_text_position`	
`update_text_size_hint_auto`	
`update_text_size_hint`	
`update_text_style`	
`update_layer_z_index`	
`duplicate_document`	
`insert_document`	
`link_layer`	
`make_space_between`	
`move_layer_relative`	
`normalize_z_index`	
`prune_snapshots`	
`remove_all_layer_edge_waypoints`	
`remove_layer_socket_schema`	
`remove_snapshot_label`	
`reorder_layer_relative`	
`reorder_layer`	
`restore_snapshot`	
`set_visibility`	
`toggle_visible`	
`unlink_layer`	

Tabelle 10.1.: Liste der serverseitig definierten Bearbeitungsoperationen des Web-Editors

```elixir
defmodule RenewCollab.Commands.CreateEdgeBond do
  defstruct [:document_id, :edge_id, :kind, :layer_id, :socket_id]

  def multi(%__MODULE__{
        document_id: document_id,
        edge_id: edge_id,
        kind: kind,
        layer_id: layer_id,
        socket_id: socket_id
      }) do
    Ecto.Multi.new()
    |> Ecto.Multi.insert(
      :new_bond,
      %Bond{}
      |> Bond.changeset(%{
        element_edge_id: edge_id,
        kind: kind,
        layer_id: layer_id,
        socket_id: socket_id
      })
    )
    |> Ecto.Multi.all(
      :affected_bond_ids,
      fn %{new_bond: new_bond} ->
        from(bond in Bond,
          where: bond.element_edge_id == ^new_bond.element_edge_id,
          select: bond.id
        )
      end
    )
    |> Ecto.Multi.append(RenewCollab.Bonding.reposition_multi())
  end
end
```

Quelltext 10.1: Beispiel eines Editor Commands als Ecto Transaktion

11. Versionierung von Dokumenten

In Kapitel 4 „Anforderungen“ wurde die Anforderung aufgestellt, dass von Benutzern und Benutzerinnen durchgeführte Änderungen an einem Dokument sich auch einfach rückgängig machen lassen sollen. In diesem Kapitel wird die Umsetzung der hierfür nötigen Schritte beschrieben. Es wird erklärt, wie nach jeder Veränderung eine Kopie des gesamten Dokuments gespeichert wird und wie diese Kopien in einer zeitlichen Struktur geordnet und in das relationale Schema gespeichert werden.

11.1. Schnappschüsse

Damit durchgeführte Änderungen sich ungeschehen machen lassen, muss ein Protokoll der durchgeführten Änderungen aufgezeichnet werden. Dies kann grundsätzlich auf zwei verschiedene Arten umgesetzt werden. Entweder nur die Änderungsoperationen zusammen mit ihren Parametern werden protokolliert oder der Gesamtzustand vor und nach einer Änderung als sogenannter Schnappschuss wird protokolliert.

Damit eine Änderung sich nur anhand der protokollierten Operation rückgängig machen lässt, muss die Operation vollständig invertierbar sein. Damit sich beispielsweise eine Löschoperation rückgängig machen lässt, muss sich die gelöschte Information auch vollständig aus dem Protokoll rekonstruieren lassen. Dies ist im allgemeinen Fall nicht einfach für jede denkbare Operation möglich.

Wird hingegen stets der vollständige Gesamtzustand eines Dokuments protokolliert, kann eine vergangene Version sehr einfach wiederhergestellt werden, unabhängig von der durchgeführten Operation. Der Nachteil dieses Vorgehens ist aber der erhöhte Speicherverbrauch, da für jede kleine Änderung eine vollständige Kopie des Dokuments gespeichert werden muss.

Für den Web-Editor wurde die Protokollierung des Gesamtzustandes als Schnappschuss umgesetzt. Um den benötigten Speicherplatz zu reduzieren, werden die Kopien der Dokumente mittels Deflate-Algorithmus komprimiert [OSK16, Bac09].

11.2. Kopplung an Transaktionen

In Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ wurde bereits erklärt, wie alle vom Web-Editor implementierten Änderungsoperationen in Form des Kommando-Entwurfsmusters umgesetzt wurden. Diese Operationen werden an zentraler Stelle verarbeitet und in Datenbanktransaktionen übersetzt.

Die Protokollierung von Dokumentenversionen konnte in der Entwicklung des Web-Editors daher sehr einfach nachgerüstet werden. An zentraler Stelle müssen alle verarbeiteten Transaktionen nur um einen weiteren entsprechenden Schritt erweitert werden.

Vor Abschluss einer jeden Transaktion wird der aktuelle Zustand des betreffenden Dokuments ausgelesen, serialisiert, komprimiert und als neuer Eintrag im Versionsprotokoll abgespeichert.

Durch die Kopplung der Versionierung an die atomaren Datenbanktransaktionen wird auch garantiert, dass ein Protokolleintrag nur für erfolgreiche Änderungen angelegt wird, und andersherum, dass für jede erfolgreiche Änderung auch ein Protokolleintrag angelegt wird.

Ein Nachteil dieser Kopplung ist, dass die Dauer einer Operation nicht nur vom Umfang der Operation, sondern auch von der Speicherung des Protokolleintrags abhängt. Für kürzere Latenzzeiten könnte die Versionierung also entkoppelt werden. Allerdings wurde für den Web-Editor die stark gekoppelte Umsetzung bevorzugt, um Wettlaufsituationen zu vermeiden.

11.3. Präzedenzgraph

Damit in jeder Situation die passende Version wiederhergestellt werden kann, muss die Menge aller protokollierten Versionen mittels einer Vorgängerrelation geordnet werden. Für ein Dokument muss zugeordnet werden, welche die aktuelle Version ist. Für jede einzelne Version muss gespeichert werden, welche die vorherige Version ist, aus der sie entstanden ist. So entsteht ein gerichteter azyklischer Graph zwischen den Versionen im Protokoll.

Dieser Graph repräsentiert den logischen Zeitstrahl, entlang dem das versionierte Dokument existiert. Dieser Zeitstrahl kann verzweigen, wenn mehrere Versionen denselben Vorgänger besitzen. In Versionskontrollsystemen wie Git können verzweigte Versionsgraphen per Merge-Operation wieder zusammengeführt werden. In Softwareanwendungen wie Photoshop, Office oder auch Renew wird grundsätzlich nur eine lineare Historie gespeichert. Sobald ein neuer Nachfolger für eine Version erstellt wird, wird der bisherige Nachfolger gelöscht.

Für den Web-Editor wurde keine Merge-Operation umgesetzt. Es werden aber alle auch verzweigten Pfade im Versionsgraphen gespeichert. Benutzer:innen müssen also manuell in diesem Graphen navigieren und sich darauf einigen, welches die aktuellste Version eines Dokuments ist, an der sie weiterarbeiten wollen.

Abb. 11.1 zeigt einen exemplarischen Präzedenzgraph. Die runden Knoten sind die Versionen des Dokuments, für die ein Schnappschuss gespeichert wurde. Die Pfeile zwischen ihnen stellen die linkseindeutige Vorgängerrelation dar. Versionen, die keinen anderen Vorgänger haben, sind ihr eigener Vorgänger. Das Dokument besitzt eine eindeutige Referenz auf die aktuelle Version. Der Graph ist nicht zusammenhängend, weil durch eventuelle Löschungen von Aufzeichnungen Lücken entstehen können. Rechts in der Abbildung sind beispielsweise 3 rote Knoten zu sehen, die mit dem Rest nicht verbunden sind. Von der aktuellen Version aus gibt es immer genau einen Vorgänger, aber eventuell mehrere Nachfolger.

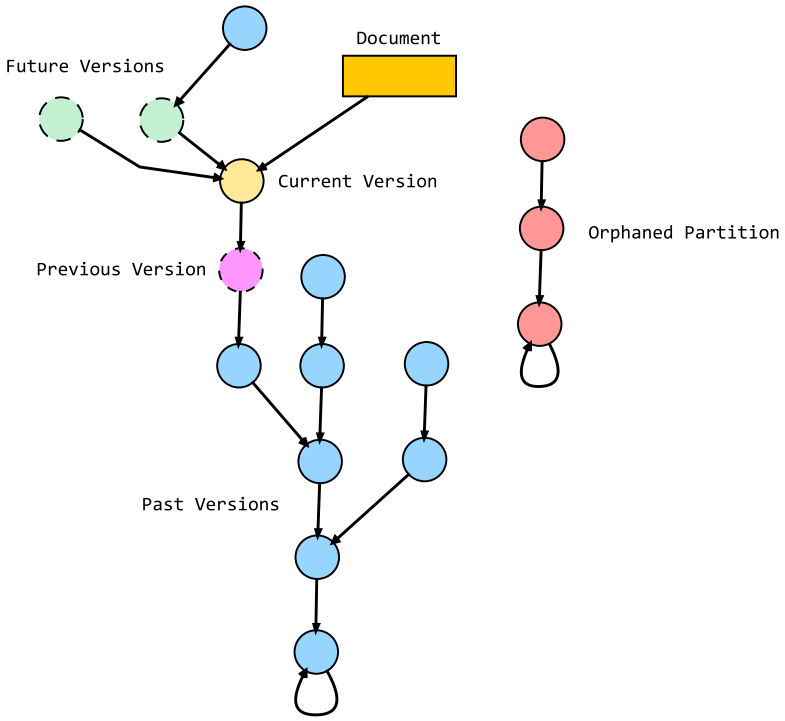


Abbildung 11.1.: Präzedenzgraph der Versionshistorie eines Dokuments

11.4. Speicherung im Relationalen Modell

Die serialisierten Dokumentenversionen und deren Vorgängerrelation werden in der gleichen relationalen Datenbank gespeichert wie die Dokumente selbst. In Abbildung 11.2 ist ein Ausschnitt eines ER-Diagrams dargestellt. Darin ist zu erkennen, wie Dokumentenversionen (Snapshot) über eine 1:N-Beziehung immer genau einem Dokument zugeordnet sind. Diese Zuordnung ist mit einem Zeitstempel ausgezeichnet. Außerdem ist zu erkennen, dass die Dokumentenversionen untereinander über eine Vorgänger-Beziehung (precedence) verknüpft sind. Der gesamte Inhalt einer gespeicherten Dokumentenversion wurde als ein einzelnes zusammengefasstes Attribut (Content) modelliert.

Eine Alternative wäre es gewesen, den Inhalt der vergangenen Dokumentenversionen auf mehrere Entitäten aufzuteilen und in einem genauso ausgefeilten Datenschema zu speichern, wie die Dokumente selbst. Der Nachteil hierbei wäre aber, dass bei jeder zukünftigen Änderung des Datenbankschemas die potenziell große Menge an gespeicherten Dokumentenversionen verlustfrei migriert werden müsste. In der jetzigen Umsetzung können Dokumentenversionen, die zu unterschiedlicher Zeit entstanden sind und die sich somit potenziell in ihrem Datenschema unterscheiden, problemlos in der Datenbank verweilen. Nach einer Veränderung am Datenbankschema kann es zwar schwierig sein, eine alte Dokumentenversion wiederherzustellen, doch es kann sichergestellt werden, dass diese in ihrem ursprünglichen Zustand noch existiert.

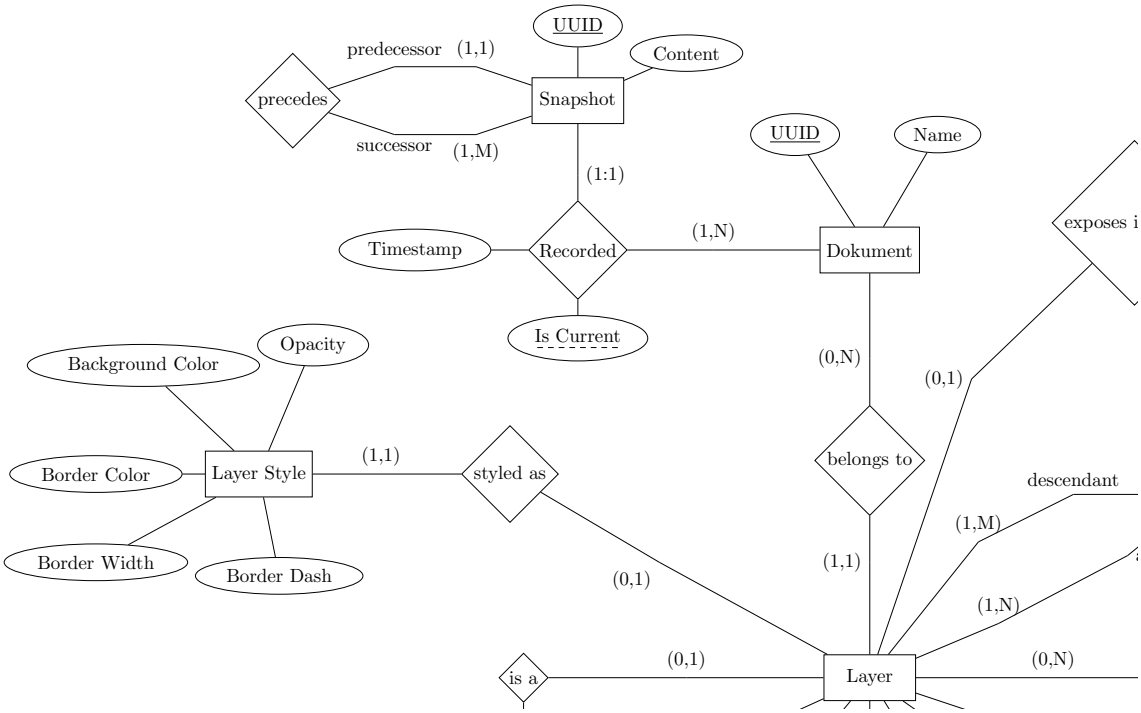


Abbildung 11.2.: Ausschnitt des ER-Diagramms zur Modellierung der Versionshistorie im relationalen Datenmodell.

11.5. Langfristige Speicherung

Im Web-Editor werden vergangene Versionen von Dokumenten grundsätzlich nicht automatisch gelöscht, sondern dauerhaft gespeichert. Versionen können bei Bedarf aber manuell gelöscht werden.

Zum einen erlaubt die dauerhafte Speicherung den Zugriff auf beliebig alte Zwischenstände. Anwender:innen müssen sich keine Gedanken darüber machen, zu sinnvollen Zeitpunkten ein Zwischenergebnis zu speichern. Zum anderen musste während der Entwicklung des Web-Editors auch gar keine automatisierte Strategie entworfen werden, um zu entscheiden, wann genau ein Protokolleintrag gelöscht werden kann.

Für den langfristigen Einsatz des Web-Editors mit großen Mengen an Dokumenten und vielen Anwenderinnen und Anwendern wäre es aber sinnvoll, eine Strategie zum automatisierten Löschen von Versionsprotokollen zu entwickeln.

12. Export von rnw-Dateien

In diesem Kapitel wird der Export von Dokumenten aus dem relationalen Schema zurück in eine von Renew verarbeitbare rnw-Datei beschrieben. In Kapitel 6 „Import von rnw-Dateien“ wurde bereits die Grammatik zum Einlesen von rnw-Dateien vorgestellt. In Kapitel 8 „Relationales Datenschema“ wurde die vom Web-Editor intern verwendete relationale Datenbank entwickelt. Der Export soll es zum einen ermöglichen, die Dokumente in Renew weiter bearbeiten zu können. Zum anderen erwartet auch der Renew-Simulator die zu simulierenden Netze im rnw-Format.

12.1. Bearbeitung in Renew

Eine Anforderung aus Kapitel 4 „Anforderungen“ war es, dass im Web-Editor erstellte Dokumente sich auch in Renew einlesen und dort weiter bearbeiten lassen.

Um dies zu ermöglichen, wurde ein Exportprozess entwickelt, der die Daten aus dem relationalen Modell des Editors in das rnw-Textformat serialisiert. Hierfür wird die schon in Kapitel 6 „Import von rnw-Dateien“ definierte Grammatik verwendet. Nur dass diese nun nicht zum Lesen, sondern zum Produzieren von rnw-Dokumenten verwendet wird.

Besonders wichtig, aber auch schwierig, beim Export von rnw-Dateien ist die korrekte Aufrechterhaltung der semantischen Verknüpfungen zwischen allen Elementen.

12.2. Simulation in Renew

Die korrekte Serialisierung von rnw-Dateien durch Web-Editor ist nicht nur wichtig, damit Anwenderinnen die Dokumente in Renew weiter bearbeiten können, sondern auch, damit der Web-Editor die gezeichneten Netze an den Renew Simulator schicken kann, um sie von diesem verarbeiten zu lassen.

Wie genau der Web-Editor an den Simulator angeschlossen wird, ist Thema von Kapitel 20 „Anbindung an Java Renew“. An dieser Stelle kann aber schon verraten werden, dass der Simulator die zu simulierenden Netze als rnw-Dokument erwartet.

Die vom Simulator erzeugte Ausgabe hängt von optional in der rnw-Datei enthaltenen Annotationen ab. Damit die Simulationsergebnisse vom Web-Editor korrekt interpretiert und dargestellt werden können, müssen alle serialisierten Elemente zusätzlich mit eindeutigen Identifizierungen versehen werden. Der Export von rnw-Dokument für die Simulation unterscheidet sich also von dem für Anwenderinnen vorgesehenen Exportformat.

12.3. Unstimmigkeit zwischen Schemata

Die größte Schwierigkeit beim Export von rnw-Dokumente aus dem Web-Editor ist die Unstimmigkeit zwischen dem objektorientierten Datenmodell von Renew und dem relationalen Modell des Web-Editors. In Kapitel 8 „Relationales Datenschema“ wurde zwar ausgiebig motiviert, warum das Datenmodell des Web-Editors von Renew abweichen soll. Außerdem wurde penibel darauf geachtet, dass das entwickelte relationale Modell vollständig kompatibel mit dem Import von rnw-Dateien ist. Doch für die umgekehrte Richtung, also für den Export aus dem relationalen Modell in rnw-Dateien, gilt diese Kompatibilität nicht automatisch.

Im Web-Editor lassen sich Dokumente erzeugen, die sich in Renew nicht erzeugen lassen, die also nicht als rnw-Datei darstellbar sind. Beispielsweise können in der Ebenenhierarchie des Web-Editors, die in Kapitel 8 „Relationales Datenschema“ Abschnitt 8.4.3 entwickelt wurde, beliebige Elemente eines Dokuments in beliebige Reihenfolge und Tiefe miteinander verschachtelt werden. In Renew können nur einige bestimmte Figurelemente als Elternknoten in der Hierarchie eines Dokuments auftreten.

Außerdem wurden die 200 verschiedenen Arten von Renew `Figure`-Elementen im Datenmodell des Web-Editors wegnormalisiert und von ihrer Darstellung als grafisches Symbol entkoppelt. Der Vorteil im Web-Editor ist, dass sich für jedes Element im Dokument die grafische Darstellung frei wählen und auch nachträglich ändern lässt. Der Nachteil beim Export ist, dass sich nicht alle im Web-Editor erzeugbaren Elemente einer Renew Java-Klasse zuordnen lassen. Für jeden Export in eine rnw-Datei muss aber jedem Element eine passende Java-Klasse zugeordnet werden.

Für die Entwicklung des Web-Editors in dieser Arbeit wurde sich darauf beschränkt, die einfach zuordenbaren Elemente zu exportieren und alle weiteren Elemente nicht weiter zu beachten. Bei Bedarf kann der Exportprozess weiter ausgearbeitet werden. Für die Verwendung des Web-Editors bedeutet dies, dass die Anwenderinnen eigenständig darauf achten müssen, nur Dokumente zu erstellen, die auch in Renew erstellbar wären.

13. Web-Schnittstelle des Editor Servers

In den vorherigen Kapiteln wurde erklärt, wie der Web-Editor rnw-Dateien einlesen, diese intern repräsentieren, sie mittels definierter Operationen manipulieren und auch wieder als rnw-Datei exportieren kann. In diesem Kapitel wird die Bereitstellung von Dokumenteninhalten behandelt. Der Export als rnw-Datei eignet sich nur für die Kompatibilität mit Renew aber nicht für die interaktive grafische Darstellung eines Dokumentes im Web-Editor.

Die in dem bisherigen Teil dieser Arbeit besprochene Server-Komponente des Web-Editors umfasst keine grafische Darstellung der Dokumente. Die grafische Darstellung und die Umsetzung der interaktiven Benutzeroberfläche werden im nächsten Teil behandelt.

Die in diesem Kapitel vorgestellten Schnittstellen sind darauf ausgelegt, von einer Client-Anwendung verwendet werden zu können, um eine grafische Benutzeroberfläche zum Bearbeiten von Dokumenten zu entwickeln.

13.1. HTTP-Schnittstelle

Für den lesenden Zugriff auf Dokumente stellt der Web-Editor eine Schnittstelle über das HTTP-Protokoll bereit. Eine sogenannte REST application programming interface (API) kann verwendet werden, um in Client-Server-Anwendungen zwischen Client und Server zu kommunizieren [Fie00].

Jedem im Web-Editor gespeicherten Dokument ist ein eindeutiger Uniform Resource Locator (URL) zugeordnet. Wird über das HTTP-Protokoll eine `GET` Anfrage an den jeweiligen URL geschickt, antwortet der Editor-Server mit dem serialisierten Inhalt des Dokuments. Der vollständige Inhalt des Dokuments wird aus der Datenbank gelesen und in das JSON-Format serialisiert [Bra17].

Das JSON-Format ist stark an die Struktur des in Kapitel 8 „Relationales Datenschema“ erarbeiteten relationalen Schemas angelehnt. Entsprechend des schon in Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ verwendeten CQRS Prinzips ist das JSON-Format nicht dafür vorgesehen, dass der vom Server gelieferte Dokumenteninhalt eigenständig verändert wird. Alle Änderungsoperationen müssen als Kommandos vom Server durchgeführt werden.

Zusätzlich zum Inhalt einzelner Dokumente können über jeweils entsprechende URLs auch weitere Informationen vom Server abgefragt werden. Beispielsweise kann die Liste aller Dokumente, die Liste aller Versionen eines Dokuments oder die Liste aller verfügbaren URLs abgefragt werden.

Tabelle 13.1 zeigt die Endpunkte der HTTP-Schnittstelle, über die auf die Dokumente zugegriffen werden kann. Wie zu erkennen ist, lassen sich Dokumente auflisten, importieren, exportieren, duplizieren und löschen. Als Antwort auf Anfragen an diese Endpunkte

wird jeweils eine JSON-Struktur mit den jeweils relevanten Informationen geliefert. Die genaue Struktur der jeweiligen Antwort kann der Implementierung entnommen werden und wird hier nicht ausführlich dokumentiert.

Listing 13.1 zeigt eine exemplarische HTTP-Nachricht, zum Abfragen des Inhalts eines Dokuments vom Server. Listing 13.2 zeigt die zugehörige Antwort vom Server. Sie enthält die Liste aller im Dokument befindlichen Elemente (`layers]`), die aktuellen Ausmaße des Dokuments (`viewbox`) und die UUID der aktuellen und vorherigen Version des Dokuments (`snapshot`).

Methode	Pfad	Beschreibung
GET	/documents	Liste aller Dokument
POST	/documents/import	rnw-Datei importieren
GET	/documents/:id/export	rnw-Datei exportieren
GET	/documents/:id/download.iex	Dokument als Elixir-Datenstruktur
GET	/documents/:id/download.json	Dokument als JSON
POST	/documents/:id/duplicate	Dokument duplizieren
POST	/documents	Dokument Erstellen
PUT	/documents/:id	Dokument umbenennen
DELETE	/documents/:id	Dokument löschen

Tabelle 13.1.: Endpunkte der HTTP-Schnittstelle zur Verwaltung von Dokumenten

```
1 GET /api/documents/f3b690be-a9f3-48c7-af2e-c4b5e88135c6 HTTP/2
2 Host: api.webeditor.exemple
3 Content-Type: application/json
4 Authorization: Bearer secret...
5 Origin: https://api.webeditor.exemple
```

Quelltext 13.1: HTTP API Anfrage

13.2. WebSocket-Schnittstelle

Das HTTP-Protokoll eignet sich nur für die aktive Abfragen der Serverinhalte durch einen Client. In einer interaktiven Anwendung wie dem Web-Editor können Änderungen an Dokumenten aber jederzeit auftreten. Im Fall von durchgeführten Änderungsoperationen muss der Editor-Server auch proaktiv über diese informieren können.

Auf Basis des WebSocket-Protokolls kann eine bidirektionale Verbindung zwischen einem Client und dem Editor-Server aufgebaut werden. Über diese Verbindung kann der Server die jeweils aktuellste Version eines Dokuments an alle verbundenen Clients schicken.

Für die Implementation der WebSocket-Schnittstelle des Web-Editors wurde in dieser Arbeit die Elixir Phoenix Channel Programmbibliothek verwendet [ME23a]. Phoenix stellt einen vordefinierten robusten Workflow zur Verwaltung von aktiven Verbindungen bereit. Verbindungsabbrüche werden erkannt und entsprechend sinnvoll behandelt.

```
HTTP/2 200
content-type: application/json; charset=utf-8

{
  "id": "f3b690be-a9f3-48c7-af2e-c4b5e88135c6",
  "href": "/api/documents/f3b690be-a9f3-48c7-af2e-c4b5e88135c6",
  "content": {
    "name": "example document",
    "kind": "de.renew.gui.CPNDrawing",
    "layers": {
      "items": [
        {
          "hidden": false,
          "id": "a2bc416b-930b-435d-8d0e-6da92ddde29b",
          "parent_id": null,
          "z_index": 0,
          "box": {
            "height": 50,
            "position_x": -52,
            "position_y": -51,
            "shape": "3B66E69A-057A-40B9-A1A0-9DB44EF5CE42",
            "shape_attributes": null,
            "width": 50
          },
          "semantic_tag": "de.renew.gui.PlaceFigure",
          "interface_id": "2C5DE751-2FB8-48DE-99B6-D99648EBDFFC"
        }
      ]
    },
    "snapshot": {
      "current_id": "f1c0fcb6-223a-4702-9d39-184ff4545b07",
      "next_ids": [],
      "prev_id": "4c9e6a9c-0261-4868-9e9d-e42eaa7ecd8d"
    },
    "viewbox": {
      "width": 250,
      "y": -151.95762794756246,
      "x": -152.90157433712122,
      "height": 250
    }
  },
  "topic": "document:f3b690be-a9f3-48c7-af2e-c4b5e88135c6"
}
```

Quelltext 13.2: HTTP API Antwort

Über die WebSocket-Verbindung können Clients auch Änderungsoperationen an den Server schicken. Die in Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ definierten Kommandos zur Manipulation von Dokumenten können über die WebSocket-Schnittstelle aufgerufen werden. Dafür muss der Name eines Kommandos zusammen mit den nötigen Parameterwerten als JSON kodiert über den WebSocket an den Server gesendet werden.

Tabelle 13.2 enthält eine Auflistung der verschiedenen Kommandos zusammen mit den zugehörigen Parametern. Die genaue Semantik der einzelnen Kommandos wird nicht im Detail beschrieben, lässt sich aber ungefähr anhand der Namen erkennen.

Listing 13.3 zeigt, wie ein JSON-kodiertes Kommando über den WebSocket-Kanal an den Server geschickt werden kann, um die Größe und Position eines Elements im Dokument zu verändern. Zunächst wird in Zeile 3 bis Zeile 5 die WebSocket-Verbindung über das Phoenix-Channel-Protokoll aufgebaut. Zeile 6 wählt sich über diese WebSocket-Verbindung in den Kommunikationskanal des Dokuments ein, welches bearbeitet werden soll. Zeile 7 bis 9 empfängt den aktuellen und zukünftige Dokumenteninhalte. Zeile 10 bis 19 senden das `update_box_size` Kommando mit den zugehörigen Parametern.

```
1  import { Socket } from "phoenix";
2  import { LiveState } from "./livestate";
3
4  const socket = new Socket("https://api.webeditor.example/socket", {
5    params: { token: cookie.auth_token }
6  });
7
8  const livestate = new LiveState(socket, { topic: "document:f3b690be-a9f3" });
9  livestate.subscribe((currentDocument) => {
10     console.log(currentDocument)
11 })
12
13 livestate.sendAction({
14     command: "update_box_size",
15     params: {
16     "layer_id": "a2bc416b-930b",
17     "value": {
18       "x": -6,
19             "y": -115,
20       "width": 50,
21             "height": 50
22     }
23     }
24 })
```

Quelltext 13.3: Beispiel eines über die WebSocket-Schnittstelle gesendeten Kommandos zur Änderung der Größe eines Elements

Befehl	Parameter
change_edge_attrs	layer_id, attrs
change_layer_shape	layer_id, shape_id
change_style	layer_id, style_key, style_value
change_text_body	layer_id, body
create_layer	points
create_layer	x, y, shape_id
create_layer	x, y, body
create_waypoint	layer_id, x, y
remote_cursor	x, y
remote_select	layer_id
delete_layer	layer_id
delete_waypoint	layer_id, waypoint_id
fetch_relative	layer_id, rel
insert_document	x, y, doc_id,
make_space	basex, basey, dirx, diry
move_layer	layer_id, dirx, diry
reorder	layer_id, rel
restore	version_id
set_meta	attributes
set_semantic_tag	layer_id, tag
set_interface	layer_id, schema_id
set_visibility	layer_id, visible
update_box_size	layer_id, x, y, width, height
update_text_pos	layer_id, x, y
update_waypoint_pos	layer_id, waypoint_id, x, y

Tabelle 13.2.: Befehle und deren Parameter zum Bearbeiten des Dokumenteninhalts über die WebSocket-Schnittstelle

13.3. JSON-Patch

JSON-Patch ist ein standardisiertes Datenformat zur Beschreibung von Unterschieden zwischen zwei JSON-Strukturen [BN13]. In der WebSocket-Schnittstelle des Web-Editors wird es verwendet, um die Menge an zu übertragenden Daten zu verringern. Anstatt nach jeder Veränderung eines Dokuments den gesamten neuen Inhalt des Dokuments an jeden Client zu übertragen, kann die Differenz zur vorherigen Version des Dokuments berechnet werden. Diese Differenz ist erwartungsgemäß kleiner als der gesamte Inhalt des Dokuments. Die Differenz kann also JSON-Patch an einen Client übertragen werden. Der Client kann diesen Differenz-Patch auf seine vorhandene Kopie des Dokuments anwenden, um die aktuelle Version des Dokuments zu erhalten.

Listing 13.4 zeigt den JSON-Patch, der infolge des Kommandos aus Listing 13.3 vom Server an die Clients geschickt wird. Der Patch beschreibt die nötigen Änderungen an der im Client gehaltenen JSON-Struktur: Die Ausmaße des Dokuments müssen angepasst werden, die X- und Y-Positionen des geänderten Elements müssen ersetzt werden, und die Referenz auf die aktuelle Dokumentenversion muss ersetzt werden.

```
{
  "patch": [
    {
      "op": "replace",
      "path": "/viewbox/x",
      "value": -106
    },
    {
      "op": "replace",
      "path": "/viewbox/y",
      "value": -215
    },
    {
      "op": "replace",
      "path": "/snapshot/prev_id",
      "value": "db70199d-3317-4681-bc8c-10ec530c49b5"
    },
    {
      "op": "replace",
      "path": "/snapshot/current_id",
      "value": "e037fc97-f0f9-4eb3-bb14-94e591ad47fa"
    },
    {
      "op": "replace",
      "path": "/layers/items/0/box/position_y",
      "value": -115
    },
    {
      "op": "replace",
      "path": "/layers/items/0/box/position_x",
      "value": -6
    }
  ]
}
```

Quelltext 13.4: Beispiel eines JSON-Patches zur Aktualisierung des Dokumenteninhalts

Teil IV.

Umsetzung des Editor-Clients

14. Kommunikation mit Editor-Server

In den bisherigen Kapiteln wurde die Umsetzung der Server-Komponente des Web-Editors behandelt. Dieses und folgende Kapitel bis Kapitel 19 „Kollaboratives Arbeiten“ widmen sich der Umsetzung der Client-Komponente, also der grafischen Benutzeroberfläche. Dafür wird in diesem Kapitel zunächst die Kommunikation zwischen der Client- und der Server-Komponente vorgestellt. Die Kommunikation findet über die vom Server bereitgestellten Schnittstellen per HTTP und WebSocket statt.

14.1. HTTP-Schnittstelle

Über die vom Server bereitgestellte HTTP-Schnittstelle kann der Client sämtliche Informationen über Dokumente und deren Inhalt abfragen. Der in JavaScript programmierte Client verwendet hierfür die `fetch` Funktion aus der JavaScript-Laufzeitumgebung.

Entsprechend der Hypertext-Konvention des HTTP Protokolls und des REST Paradigmas ist die vom Server bereitgestellte Schnittstelle so entworfen, dass die meisten zur Kommunikation notwendigen URLs mit den Antworten auf vorherige HTTP-Anfragen enthalten sind. Somit kann der Client ausgehend von einer Start-URL durch die vom Server gebotene Schnittstelle navigieren, ohne die Menge aller URLs fest einprogrammiert zu haben. Dies erlaubt auch die spätere Veränderung der Schnittstelle am Server, ohne dass der Client angepasst werden muss.

Für die statische Darstellung von Dokumenten genügt der lesende Zugriff über die HTTP-Schnittstelle.

14.2. WebSocket-Schnittstelle

Um auf Änderungen an Dokumenten reagieren zu können, muss der Client auch eine WebSocket-Verbindung zum Server aufbauen. Für die serverseitige Programmierung der WebSocket-Schnittstelle wurde die Elixir Phoenix Channel Programmbibliothek verwendet. Diese enthält neben der Implementierung in Elixir auch eine Implementierung des Channel-Protokols in JavaScript. Der Phoenix JavaScript Adapter wird im Client des Web-Editors verwendet, um eine Verbindung zum Server aufzubauen.

Auf Basis des Phoenix Channel Protokolls können WebSocket-Verbindungen in einzelne Kommunikationskanäle aufgeteilt werden. Die vom Client hergestellte Verbindung kann dadurch einem bestimmten Dokument zugeordnet werden, für dessen Änderungen sich der Client interessiert. Es können auch mehrere Kanäle über eine WebSocket-Verbindung gemultiplext werden, sodass ein Client sich für Änderungsinformationen an mehreren Dokumenten registrieren kann.

Zu jedem Dokument gehört also ein Kommunikationskanal. Der Name des zugehörigen Kanals kann für ein Dokument über die HTTP-Schnittstelle abgefragt werden. Der Client muss also zuerst ein Dokument über die HTTP-Schnittstelle abrufen und kann dann eine WebSocket-Verbindung zu dem zugehörigen Kanal aufbauen.

Über diesen Kanal kann der Client dann die am Dokument durchgeführten Änderungen in Form von JSON-Patch-Anweisungen empfangen. Durch Auswertung der JSON-Patch-Anweisungen bringt der Client seine lokale Version des Dokuments auf den neuesten Stand. Der Client soll aber keine eigenen Änderungen an seiner lokalen Kopie des Dokuments vornehmen. Die vom Server empfangenen Patches können ansonsten nicht korrekt umgesetzt werden, weil der lokale Zustand des Dokuments von dem erwarteten Zustand abweicht.

14.3. JSON

Der Server stellt alle Daten sowohl über die HTTP- als auch über die WebSocket-Schnittstelle im JSON-Format bereit. Für den in JavaScript programmierten Client ist es entsprechend sehr einfach, diese Daten zu lesen und in JavaScript-Datentypen zu übersetzen.

Das JSON-Format ist aber auch in vielen anderen Programmiersprachen implementiert. Somit lassen sich auch alternative Clients entwickeln, die sich mit dem Editor-Server verbinden. Die exakte Struktur der vom Server bereitgestellten Daten wird im Rahmen dieser Arbeit nicht genau spezifiziert, sondern ergibt sich aus der Implementierung. Diese Arbeit ist in dieser Hinsicht nur als Machbarkeitsnachweis der verwendeten Architektur zu betrachten.

14.4. Freie Wahl des Servers

Durch die HTTP-Schnittstelle sind die Client- und Serverkomponenten zu stark voneinander entkoppelt, dass Benutzer:innen bei Verwendung des Clients eigenständig zur Laufzeit einen Server wählen können, mit dem sich der Client verbinden soll.

Abb. 14.1 zeigt einen Screenshot von der Benutzeroberfläche beim Start des Editor-Clients. Über das Server-Eingabefeld kann der HTTP-Server angegeben werden, der als Editor-Server verwendet werden soll.

Abb. 14.2 zeigt die Benutzeroberfläche nach einem erfolgreichen Login. Es wird angezeigt, zu welchem Endpunkt der Client aktuell verbunden ist und welches Benutzerkonto für die Verbindung verwendet wurde. Über die Logout-Schaltfläche lässt sich die Verbindung wieder trennen.

Solange ein Server die in dieser Arbeit entwickelte HTTP- und WebSocket-Schnittstelle erfüllt, könnte er als alternativer Endpunkt vom Client verwendet werden. In der Praxis müsste die Schnittstelle dafür natürlich noch ausführlicher spezifiziert werden.

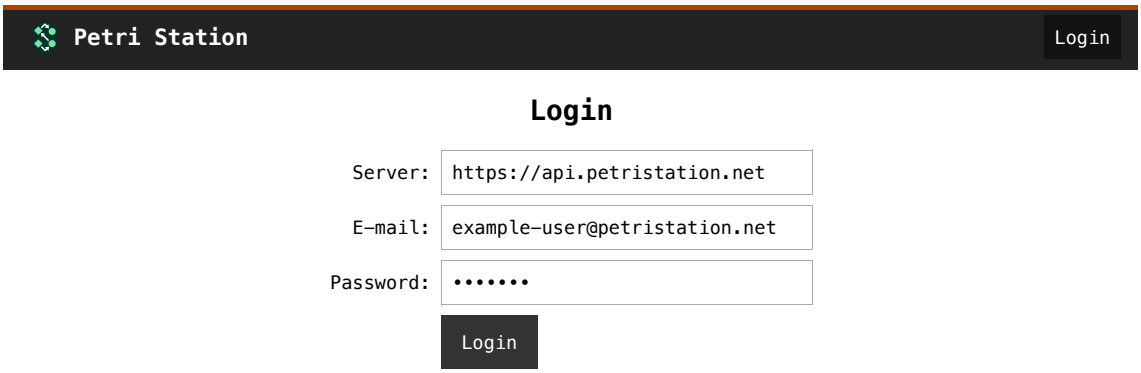


Abbildung 14.1.: Screenshot der Login-Eingabemaske des Web-Editors.

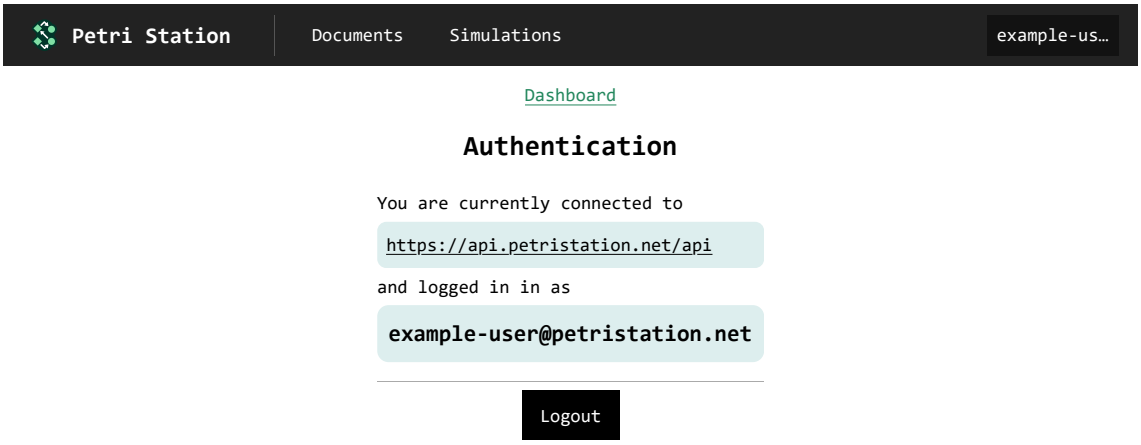


Abbildung 14.2.: Screenshot der Logout-Eingabemaske des Web-Editors.

15. Grafische Darstellung

Dieses Kapitel beschreibt die grafische Darstellung von Dokumenten. In diesem Kapitel wird davon ausgegangen, dass der aktuelle Zustand eines Dokuments über die zuvor beschriebene Schnittstelle vom Server empfangen wurde. Für die grafische Darstellung muss die in JSON kodierte Struktur des Dokuments interpretiert und in eine Grafik übersetzt werden.

Als Machbarkeitsnachweis wurde dies im Rahmen dieser Arbeit auf zwei unterschiedlichen Arten umgesetzt. Zum einen wird eine grafische Darstellung der Dokumente als Rastergrafik mittels WebGL erzeugt und zum anderen wird eine Vektorgrafik-basierte Darstellung mittels SVG umgesetzt.

In Renew ist die aus Figure-Elementen zusammengesetzte Dokumentstruktur sehr stark an das zur Darstellung verwendete Java AWT gekoppelt. Daher ist es für den Web-Editor dieser Arbeit besonders interessant, eine lose Kopplung zu erreichen und zu demonstrieren.

15.1. Darstellung als Rastergrafik per WebGL

Zunächst wurde eine Darstellung von Dokumenten mittels WebGL-Schnittstelle entwickelt. Mittels WebGL können Darstellungsberechnungen an die Grafikkarte eines Geräts ausgelagert werden. Dadurch können aufwändige Grafikberechnungen sehr performant und nebenläufig durchgeführt werden.

Falls besonders komplexe Dokumente sich nicht effizient mittels anderer Grafikschnittstellen im Webbrowser darstellen lassen, bietet WebGL einen großen Spielraum an Optimierungsmöglichkeiten. Daher ist es schon im Vorfeld interessant, die Darstellung mittels WebGL auf ihre prinzipielle Machbarkeit zu untersuchen.

WebGL ist aber auch eine sehr komplexe Schnittstelle. Daher existieren einige Programmbibliotheken, die als Abstraktionsebene auf WebGL aufbauen und vereinfachte Schnittstellen bereitstellen. Eine dieser Bibliotheken ist ThreeJS [D+13]. Im begrenzten Zeitrahmen dieser Arbeit wurde ThreeJS verwendet, um die Darstellung von Dokumenten im Webbrowser mittels WebGL umzusetzen.

Hierfür wurde eine Abbildung definiert, die den Inhalt eines Editor-Dokuments in ThreeJS-Grafikobjekte übersetzt. Es wird also jedem Element eines Dokuments ein zugehöriges ThreeJS-Objekt zugeordnet. ThreeJS stellt Objekte zur Darstellung von Rechtecken, Ellipsen, Text und Linien bereit.

Abb. 15.1 zeigt einen Screenshot eines durch WebGL dargestellten Dokuments. Zu erkennen sind die korrekt positionierten und eingefärbten Knoten und Kanten, sowie die einfache Darstellung von Textanschriften. Pfeilspitzen und komplexere geometrische Formen sind in der WebGL-Darstellung nicht enthalten.

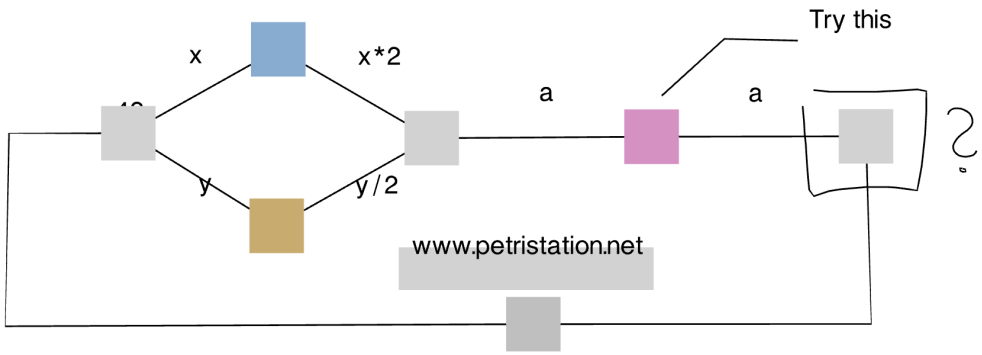


Abbildung 15.1.: Screenshot der experimentelle Darstellung eines Dokuments mittels WebGL

Für die vereinfachte Darstellung von Dokumenten ist so mit sehr wenig Aufwand gelungen. Für die Darstellung von formatiertem Text und den in Kapitel 7 „Parametrische Grafiksymbole“ behandelten Grafiksymbolen ist aber weitaus mehr Arbeit nötig. Das liegt daran, dass WebGL keine eigene Darstellung von Vektorgrafiken unterstützt. Jegliche Darstellung von komplexen geometrischen Formen und Text muss in die von WebGL erwarteten Polygonnetze übersetzt werden.

Auch wenn die einfache Darstellung von Dokumenten grundsätzlich demonstriert werden konnte, wurde die Verwendung von WebGL daher in dieser Arbeit nicht weiter verfolgt.

15.2. Darstellung als Vektorgrafik per SVG

Stattdessen wurde die Darstellung von Dokumenten als Vektorgrafik untersucht. Mittels SVG lassen sich Vektorgrafiken konstruieren und ein vom Webbrowser dargestelltes HTML-Dokument einbetten. Über die DOM-Schnittstelle des SVG-Formats kann per JavaScript auf die einzelnen Bestandteile einer Grafik zugegriffen werden, um diese zur Laufzeit zu manipulieren. Es können auch neue SVG-Dokumente per JavaScript erzeugt werden.

Für den Editor-Client bedeutet es, dass eine Funktion zur Übersetzung der JSON-Struktur eines Editor-Dokuments in eine SVG-DOM-Struktur übersetzt werden muss. Diese Übersetzung wird dadurch vergleichsweise leicht gemacht, dass im SVG-Format all die Bestandteile, die zur Darstellung von Renew-Figuren nötig sind, unterstützt werden. Bézierkurven, formatierter Text, Ellipse und Rechtecke können im SVG Format zu einer Grafik zusammengesetzt und im Webbrowser dargestellt werden.

Listing 15.1 zeigt die Definition einer Svelte-Komponente, die ein vom Server geladenes Dokument in ein SVG-Objekt umwandelt. Die einzelnen Ebenen, aus denen das

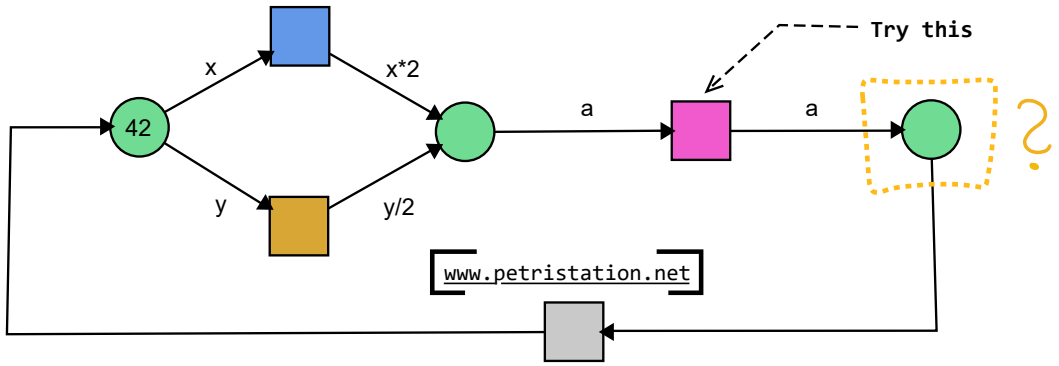


Abbildung 15.2.: Screenshot der Darstellung eines Dokuments mittels SVG

Dokument besteht, werden jeweils auf ein `<rect>`-, `<text>`- oder `<polyline>`-Element abgebildet. Der hier gezeigte Quelltext stellt das Dokument nur in vereinfachter Form dar. Für die vollständige Darstellung sind einige weitere Iterationen und Fallunterscheidungen nötig. Diese wurden im praktischen Teil dieser Arbeit vollständig umgesetzt.

Damit ist es im Rahmen dieser Arbeit gelungen, einen JavaScript-Client zu entwickeln, der im Webbrowser ausgeführt wird, Dokumente vom Server im JSON-Format empfängt, diese in ein SVG-Objekt übersetzt und dieses im Browserfenster darstellt.

In Abb. 15.2 ist die Darstellung eines Dokuments als SVG zu sehen. Es ist das gleiche Dokument, welches in Abb. 15.1 per WebGL dargestellt wurde. In der SVG-Darstellung ist nun zu erkennen, dass Pfeilspitzen, kreisförmige Knoten, formatierter Text und gestrichelte Linien korrekt dargestellt werden.

Neben der reinen Darstellung von statischen Grafiken bietet das SVG zusammen mit dem DOM auch noch einige weitere Funktionen [DGL+11]. Beispielsweise können auch animierte Grafiken definiert werden. Per CSS kann die Darstellung einzelner Elemente einer Grafik nachträglich verändert werden. Damit ist es Anwendern beispielsweise möglich, die lokale Darstellung von Dokumenten im Web-Editor individuell anzupassen und zu überschreiben.

```
<svg preserveAspectRatio="xMidYMid meet">
  {#each currentDocument.layers.items as el}
    {#if el.box && !el.hidden}
      {@const box = el?.box}

      <rect
        fill={el?.style?.background_color ?? 'green'}
        x={box.position_x}
        y={box.position_y}
        width={box.width}
        height={box.height}
      ></rect>
    {/if}
    {#if el.text && !el.hidden}
      {@const text = el?.text}
      {@const textStyle = el?.text?.style}
      <text
        fill={textStyle?.text_color ?? 'black'}
        x={text.position_x}
        y={text.position_y}
        font-size={textStyle?.font_size || 12}
      >
        {#each text.body.split('\n') as line, li}
          <tspan x={text.position_x} dy={textStyle?.font_size || 12}>
            {line}
          </tspan>
        {/each}
      </text>
    {/if}
    {#if el.edge && !el.hidden}
      <polyline
        points="{el.edge.source_x} {el.edge.source_y} {el.edge.waypoints
          .map((w) => `w.x {w.y}`)
          .join(' ')} {el.edge.target_x} {el.edge.target_y}"
        stroke="black"
        fill="none"
        stroke-width="2"
      />
    {/if}
  {/each}
</svg>
```

Quelltext 15.1: vereinfachte Svelte Komponente zur Darstellung eiens Dokuments als SVG

16. Reaktives Datenmodell

Dieses Kapitel behandelt die dynamische Darstellung von veränderlichen Dokumenten, im Speziellen das Beobachter-Muster, Signale und deren Kompositionen. Bisher wurde für die Darstellung von Dokumenten davon ausgegangen, dass der Inhalt des darzustellenden Dokuments vom Server empfangen wurde und dann in eine grafische Repräsentation übersetzt wird.

In Kapitel 14 „Kommunikation mit Editor-Server“ wurde aber schon erklärt, dass der Client über die WebSocket-Verbindungen auch Änderungen am Dokument empfangen kann. Das dargestellte Dokument ist also nicht statisch, sondern dynamisch. Verändert sich der Inhalt des Dokuments, muss auch die grafische Darstellung, also die berechnete SVG aktualisiert werden.

16.1. Beobachter-Muster

In der objektorientierten Programmierung ist das Beobachter-Muster ein Verhaltensmuster, um in einem System auf Änderungen zu reagieren [GHJV95]. Im Falle des Web-Editors muss der Client auf die vom Server mitgeteilte Änderung reagieren. Mittels Beobachter-Muster kann aber nur auf eine feste Menge an erwarteten Änderungen reagiert werden. Für jedes Objekt, von dem Änderungen erwartet werden, muss ein zugehöriger Beobachter angemeldet und gegebenenfalls wieder abgemeldet werden.

Im Kontext des Web-Editors können Änderungen in einem Dokument in sehr unterschiedlicher Granularität auftreten. Der Name eines Dokuments kann sich ändern. Ein Element kann in ein Dokument eingefügt werden. Elemente aus einem Dokument können entfernt werden. Viele verschiedene Änderungen können in einem Bearbeitungsschritt passieren.

Der Client bekommt jede Änderung vom Server per JSON-Patch zugeteilt. Abhängig davon, in welcher Granularität das Beobachter-Muster umgesetzt wird, um auf Änderungen zu reagieren, muss der Client für jede Änderung entweder die gesamte SVG von Grund auf neu generieren oder nur die kleinste nötige Menge an Änderungen an der zuvor erzeugten SVG durchführen. Idealerweise würde der Client jeden vom Server empfangenen Patch präzise in eine Änderung an der Vektorgrafik übersetzen. Soll dies mittels Beobachter-Muster umgesetzt werden, wären aber sehr viele feingranulare Beobachter-Objekte nötig, die verwaltet werden müssen.

16.2. Signale

Signale sind eine Datenstruktur zur reaktiven Programmierung, die die Menge von aktiven Beobachtern zur Laufzeit automatisch verwaltet. Beim lesenden Zugriff auf einen

Signalwert wird automatisch ein Beobachter registriert, der den gelesenen Wert im Falle einer Änderung automatisch aktualisiert. Nicht länger benötigte Beobachter werden automatisch wieder abgemeldet [Sta24, Pil24, CC13].

Mittels Signalen ist es möglich, feingranulare und komplexe Abhängigkeitsbeziehungen zwischen Berechnungen zu erzeugen. Im Falle der Änderung eines Wertes wird dieser präzise durch das System propagiert, um alle davon abhängigen Werte neu zu berechnen.

JavaScript-Bibliotheken wie SolidJS oder Svelte verwenden ein Signalsystem zur Konstruktion eines global konsistenten Zustands der Benutzeroberfläche. Im Rahmen dieser Arbeit wird Svelte für die Programmierung der Benutzeroberfläche eingesetzt. Abgesehen von der Signalsemantik bietet Svelte, genau wie vergleichbare Bibliotheken, auch eine deklarative Schnittstelle zur Konstruktion von HTML und SVG bzw. im Allgemeinen XML-Dokumenten. Daher kann Svelte verwendet werden, um reaktive Benutzeroberflächen, also reaktive HTML-Dokumente und reaktive SVG-Grafiken zu konstruieren.

Damit ist der Client unter Einsatz von Signalen in der Lage, die vom Server empfangenen Änderungen in eine möglichst kleine Sequenz an Änderungen an der dargestellten Vektorgrafik zu übersetzen.

16.3. Komposition von Signalen

Die CalmmJS-Architektur ist ein Konzept, welches die Signale um einen bidirektionalen Datenfluss erweitert, um kompositionierbare Benutzeroberflächen zu konstruieren [Kar16a]. Dabei wird auf einige Entwurfsmuster aus der funktionalen Programmierung zurückgegriffen. Im Zentrum steht hierbei das Konzept der Optiks und Linsen. Diese Entwurfsmuster im Detail zu behandeln sprengt den Rahmen dieser Arbeit. Stattdessen sei hiermit auf die ausführliche Dokumentation der CalmmJS-Architektur verwiesen.

Zusammenfassend lässt sich sagen, dass die CalmmJS-Architektur in der Entwicklung der Client-Komponente umgesetzt wurde, um die globale Konsistenz aller Komponenten der Benutzeroberfläche zu erzielen und gleichzeitig allen einzelnen Komponenten lokale Schnittstellen für nötige Schreibzugriffe bereitzustellen.

17. Kamera-Navigation

Die grafische Benutzeroberfläche soll die nötigen Werkzeuge anbieten, um an Dokumente in ihrer Darstellungsgröße zu skalieren und für ein Dokument, dessen Darstellung über die Grenzen der Zeichenfläche herausragt, die Position des Sichtbereichs frei zu wählen. Benutzer:innen sollen also die metaphorische Kamera, durch die auf ein Dokument geschaut wird, frei navigieren können. Dieses Kapitel behandelt die Modellierung dieser interaktiven Kamera in der Client-Komponente.

17.1. Koordinatensysteme

Das Konzept der Kamera lässt sich am besten in Form von verschiedenen zweidimensionalen Koordinatensystemen verstehen. Das Welt-Koordinatensystem ist das, in welchem sich die Elemente eines Dokuments befinden. Ist ein gezeichnetes Rechteck beispielsweise an Position $(50, 100)$ positioniert, ist damit die Position im Dokument, also in Weltkoordinaten gemeint. Das Bildschirm-Koordinatensystem hingegen bezieht sich auf die Zeilen und Spalten, in denen die Pixel eines Bildschirms angeordnet sind. Position $(0, 0)$ bezieht sich etwas auf den Pixel in der oberen linken Ecke eines Bildschirms.

Eine von Benutzerinnen und Benutzern steuerbare Kamera entspricht einer durch Eingaben veränderbaren Abbildung des einen Koordinatensystems auf das andere Koordinatensystem. Rechnerisch entspricht das einer linearen Transformation. Metaphorisch lässt sich aber ausdrücken, wo die Kamera sich im Weltkoordinatensystem befindet und wie groß ihr Blickfeld ist. Rechnerisch entspricht das den Eigenwerten und Eigenvektoren der linearen Abbildung. Metaphorisch ist gemeint, wie eine Aufnahme des Dokuments aussehen würde, wenn eine Kamera mit einem bestimmten Blickfeld an einem Punkt im Dokument platziert und auf das Dokument gerichtet wird. Im Folgenden wird sich auf die metaphorische Intuition gestützt.

In Abb. 17.1 sind vier verschiedene Koordinatensysteme dargestellt, die verwendet werden können, um Inhalte eines Dokuments auf Pixelpositionen des Bildschirms abzubilden. Die Abbildung von Fenster- auf Bildschirmkoordinaten übernimmt für gewöhnlich die Fensterverwaltung vom Betriebssystem. Für die Steuerung der Kamera im Editor ist vor allem die Abbildung zwischen Weltkoordinaten und Sichtbereich von Bedeutung.

17.2. Problem der Kamerasteuerung in Renew

Die Umsetzung der Kamera ist für den Web-Editor besonders interessant, weil die Kameranavigation in Renew einige Mängel aufweist. Diese sollen zunächst untersucht werden.

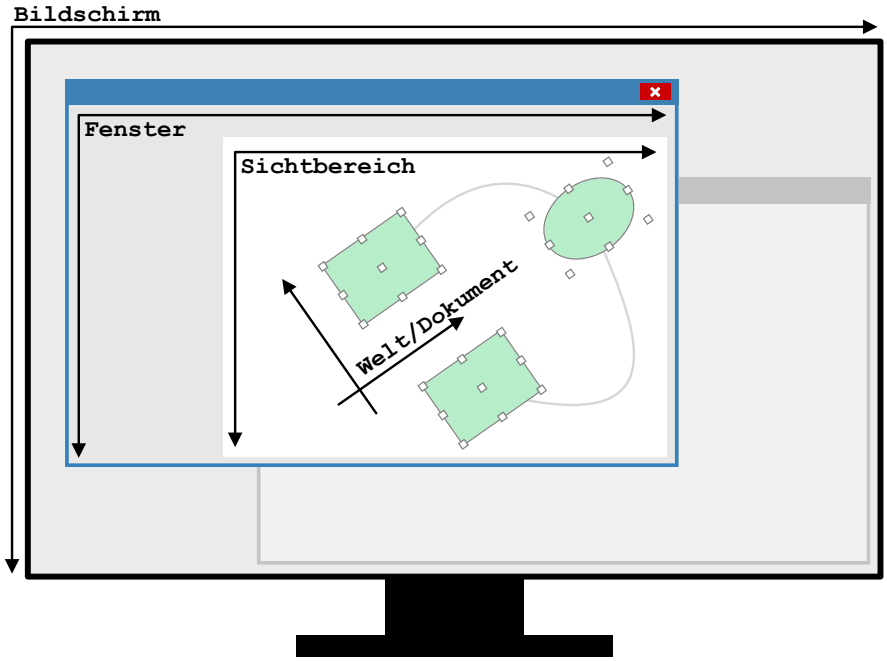


Abbildung 17.1.: Koordinatensysteme zur Positionieren von grafischen Elementen auf einem Gerätebildschirm

17.2.1. Kamera ist auf positiven Quadranten beschränkt

Ein in Renew sehr auffälliges Verhalten ist, dass die Kamera sich nicht in den negativen Bereich der X- und Y-Achse bewegen lässt. Der Nullpunkt des Dokuments befindet sich initial in der oberen linken Ecke der Zeichenfläche. Durch die Verwendung der Scrollbalken kann die Kamera zwar nach rechts unten navigiert werden, aber nicht weiter nach links oben als ihre Ausgangsposition. Elemente des Dokuments lassen sich aber nach links oben in den negativen Bereich des Weltkoordinatensystems schieben. Diese Elemente befinden sich dann aber auf Dauer außerhalb des durch die Steuerung erreichbaren Sichtbereichs.

17.2.2. Pivotpunkt beim Zoomen

Diese Einschränkung der Kamerabewegung hat einige weitere Auswirkungen. Renew erlaubt es, unter Verwendung des Mausrades heran und heraus zu zoomen. Typischerweise ist die Zoomfunktion in gängigen Grafikanwendungen so umgesetzt, dass die Position des Mauszeigers als Pivotpunkt der Skalierung verwendet wird. Befindet sich ein Element des Dokuments unter dem Mauszeiger, sollte es sich auch nach dem Heran- oder Herauszoomen weiterhin am Mauszeiger befinden. Renew versucht dieses Verhalten auch umzusetzen, allerdings scheitert es daran, dass der nötige Freiheitsgrad der Kamera nicht gegeben ist. Je nachdem, wo der Mauszeiger im Dokument platziert ist, müsste sich die Kamera beim Zoomen in den negativen Bereich des Koordinatensystems bewegen. In Renew springt die Ansicht dann aber in eine unerwartete Position.

17.2.3. Skalierung von interaktiven Elemente

Interaktive Elemente auf der Zeichenfläche, beispielsweise Greifer zum Skalieren eines ausgewählten Elements, werden in Renew zusammen mit dem Zoomfaktor der Kamera skaliert. Wird also sehr weit herausgezoomt, um einen größeren Ausschnitt des Dokuments zu sehen, werden die interaktiven Elemente nicht sehr klein dargestellt und sind nicht mehr gut zu erkennen und nicht anklickbar.

17.2.4. Ausmaße des Dokuments

Renew passt die Ausmaße eines Dokuments automatisch die Position der enthaltenen Elemente an. Wird ein Element sehr weit entfernt vom Nullpunkt platziert, wird das Dokument zusammen mit dem Bewegungsfreiraum der Kamera entsprechend vergrößert. Die Ausmaße werden aber nicht wieder reduziert, wenn Elemente entfernt werden. Dadurch kann die Navigation in einem versehentlich vergrößerten Dokument unübersichtlich werden.

17.2.5. Gemeinsame Ursache

All das beschriebene Verhalten in Renew ist auf die Umsetzung der Dokumentendarstellung mittels Java AWT zurückzuführen. Nämlich ist die Navigation der Kamera in Renew gar nicht als eigenständiges Kameramodell umgesetzt, sondern basiert einfach auf dem Verhalten der Klasse, die für die Darstellung von Scrollbalken verwendet wird. AWT ist aber gar nicht für die Umsetzung von interaktiven Zeichenflächen mit dynamischer Kamera ausgelegt.

Die effektive Position der Renew Kamera ergibt sich somit aus der Kombination mehrerer interner Zustände und dem Verhalten verschiedener AWT-Komponenten. Für die Steuerung der Kamera müssen diese Zustände koordiniert werden. Dies ist nur schwierig umsetzbar.

17.3. Geometrisches Modell für die Kamera

Um eine flüssige Navigation der Kamera im Web-Editor muss also ein passendes, einheitliches Modell entwickelt werden, welches sich nicht an inkompatiblem Verhalten von fremden GUI-Toolkits bedient.

Abb. 17.2 stellt die Maße, aus denen sich das Kameramodell zusammensetzt, grafisch dar. Die Position des Kamerafokus $(X^{\text{Cam}}_{\text{focus}}, Y^{\text{Cam}}_{\text{focus}})$, der Zoomfaktor S und der Rotationswinkel α können durch Benutzereingaben gesteuert werden. Daraus ergibt sich zusammen mit der Bildschirm- bzw. Fenstergröße der Sichtbereich. Dieser ist in der Grafik als das innerste Rechteck mit den hellen Eckpunkten dargestellt.

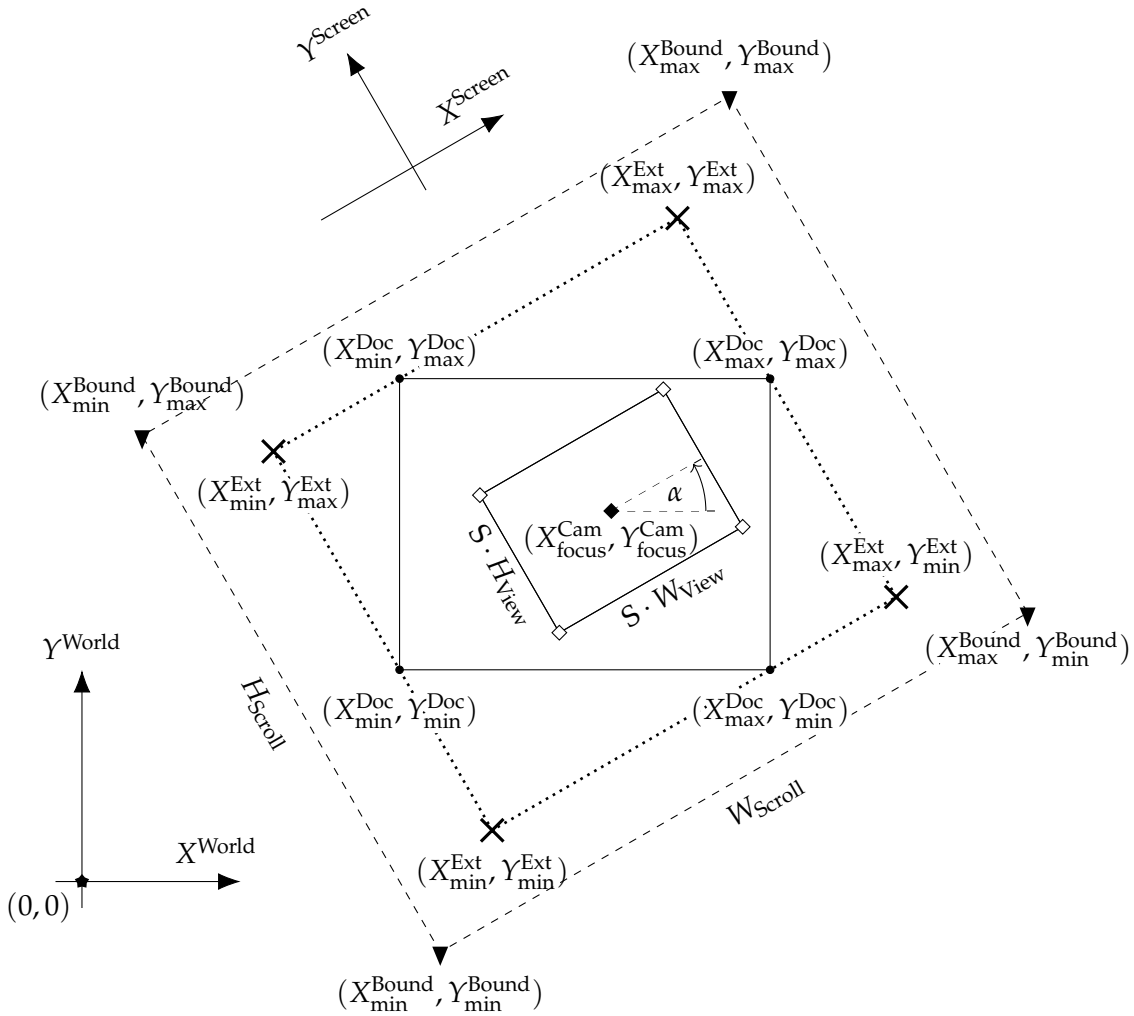


Abbildung 17.2.: Maße des Kameramodells im Weltkoordinatensystem. Die gefüllte Raute in der Mitte des innersten Rechtecks ist der Punkt, auf den der Mittelpunkt der Kamera fokussiert ist. Das innerste, um α rotierte Rechteck ist der durch die Kamera erfasste Sichtbereich. Der Sichtbereich hat die jeweils um S skalierte Breite W_{View} und Höhe H_{View} des Bildschirms. Dabei ist mit $S = \exp(Z^{\text{Cam}}_{\text{focus}})$ der effektive Zoomfaktor gemeint, sodass sich aus einem negativen $Z^{\text{Cam}}_{\text{focus}}$ ein kleiner Sichtbereich ergibt. Das zum Koordinatensystem parallele Rechteck mit den Kreisen auf den Eckpunkten (Doc) stellt die Ausmaße des Dokuments dar. Das gepunktete Rechteck mit den Kreuzen auf den Eckpunkten (Ext) ist das kleinste im Kamerawinkel gedrehte Rechteck, von dem das gesamte Dokument (Doc) umschlossen wird. Das äußerste, gestrichelte Rechteck mit den Dreiecken auf den Eckpunkten (Bound) ist mit einer Breite von W_{Scroll} auf beiden Seite um jeweils genau $\frac{W_{\text{View}}}{2}$ breiter und mit einer Höhe von H_{Scroll} auf beiden Seite um jeweils genau $\frac{H_{\text{View}}}{2}$ höher als das gepunktete Rechteck (Ext).

Das nächst äußere Rechteck (Doc) mit den kreisförmigen Ecken ist stets parallel zu den Koordinatenachsen ausgerichtet. Es stellt die Ausmaße des Dokuments dar und ergibt sich aus den im Dokument positionierten Elementen.

Das gepunktete Rechteck (Ext), dessen Ecken durch Kreuze markiert sind, ergibt sich aus der Kamerarotation (α) und den Ausmaßen des Dokuments. Es ist das kleinste zur Kamera parallele Rechteck, welches das gesamte Dokument umfasst.

Das äußerste gestrichelte Rechteck (Bounds), dessen Eckpunkte mit Dreiecken markiert sind, ergibt sich aus einer Vergrößerung des gepunkteten Rechtecks (Ext). Es ist in der Breite auf beiden Seiten um genau die Hälfte des Sichtbereichs breiter. In der Höhe ist es in beide Richtungen um genau die Hälfte des Sichtbereichs höher.

Solange sich der Mittelpunkt der Kamera innerhalb des äußersten Rechtecks (Bounds) befindet, befindet sich ein Teil des Dokuments im Sichtbereich. Liegt der Kameramittelpunkt aber exakt auf der Kontur des äußeren Rechtecks, ist der Dokumenteninhalt knapp nicht zu sehen, weil er sich genau außerhalb des Sichtbereichs befindet.

Diese Anordnung ermöglicht es, eine weit vom Dokument entfernt positionierte Kamera schlagartig zurück zum Dokument zu positionieren, ohne dass dies auf dem Bildschirm als sprunghafte Bewegung wahrgenommen wird.

17.4. Datenstruktur für die Kamera

Quellcode 17.1 zeigt die für diese Arbeit entwickelte Struktur. Die Struktur besteht aus drei Teilen. Der `focus` setzt sich mit `w`, `x`, `y` und `z` aus vier Gleitkommazahlen zusammen und gibt die aktuelle Rotation (`w`), Position (`x`, `y`) und die Zoomstufe (`z`) der Kamera an. Dies sind die vier Parameter, die direkt durch Benutzereingaben gesteuert werden.

Über die Breite (`width`) und Höhe (`height`) des `screen`-Feldes kann festgelegt werden, wie groß ein Ausschnitt des Dokuments bei einer initialen Zoomstufe von $z = 0$ sichtbar sein soll. Das entspricht den Maßen W_{View} und H_{View} in der Abb. 17.2. Diese Maße werden nicht durch Benutzereingabe gesteuert, sondern bei der Programmierung festgelegt.

Das `viewport`-Feld enthält ebenfalls Angaben zur Breite und Höhe der Darstellung. Diese beziehen sich aber auf die Größe des Fensters oder des Bildschirms, in dem der Web-Editor dargestellt wird. Sie werden automatisch mit der aktuellen Fenstergröße abgeglichen und lassen sich entsprechend durch Benutzereingaben beeinflussen.

Die `autoFit` Optionen legen entlang der `x`- und `y`-Achse fest, wie die Darstellung skaliert werden soll, wenn die in `screen` festgelegten Maße ein anderes Seitenverhältnis haben als der von `viewport` beschriebene Bildschirm. Das Verhalten entspricht dem des durch SVG definierten `preserveAspectRatio`-Attributs [BM20].

Auf Basis dieser einheitlichen Datenstruktur können alle Navigationsoperationen definiert werden. Außerdem sind in der Datenstruktur alle Informationen enthalten, die zur Darstellung eines Dokuments in der gewünschten Skalierung und Orientierung per SVG, WebGL oder einer alternativen Schnittstelle benötigt werden.

```
type Camera = {
    // Ausrichtung der Kamera im Weltkoordinatensystem
    focus: {
        // Rotation der Kamera
        w: float = 0,
        // X-Position des Blickmittelpunktes
        x: float = 0,
        // Y-Position des Blickmittelpunktes
        y: float = 0,
        // Zoomlevel, negativ zum heranzomen, positiv zum herauszoomen
        z: float = 0,
    },
    // Größe des angesrebten Blickfeldes
    // in Weltkoordinaten bei Zoomstufe 0
    screen: {
        // Breite des Blickfeldes
        width: float = 300,
        // Höhe des Blickfeldes
        height: float = 200,
    },
    // Tatsäche Größe der Darstellung auf dem Bildschirm
    viewport: {
        // Breite in Pixel
        width: float,
        // Höhe in Pixel
        height: float,
        // Optionen zu automatischen Anpassung
        // falls Maße des Blickfeldes nicht zur
        // Tatsächlichen Bildschirmgröße passen
        autoFit: {
            // verhält sich wie preserveAspectRatio in SVG
            x: "min" | "max" | "mid",
            y: "min" | "max" | "mid",
            slice: false,
        }
    }
}
```

Quelltext 17.1: Datenstruktur für die Modellierung des Kamerazustands

17.5. Navigationsoperationen

Die Bewegungsfreiheit der Kamera im zweidimensionalen Raum lässt sich in drei Bewegungsoperationen unterteilen.

17.5.1. Pan

Die Panoperation verschiebt die Position der Kamera, also auch das Blickfeld entlang der x- und y-Achse. Im Web-Editor wird der per Panoperation navigierbare Bereich anders als in Renew nicht durch die Größe des Dokuments eingeschränkt. Die Kamera lässt sich beliebig weit vom Koordinatenursprung entfernen. Damit werden die beiden in Renew auftretenden Probleme der unerreichbaren Elemente und des springenden Pivotpunktes vermieden.

17.5.2. Zoom

Die Zoomoperation verkleinert oder vergrößert den Sichtbereich der Kamera und wirkt sich somit auf die Darstellungsgröße des Dokuments auf dem Bildschirm aus. Eine Zoomoperation findet stets um einen gegebenen Pivotpunkt statt, damit die Navigation im Dokument kontinuierlich und nachvollziehbar ist.

17.5.3. Rotation

Die Rotation dreht den Blickwinkel der Kamera um die zur (x, y)-Ebene senkrecht stehende z-Achse. In Renew ist die Rotation der Kamera gar nicht möglich. Im Web-Editor ergibt sich die Umsetzung der Operation aber natürlich aus dem gewählten Datenmodell und ist besonders für die Arbeit auf Endgeräten mit Touchscreen sinnvoll zu verwenden.

Genau wie der Zoom findet auch die Rotation um einen Pivotpunkt statt. Dies kann entweder die Zeigerposition oder die aktuelle Bildschirmmitte sein.

17.6. Eingabemodi

Der Web-Editor soll sich entsprechend der aufgestellten Anforderungen auf verschiedenen Endgeräten verwenden lassen. Besonders die Navigation der Kamera ist auf unterschiedlichen Geräten auf verschiedene Art und Weise möglich. In JavaScript stehen für die Verarbeitung verschiedener Eingabewerkzeuge wie Maus oder Fingergesten verschiedene Programmierschnittstellen zur Verfügung. Die `MouseEvent`-Schnittstelle kann sowohl Maus- als auch einfache Touchscreen-Eingaben verarbeiten. Die `TouchEvent`-Schnittstelle ist auf die Verarbeitung von auch komplexen Touchscreen-Eingaben ausgelegt. Die `GestureEvent`-Schnittstelle steht nur auf iOS zur Verfügung und kann eine Menge von vordefinierten Gesten verarbeiten. Die `PointerEvent`-Schnittstelle ist eine vereinheitlichte Schnittstelle zum Verarbeiten von Maus-, Finger- und Stifteingaben.

Für die Umsetzung der Kameranavigation wird eine fein abgestimmte Mischung aller Schnittstellen verwendet, um ein auf alle Endgeräte abgestimmtes Verhalten zu erzielen.

17.6.1. Scrollbalken

Auf Desktopgeräten ist es üblich, vertikale und horizontale Scrollbalken am Rand eines Arbeitsbereichs darzustellen, um die vertikale und horizontale Navigation zu ermöglichen.

Für die Darstellung von Scrollbalken ist es nötig, das Größenverhältnis zwischen sichtbarem und insgesamt navigierbarem Bereich angeben zu können. Dies ist nötig, damit die Größen der Scrollbalken proportional zu diesem Verhältnis dargestellt werden können.

Der navigierbare Bereich ist im Web-Editor in (x, y)-Richtung nicht eingeschränkt, sondern beliebig groß, um wie zuvor erklärt die nötige Bewegungsfreiheit für den Zoom und die Rotation mittels Pivotpunkt zu bieten. Für die Darstellung der Scrollbalken werden

aber die aktuellen Ausmaße des Dokumenteninhalts in Richtung der Kamerarotation verwendet. Bezogen auf Abb. 17.2 wird die Gesamtgröße des scrollbaren Bereichs mit den Maßen H_{Scroll} und W_{Scroll} angegeben. Die Länge des horizontalen Scrollbalkens ergibt sich entsprechend aus dem Verhältnis von $S \cdot W_{\text{View}}$ zu W_{Scroll}.

Über die Verwendung der Scrollbalken lässt sich der Kamerafokus also innerhalb des äußersten Rechtecks aus Abb. 17.2 in frei bewegen. Der horizontale Scrollbalken bewegt die Kamera entlang der X^{Screen}-Achse. Der vertikale Scrollbalken bewegt die Kamera entlang der Y^{Screen}-Achse.

Durch die anderen Arten der Steuerung kann der Fokus auch außerhalb dieser Rechteckbegrenzung navigiert werden. Sobald ein Scrollbalken aktiv zum Scrollen eingesetzt wird, erzwingt dieser die Kameraposition wieder in den durch das äußerste Rechteck begrenzten Bereich.

Diese Verwendung der Dokumentenausmaße für die Scrollbalken erlaubt es, trotz unbegrenztem Navigationsbereich schnell zur Mitte des Dokuments zurückzufinden.

Der Web-Editor verwendet für die Darstellung der Scrollbalken die nativen Scrollbalken des Browsers bzw. des zugrundeliegenden Betriebssystems. Viele andere webbasierte Grafikeditoren implementieren entweder gar keine Scrollbalken oder eine vereinfachte selbst-programmierte Variante, deren Verhalten von dem vertrauten Verhalten der nativen Scrollbalken abweicht. Grund hierfür ist vermutlich die Herausforderung, unter Verwendung der nativen Scrollbalken ein konsistentes Kameraverhalten zu erzielen. Im Rahmen dieser Arbeit ist dies für den Web-Editor aber gelungen.

Abb. 17.3 zeigt die Darstellung eines Dokuments im Web-Editor mit Scrollbalken am unteren und rechten Rand. Das Dokument ist mittig im Sichtbereich dargestellt und an der Position und Größe der Scrollbalken lässt sich erkennen, dass diese sich nach links, rechts, oben und unten frei bewegen lassen.

Im Gegensatz dazu sind in Abb. 17.4 die Scrollbalken in Renew zu sehen. Diese befinden sich bereits links bzw. oben am Anschlag, sodass die Kamera nur nach unten rechts navigiert werden kann.

Dank der Verwendung von nativen Scrollbalken des Betriebssystems funktionieren auch einige weitere Kurztasten für die Navigation der Kamera. Unter Windows kann mit gedrückter mittlerer Maustaste diagonal navigiert werden. Per Pfeiltasten, Ende-Taste und Bildauf- sowie Bildab-Tasten kann in die jeweilige Richtung navigiert werden. Unter MacOS kann das Multitouch-Trackpad zur Navigation verwendet werden, ohne dass die Erkennung entsprechender Gesten für die Kamera extra programmiert werden muss.

17.6.2. Mausgesten

Alle Navigationsoperationen sollen sich unter Verwendung einer Maus als Eingabegerät durchführen lassen. Die Bewegung der Maus bei gedrückter mittlerer Maustaste führt eine Panoperation in Bewegungsrichtung durch.

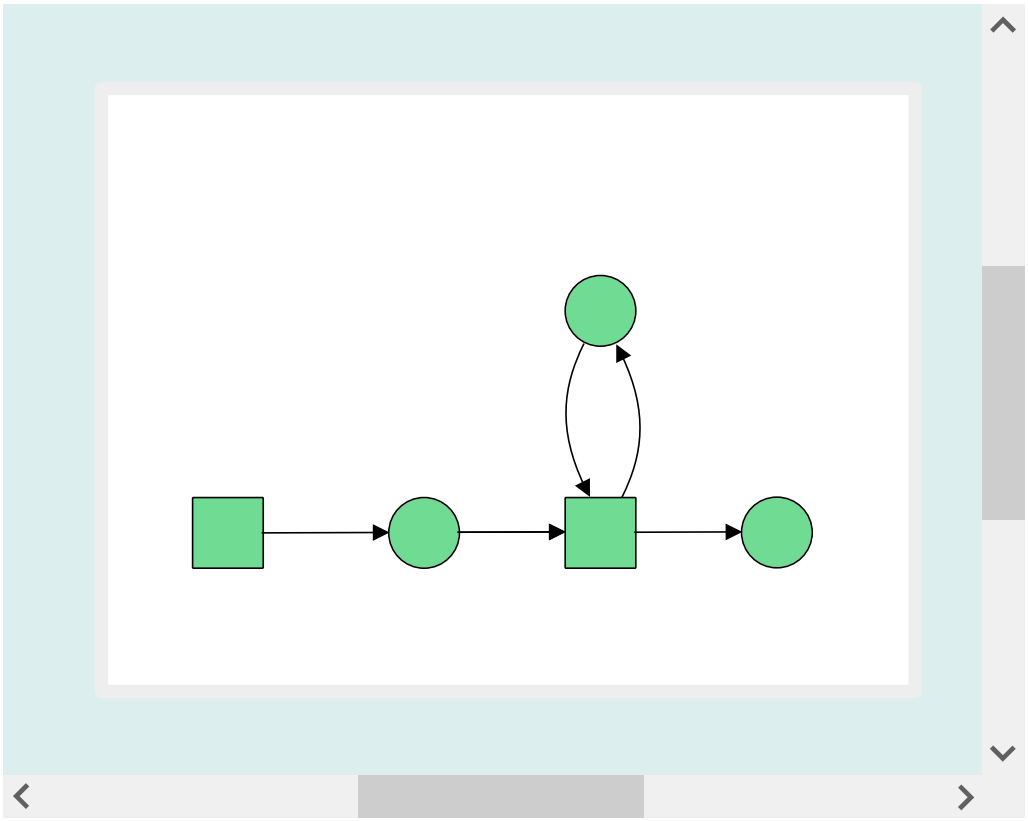


Abbildung 17.3.: Screenshot der zentrierten Scrollbalken nur Navigation des Kamera im Web-Editor

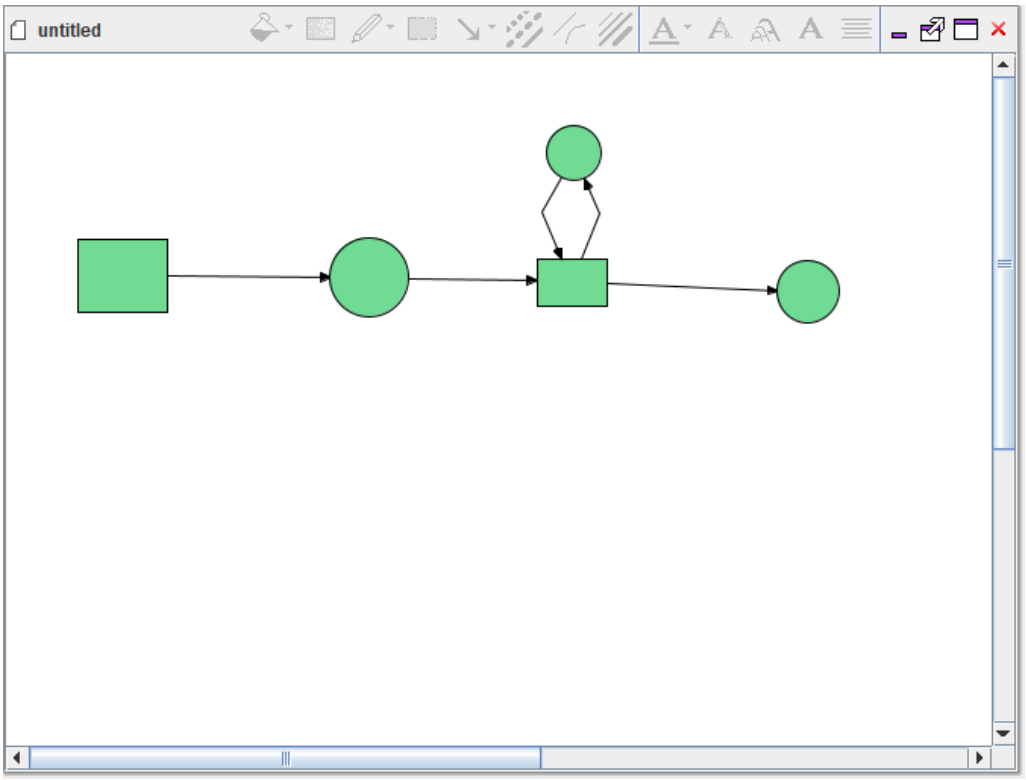


Abbildung 17.4.: Screenshot der durch AWT erzeugten Scrollbalken in Renew

Das Drehen des Mausrades ohne gedrückte Taste führt eine Panoperation in vertikale Blickrichtung durch. Dies entspricht dem vertikalen Scrollen in einer klassischen GUI-Komponente wie einer Liste. Das Drehen des Mausrads bei gedrückter ⇧-Taste führt eine Panoperation in horizontaler Blickrichtung durch. Dies entspricht dem Verhalten von Webbrowsern bei der Navigation eines horizontal scrollbaren Dokuments.

Das Drehen des Mausrades bei gedrückter Ctrl-Taste führt eine Zoomoperation durch. Als Pivotpunkt wird die Position des Mauszeigers im Dokument verwendet. Dies entspricht dem typischen Verhalten von Webbrowsern und anderen Anwendungen in der Darstellung von Dokumenten.

Das Drehen des Mausrades bei gedrückter Alt-Taste führt eine Rotationsoperation durch. Als Pivotpunkt wird die Position des Mauszeigers im Dokument verwendet.

17.6.3. Fingergesten

Auf Endgeräten mit Touchscreen lassen sich Scroll- und Zoomoperationen typischerweise per Fingergesten durchführen. Für das Auslösen von Fingergesten müssten mindestens zwei Finger den Bildschirm berühren.

Sowohl die relative Bewegung der Finger zueinander als auch die Bewegung des Mittelpunktes der Finger relativ zum Bildschirm lösen Navigationsoperationen aus. Damit aus den Fingerbewegungen verständliche Navigation entsteht, muss die Kamera so rotiert, bewegt und gezoomt werden, dass sich das dargestellte Dokument relativ zu den Fingerspitzen nicht verändert. Es muss also der Eindruck entstehen, dass der Benutzer oder die Benutzerin das Dokument fest im Griff der Finger hat und es seinen Bewegungen strikt folgt.

17.7. Zusätzliche Werkzeuge

Zusätzlich zu der Navigation mittels Maus- und Fingergesten kann die Kamera des Web-Editors auch mittels zusätzlicher Werkzeuge navigiert werden. Abb. 17.5 zeigt den Arbeitsablauf dieser Werkzeuge in jeweils zwei Schritten.

Ein Lupenwerkzeug erlaubt das Heranzoomen an einen durch ein Rechteck ausgewählten Ausschnitt des Dokuments. Abb. 17.5(a) zeigt das mit der Maus oder dem Finger aufgespannte Rechteck, welches dabei hell hervorgehoben wird. In Abb. 17.5(b) wird das Ergebnis nach dem Loslassen der Maus oder des Fingers gezeigt. Der gewählte Ausschnitt ist nun herangezoomt und füllt den Bildschirm.

Ein stufenloses Zoomwerkzeug erlaubt das Zoomen um einen Pivotpunkt mittels kreisenden Finger- oder Mausbewegungen. In Abb. 17.5(c) ist das Werkzeug als ein dunkler Ring mit einem langen blauen Hebel auf der Zeichenfläche zu erkennen. Der Ring wird per gedrückter Maus oder Fingerberührung auf der Zeichenfläche platziert. Der Hebel kann durch die Bewegung der Maus oder eines Fingers um den Mittelpunkt des Ringes rotiert werden. In Abb. 17.5(d) ist zu erkennen, wie sich der Drehwinkel auf den Kame-

razoom auswirkt. Der Hebel wurde um ein paar Grad gegen den Uhrzeigersinn gedreht und die Kamera entsprechend herangezoomt. Als Pivotpunkt für den Zoom wird der Mittelpunkt des Ringes verwendet. Der Radius des Hebels ergibt sich aus dem Abstand der Maus zum Mittelpunkt und wirkt sich als Multiplikator auf die Veränderung des Zooms aus.

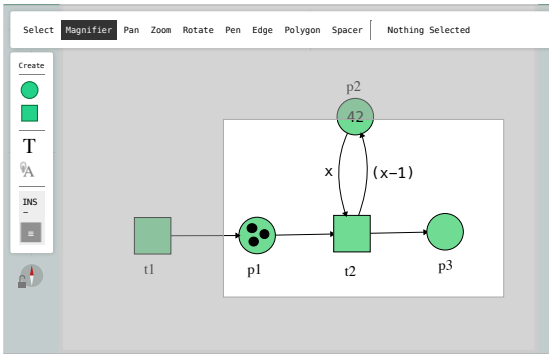
Ein stufenloses Rotationswerkzeug erlaubt die Rotation der Kameraperspektive mittels kreisender Finger- oder Mausbewegungen. In Abb. 17.5(e) ist zu erkennen, wie hierfür ebenfalls ein Ring auf der Zeichenfläche platziert wird. Durch eine Maus- oder Fingerbewegung außerhalb des Ringes wird die Kamera um dessen Mittelpunkt rotiert, wie in Abb. 17.5(f) zu erkennen ist.

Ein eigenständiges Panwerkzeug erlaubt das Verschieben der Kameraposition mit nur einem statt zwei Fingern und mittels linker statt mittlerer Maustaste. Die Rotationssperre erlaubt das Einfrieren des Kamerawinkels, um beim Verwenden von Fingergesten die Kamerarotation nicht unbeabsichtigt zu verändern.

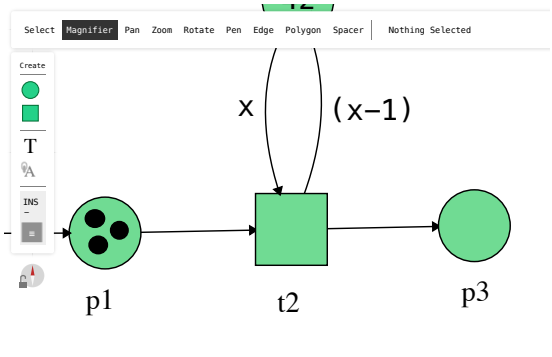
17.8. Minimap

Genau wie Renew bietet auch der Web-Editor eine Minimap, also eine Miniaturdarstellung des Dokuments. Damit ist es möglich, den gesamten Inhalt eines Dokuments stets aus der Vogelperspektive im Blick zu behalten. Innerhalb der Minimap wird auch die aktuelle Position der Kamera zusammen mit dem sich daraus ergebenden Blickfeld hervorgehoben. Das SVG Format erlaubt es, mehrere Referenzen auf zuvor definierte Elemente einer Grafik zu erzeugen. Diese Referenzen werden dann als Kopien der referenzierten Grafik mehrfach dargestellt [Com20]. Die Minimap im Web-Editor konnte als eine solche Referenz auf die primäre Darstellung des Dokuments umgesetzt werden.

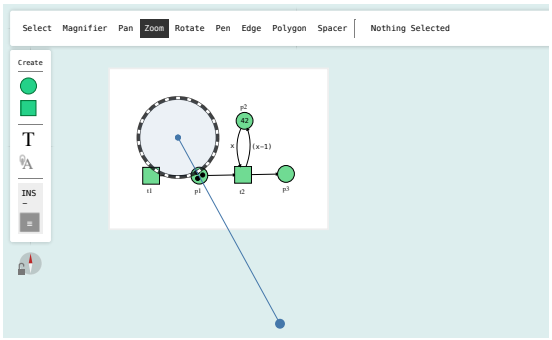
Abb. 17.6 zeigt die Benutzeroberfläche des Web-Editor-Clients mit der Minimap oben rechts in der Ecke. Die primäre und rotierte Darstellung des Dokuments ist im Hintergrund zu sehen. Die Minimap wird etwas transparent dargestellt. Es ist zu erkennen, wie das gesamte Dokument in der Minimap enthalten ist. Außerdem ist bei genauer Betrachtung ein graues rotiertes Rechteck mit einer orangen Pfeilspitze zu erkennen. Diesen stellen den sichtbaren Bereich und die Blickrichtung der Kamera in der Minimap dar.



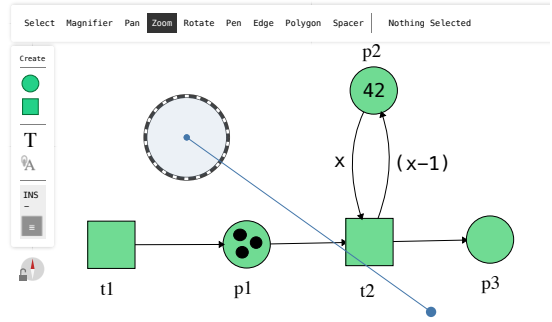
(a) Lupenzoom Start



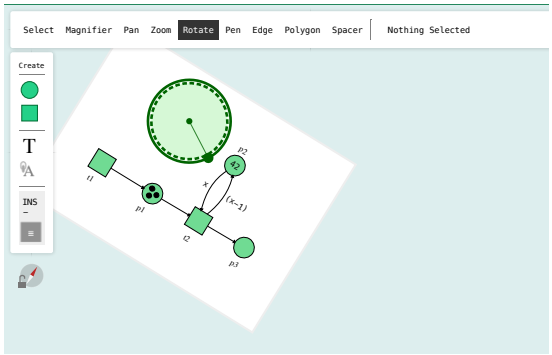
(b) Lupenzoom Ende



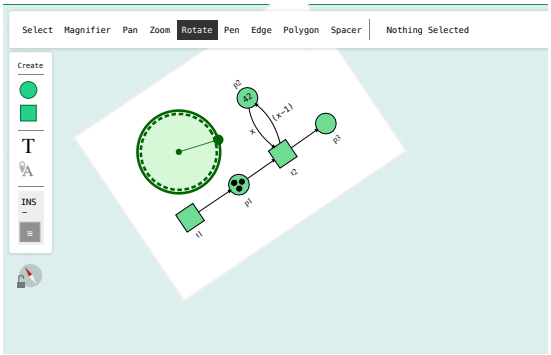
(c) Polar-Zoom Start



(d) Polar-Zoom Ende



(e) Rotation Start



(f) Rotation Ende

Abbildung 17.5.: Werkzeuge zur einhändigen Navigation der Kamera, Lupenzoom in der oberen Zeile, Polarzoom in der mittleren Zeilen. Rotation in der unteren Zeile. Auf der linken Seite jeweils zu Beginn der Operation. Auf der rechten Seite das Ergebnis der Operation.

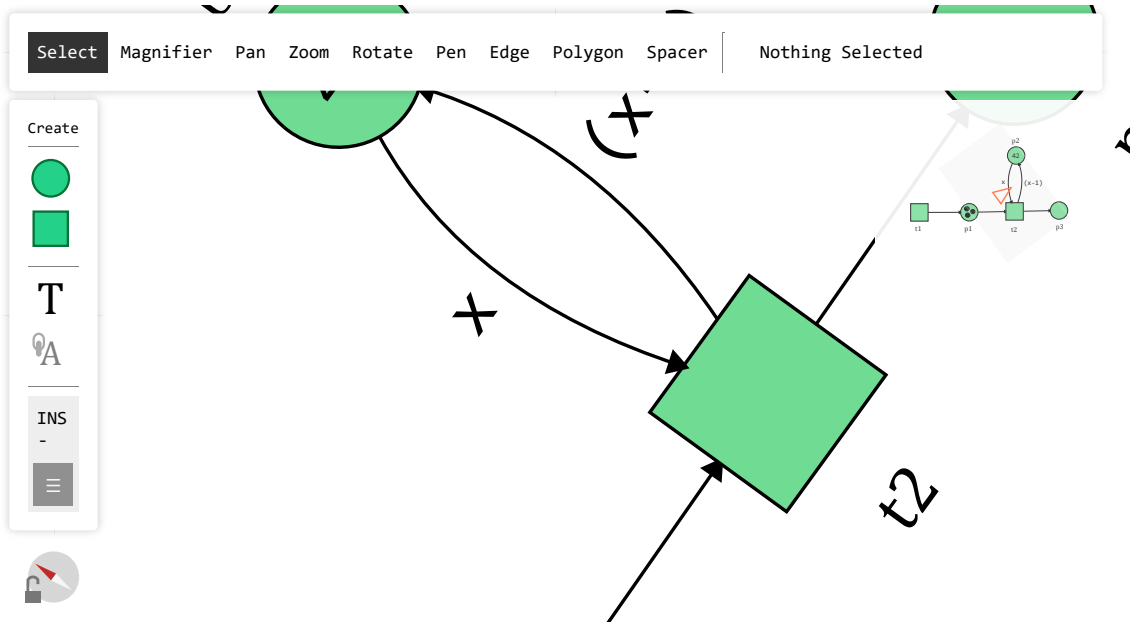


Abbildung 17.6.: Screenshot der Benutzeroberfläche des Web-Editor mit Minimap auf der rechten Seite

18. Lokale und verteilte Bearbeitung

In den vorherigen Kapiteln zur Umsetzung der Client-Komponente wurde hauptsächlich die Darstellung der vom Server bereitgestellten Dokumente behandelt. In diesem Kapitel geht es um die Umsetzung von Änderungsoperationen durch den Editor-Client. Dafür werden sowohl Kommandos an den Server geschickt, als auch einfache Änderungen direkt lokal durchgeführt. Letztendlich wird zum Verringern der beobachtbaren Latenz zwischen Eingabe und Änderung auf eine Kombination aus entfernten und lokalen Bearbeitung gesetzt.

18.1. Bearbeitung mittels Kommandos am Server

Um eine Veränderung an einem Dokument durchzuführen, muss das entsprechende Kommando vom Editor-Server durchgeführt werden. Um beispielsweise die Schriftfarbe des Textelements mit der ID 23 auf Blau zu verändern, muss der Befehlsaufruf `set_text_style(23, blue)` an den Server geschickt werden. Die Menge der verfügbaren Befehle wurde in Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ aufgeführt.

Aus Perspektive der Client-Komponente des Web-Editors besteht der Arbeitsablauf an einem Dokument also sehr einfach aus: Der initiale Inhalt des Dokuments wird über die HTTP-Schnittstelle abgerufen. Der Inhalt des Dokuments wird als SVG formatiert und dargestellt. Es wird eine Web-Socket-Verbindung aufgebaut. Über diese können Änderungen in Form von Patch-Anweisungen empfangen werden. Den Benutzenden wird in der grafischen Oberfläche eine Auswahl an verfügbaren Änderungsoperationen präsentiert. Bei entsprechender Nutzerinteraktion wird eine Änderungsoperation an den Server geschickt. In Folge einer erfolgreich durchgeführten Änderung wird eine entsprechende Patch-Anweisung empfangen. Werden Änderungen empfangen, wird die lokale Kopie des Dokuments entsprechend verändert und die SVG-Darstellung angepasst.

Der Client muss entsprechend nicht in der Lage sein, umfangreiche Änderungsoperationen eigenständig am Dokument durchzuführen. Dafür ist er aber von der serverseitigen Verarbeitung der Änderungen abhängig.

18.2. Clientseitige Bearbeitung im Reaktiven Modell

Einzelne Änderungen, die sich nicht auf die Struktur des Dokuments auswirken, sondern nur einzelne Attribute von Elementen verändern, können im Client aber auch lokal durchgeführt werden. Beispielsweise kann der Client den Farbwert eines Elements eigenständig in seiner lokalen Kopie verändern.

Die Verwendung der in Kapitel 16 „Reaktives Datenmodell“ vorgestellten Signaldatenstruktur führt dazu, dass sich auch lokal durchgeführte Änderungen unmittelbar auf die dynamisch daraus abgeleitete SVG-Darstellung auswirken.

Lokal durchgeführte Änderungen sind aber natürlich flüchtig und werden nicht in dem auf dem Server gespeicherten Dokument persistiert, also auch nicht in der Versionshistorie, die in Kapitel 11 „Versionierung von Dokumenten“ behandelt wurde, protokolliert.

Lokale Änderungen können durch darauffolgend empfangene Patch-Anweisungen überschrieben werden. Dies geschieht aber nur, wenn sich die Patch-Anweisung auf den entsprechenden Teil des Dokuments bezieht. Denn bei der Verarbeitung der Patches wird davon ausgegangen, dass sich das Dokument des Clients in einem zuvor mit dem Server konsistenten Zustand befindet und nur geringfügige Änderungen vorgenommen werden müssen.

Der Client muss also darauf achten, nur solche lokalen Änderungen vorzunehmen, die nicht mit der Verarbeitung von empfangenen Patch-Anweisungen kollidieren. Die Änderung von Farbwerten ist ein sinnvoller Einsatz von lokalen Änderungen, weil den Benutzenden die Vorschau einer gewählten Farbe schon in der Dokumentendarstellung angezeigt werden kann, bevor er die Farbänderung bestätigt und diese vom Server als Änderung verarbeitet wird.

18.3. Latenz

Die vom Server verarbeiteten Änderungen unterscheiden sich von den lokal verarbeiteten Änderungen deutlich wahrnehmbar in ihrer Latenz. Die Zeit zwischen dem Senden der Operation an den Server bis zur aktualisierten grafischen Darstellung des Dokuments in der clientseitigen Benutzeroberfläche kann einige Millisekunden dauern. Neben der Verzögerung durch die Netzwerkkommunikation mit dem Server hat die Menge an Elementen im Dokument den primären Einfluss auf die Latenz.

Dokumente mit vielen Elementen sind in der Datenbank aufwändiger zu verarbeiten, weil kombinatorisch mehr Konsistenz-Bedingungen zwischen den Elementen entstehen, die überprüft werden müssen. Das Anlegen eines Eintrags in der Versionshistorie ist für große Dokumente aufwändiger, weil eine vollständige Kopie des Inhalts abgespeichert werden muss. Die Differenz zweier Zustände zur Erstellung eines JSON-Patches zu berechnen, ist für Dokumente mit vielen Elementen aufwändiger.

Clientseitig wird die Latenz sehr stark dadurch bestimmt, wie aufwändig die Darstellung der erzeugten SVG für den Webbrowser ist. Das wiederum hängt primär von den intern im Browser verwendeten Algorithmen zur Rasterung von Vektorgrafiken ab. In Experimenten dieser Arbeit hat sich gezeigt, dass je nach Browser und Browserversion eine große Menge an dargestellten Textelementen in einer SVG die Leistung des Web-Editors in die Knie zwingen kann.

Für die Arbeit an typischen Renew Dokumenten ist die Latenz des gesamten Web-Editors aber gering genug, um ihn ohne Frustration verwenden zu können.

18.4. Kombination aus lokaler und verteilter Bearbeitung

Die Latenz der Netzwerkkommunikation kann zusätzlich kaschiert werden. Die Werkzeuge zum Bearbeiten von Dokumenten in der Benutzeroberfläche wurden so konzipiert, dass lokale und serverseitige Änderungen in Kombination und aufeinander abgestimmt durchgeführt werden.

Besonders bei kontinuierlichen Nutzereingaben können sehr viele kleinschrittige Änderungen ausgelöst werden. Eine Mausbewegung kann beispielsweise die Position eines Elements auf der Zeichenfläche annähernd kontinuierlich verändern. Das Drehen des Mausrads kann die Schriftgröße eines Textelements in sehr kurzer Zeit schrittweise um viele kleine Schritte verändern.

Für diese kontinuierlichen Eingaben wäre es aus zwei Gründen nicht sinnvoll, Änderungsoperationen an den Server zu schicken. Zum einen ist mit jeder Änderungsoperation auch ein Protokolleintrag in der Dokumentenhistorie verbunden. Damit würden die beschriebenen Nutzereingaben eine Vielzahl an Protokolleinträgen erzeugen, die sich dann nur schwer navigieren lassen. Zum anderen würde sich die beschriebene Latenz bei der Verarbeitung von so vielen Änderungsoperationen in kurzer Zeit auch aufsummieren.

Daher sind alle Werkzeuge des Editor-Clients so entwickelt worden, dass für kontinuierliche Nutzereingaben nur lokale Änderungen am Dokument im Client durchgeführt werden. Erst nach dem Abschluss einer Eingabesequenz wird der akkumulierte Änderungsbefehl an den Server geschickt.

Beim Umsetzen von rein lokalen Änderungen kann zeitweilig die lokale Konsistenz eines Dokuments verletzt werden. Diese wird aber immer wieder hergestellt, sobald der vom Server berechnete Patch empfangen und verarbeitet wird. Ein Beispiel hierfür ist das Verschieben eines Knotens auf der Zeichenfläche mit der Maus bei gedrückter Maustaste. Damit die Knotenposition ohne Latenz der Mausbewegung folgt, wird die Position des Elements vom Client nur lokal verändert, ohne ein Kommando an den Server zu schicken. Dadurch werden aber die Positionen von Kantenendpunkten nicht neu berechnet. In der grafischen Darstellung lösen sich die ein- und ausgehenden Kanten also vom Knoten. Es wirkt auf den Benutzenden, als wären die Kanten nicht mit dem Knoten verknüpft. Sobald die Maustaste aber losgelassen wird, um die neugewählte Position des Knotens zu bestätigen, wird das entsprechende Kommando an den Server geschickt. Der Server verarbeitet die Änderung, berechnet alle Kantenpositionen neu, legt einen Versionsprotokolleintrag an und schickt resultierenden Patch-Anweisungen an alle verbundenen Clients. Schlagartig springen in der grafischen Darstellung die Kantenendpunkte an die nun korrekte Position.

Aus dieser Verarbeitungsreihenfolge ergibt sich auch, dass mehrere Benutzer:innen, die gemeinsam an einem Dokument arbeiten, immer nur die final durchgeführten Änderungen der jeweils anderen Benutzer:innen beobachten können. Die beschriebenen optisch wahrnehmbaren Wiederherstellung der Konsistenz kann für die Benutzenden auch als Empfangsbestätigung des Servers interpretiert werden. Ändert sich der dargestellte Do-

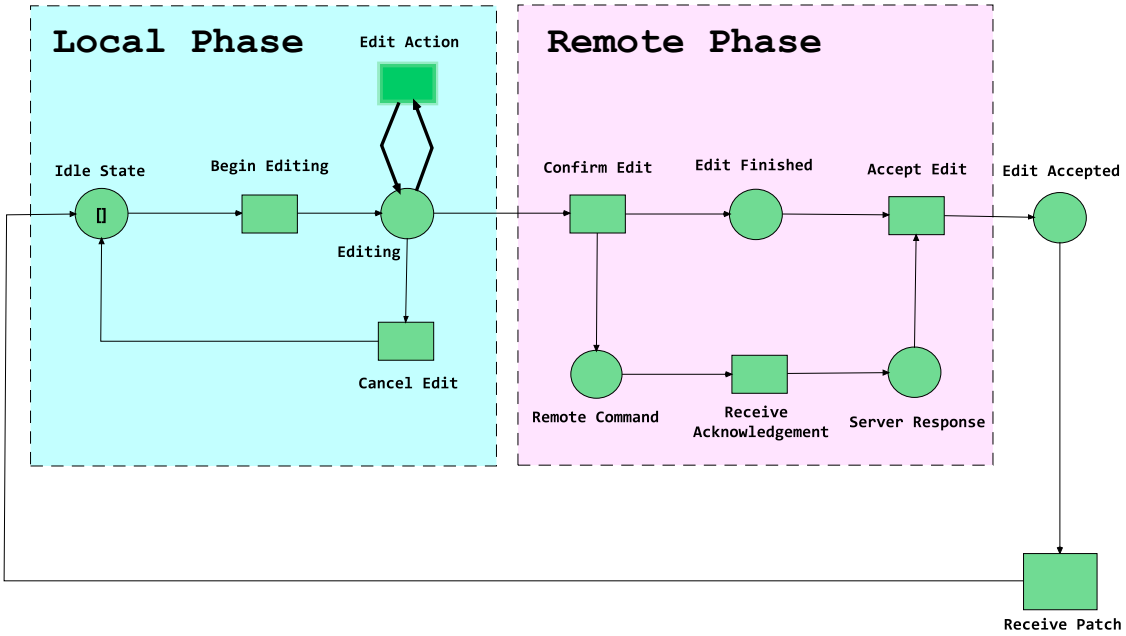


Abbildung 18.1.: Workflow zur Koordination von lokalen und verteilten Bearbeitungsoperationen in zwei Phasen.

kumenteninhalt auf geringfügige Weise, kann man sich sicher sein, dass die durchgeführte Änderung persistiert wurde.

Abb. 18.1 zeigt den Ablauf einer Bearbeitungsoperation. Im *Idle State* ganz links ist das Dokument in einem konsistenten Zustand. Sobald mit der Bearbeitung begonnen wird, befindet sich das Dokument im Zustand *Editing* und ist gegebenenfalls nicht mehr vollständig konsistent. Es kann aber durch beliebige Wiederholungen der *Edit Action* ohne Latenz bearbeitet werden. Die Bearbeitung kann abgebrochen werden und damit das Dokument in den zuletzt konsistenten Zustand zurückgesetzt werden. Alternativ kann die Änderung bestätigt werden. Dann wird ein Änderungskommando an den Server geschickt und auf dessen Antwort gewartet. Der vom Server empfangene Patch überführt das Dokument lokal wieder in einen konsistenten Zustand.

19. Kollaboratives Arbeiten

In diesem Kapitel werden einige weitere Funktionen behandelt, die das gemeinsame Arbeiten an einem Dokument unterstützen. Aus der bisher beschriebenen Architektur ergibt sich auf natürliche Weise, dass mehrere Anwender gemeinsam an einem Dokument arbeiten können. Alle Änderungen werden zentral vom Server koordiniert und der daraus resultierende aktuellste Zustand eines Dokuments wird allen Client-Anwendungen zur Verfügung gestellt.

Im Folgenden werden einige Werkzeuge vorgestellt, die das gemeinsame Arbeiten an einem Dokument unterstützen. Dazu gehören die Auflistung anderer aktiver Benutzer:innen, die Darstellung mehrerer Mauszeiger und die farbliche Hervorhebung von ausgewählten Elementen.

Abb. 19.1 zeigt einen Screenshot der Benutzeroberfläche, während mehrere Personen an dem geöffneten Dokument arbeiten. Zu sehen ist, dass oben rechts ein lila Kreis die Anwesenheit einer weiteren Person markiert. Außerdem wird ein Element rechts im Dokument farbig hervorgehoben. Es ist auch ein farbiger Mauszeiger rechts im Dokument zu erkennen.

19.1. Onlinestatus

Während man an einem Dokument arbeitet, ist es hilfreich zu wissen, ob jemand anderes ebenfalls an diesem Dokument arbeitet. Dafür wird im Web-Editor Client eine Liste aller anderen Personen angezeigt, die derzeit im Dokument aktiv sind.

Diese dargestellte Liste an Personen bezieht sich darauf, wie viele WebSocket-Verbindungen zu dem Kommunikationskanal des jeweiligen Dokuments bestehen. Das vom Web-Editor verwendete Phoenix Channel Protokoll bietet eine für solche Anforderungen vorgesehene `Presence` Schnittstelle [ME23b]. Jede über die `Presence` Schnittstelle erfasste Verbindung kann auch mit zusätzlichen Metadaten annotiert werden. So ordnet der Web-Editor Server beispielsweise jeder Person einen individuellen Farbwert zu, der für die Darstellung des Listeneintrags verwendet wird.

In Abb. 19.1 wurde der anderen Person die Farbe Lila zugeordnet. Der Anfangsbuchstabe ihres Namens wird zusammen mit der Farbe oben rechts als kleiner Kreis dargestellt.

19.2. Auswahl von Elementen

Zusätzlich zu der Auflistung anderer aktiven Personen wird auf der Zeichenfläche des Web-Editors auch grafisch dargestellt, welches Element jeweils eine andere Person ausgewählt hat. Diese Darstellung ist nützlich, um sich bei gemeinsamer Arbeit in einem

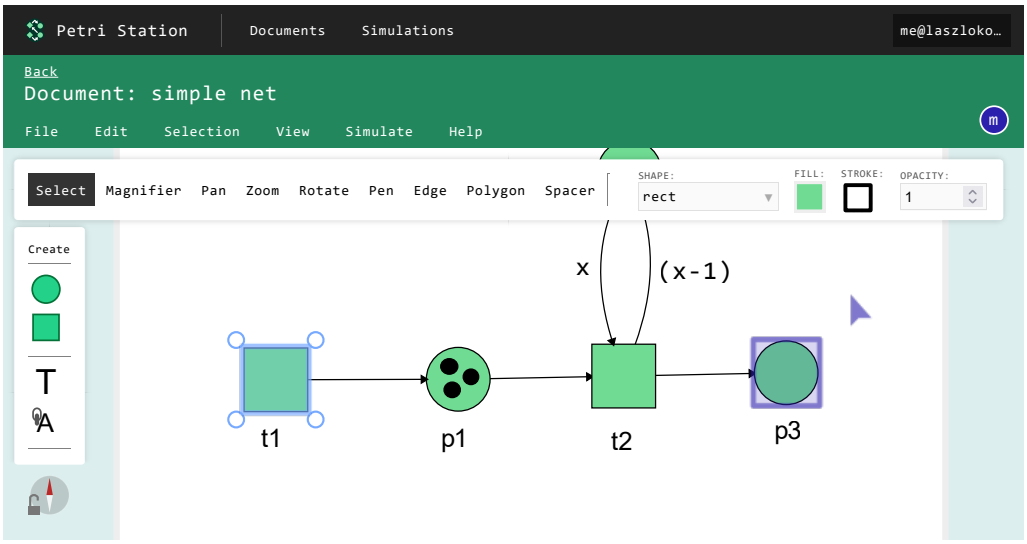


Abbildung 19.1.: Screenshot der kollaborativen Werkzeuge des Web-Editors.

Dokument nicht in die Quere zu kommen. Von anderen Personen ausgewählte Elemente werden jeweils in der ihr automatisch zugeordneten Farbe dargestellt.

In Abb. 19.1 ist zu erkennen, dass die Person, die den Screenshot aufgenommen hat, selbst das Element ganz links im Dokument ausgewählt hat. Dieses ist hellblau hervorgehoben und bietet kreisförmige Schaltflächen für die Bearbeitung an. Eine weitere Person hat das Element ganz rechts im Dokument ausgewählt. Das ist an der lilafarbenen Umrandung des Elements zu erkennen.

Welche Person welches Element ausgewählt hat, wird als Metadaten in der von Phoenix Channel verwalteten `Presence` gespeichert.

Im Rahmen dieser Arbeit werden für den Web-Editor aber keine strengen Sperrmechanismen umgesetzt. Alle Benutzer:innen können alle Dokumente und alle darin enthaltenen Elemente bearbeiten. Kommen sich zwei Bearbeitungsoperationen an einem Dokument in die Quere, werden diese einfach seriell ausgeführt. Die meisten Kommandos im Web-Editor verhalten sich idempotent, sodass durch die Kombination mehrerer Aktionen verschiedener Benutzer:innen keine unvorhersehbaren Auswirkungen entstehen.

19.3. Position des Mauszeigers

Auch die Position des Mauszeigers auf der Zeichenfläche wird vom Web-Editor Client an den Server übertragen und dort in der `Presence` Datenstruktur abgelegt. Dadurch können in der Benutzeroberfläche die Mauszeiger aller anderen aktiven Benutzer:innen angezeigt werden. Die Mauszeiger werden auch jeweils in der der Person zugeordneten Farbe dargestellt. Die Darstellung der Mauszeiger lässt sich über einen Eintrag im Ansicht-Menü deaktivieren.

In Abb. 19.1 wird der eigene Mauszeiger nicht dargestellt. Allerdings ist auf der rechten Seite in Lila der Mauszeiger einer anderen am Dokument arbeitenden Person zu erkennen.

Für die Zusammenarbeit in einem Dokument können die Mauszeiger verwendet werden, um auf Inhalte eines Dokuments zu zeigen oder um noch genauer nachzuvollziehen, in welchen Bereichen eines Dokuments andere Personen derzeit arbeiten.

Teil V.

Umsetzung des Simulators

20. Anbindung an Java Renew

In diesem Kapitel wird die Einbindung des bestehenden Referenznetzsimulators von Renew in den Web-Editor behandelt. In den vorherigen Kapiteln lag der Fokus auf dem Importieren, Bearbeiten und Darstellen von Petrinetz-Dokumenten. Damit entspricht die bisher erreichte Funktionalität einem kollaborativen Grafikeditor. Nun sollen die erstellbaren Netze mit einer simulierbaren Semantik versehen werden.

Für die Simulation soll der in Renew enthaltene Simulator zum Einsatz kommen. Dafür wird Renew über die Befehlszeile angesteuert. Zum einen, um die Dokumente in Shadow-Net-Systems (SNS) zu übersetzen. Zum anderen, um einen Simulationsprozess zu starten, zu steuern und den Simulationszustand im Laufe der Simulation auszulesen.

20.1. Ansteuerung von Renew per Befehlszeile

Um die im WebEditor gezeichneten Netze mittels Renew zu simulieren, muss das Renew-Simulator-Modul aus dem WebEditor heraus angesteuert werden. Renew enthält mit dem `Console` Plugin eine befehlszeilenbasierte Benutzeroberfläche, die sich unabhängig von der grafischen Benutzeroberfläche verwenden lässt. Wird Renew im sogenannten Headless-Modus gestartet, können über den Standardeingabedatenstrom Textbefehle an Renew gesendet werden. Diese Befehle werden dann sequenziell von Renew verarbeitet. Über den Standardausgabe-Datenstrom werden die zugehörigen Ergebnisse oder Fehlmeldungen zurückgemeldet.

Anstatt die Befehle manuell einzugeben, können auch andere Prozesse und Anwendungen mit den Ein- und Ausgabedatenströmen verbunden werden, um die Befehle und Rückmeldungen automatisiert zu verarbeiten. Für die Simulation von Netzen soll der Web-Editor also an diese Befehlszeilenschnittstelle angeschlossen werden, um mit dem Simulator in Renew zu kommunizieren.

Diese Schnittstelle wurde gewählt, da sie bereits zur Verfügung steht und Renew somit im Rahmen dieser Arbeit nicht verändert werden musste. Allerdings können so auch nur die über die Befehlszeile zur Verfügung stehenden Funktionen von Renew verwendet werden.

20.1.1. Interprozesskommunikation mittels Elixir Port

In Elixir kann der Erlang `Port` Modul verwendet werden, um externe Prozesse zu starten und mit diesen über die Standarddatenströme zu kommunizieren. Das `Port` bietet eine Vielzahl an Optionen für die Konfiguration der Interprozesskommunikation. Diese Konfiguration musste auf das Verhalten des Renew-Console-Plugins abgestimmt werden.

Eine Herausforderung war es, eine mit unterschiedlichen Betriebssystemen kompatible Konfiguration zu finden, da sich Windows, Mac und Linux besonders in der Verarbeitung und der Pufferung von Datenströmen unterscheiden. Listing 20.1 zeigt die verwendete Konfiguration.

```
Port.open({:spawn_executable, System.find_executable("java")}, [
    :binary,
    :eof,
    :stream,
    :stderr_to_stdout,
    :exit_status,
    :use_stdio,
    :hide,
    line: 2048,
    env: [{to_charlist("DISPLAY"), 23}],
    cd: "/tmp",
    args: [
      "-jar",
      "Adapter.jar",
      "java",
      "-Dde.renew.plugin=ERROR,nullAppender",
      "-Dlog4j.appender.nullAppender=org.apache.log4j.varia.NullAppender",
      "-Dlog4j.configuration=simulator.properties",
      "-Djline.terminal=off",
      "-Dorg.jline.terminal.dumb=true",
      "-Dde.renew.splashscreen.enabled=false",
      "-Dde.renew.gui.autostart=false",
      "-Dde.renew.simulatorMode=-1",
      "-Dde.renew.plugin.autoLoad=false",
      "-Dde.renew.plugin.load=Renew Util, Renew Simulator, Renew Formalism,
          Renew Misc, Renew PTChannel, Renew Remote, Renew Window Management,
          Renew JHotDraw, Renew Gui, Renew Formalism Gui, Renew Logging, Renew
          NetComponents, Renew Console, Renew FreeHep Export",
      "-p",
      "renew41:renew41/libs",
      "-m",
      "de.renew.loader"
    ]
  ]
)
```

Quelltext 20.1: Die für die Simulatoranbidung verwendete Port-Konfiguration

20.1.2. Standardeingabe Datenstrom

In Kapitel 2 „Grundlagen“ und Kapitel 5 „Strategie zur Umsetzung“ wurde schon erwähnt, dass Elixir und Erlang eine eigene Semantik für die robuste Behandlung von Laufzeitfehlern mitbringen. Das Modell der Supervisor-Bäume ermöglicht das Neustarten und Zurücksetzen von einzelnen Modulen, ohne das Gesamtsystem zu beeinträchtigen. Bei der Verwendung des `Port` Moduls zur Kommunikation mit externen Prozessen gelten in diesem Zusammenhang zusätzlich Anforderungen. Erlang erwartet, dass ein externer Prozess automatisch beendet wird, wenn der zugehörige Standardeingabedatenstrom seitens Erlang geschlossen wird. Dieses Verhalten ist notwendig, damit — unabhängig des zugrundeliegenden Betriebssystems — externe Prozesse durch Erlang zuverlässig beendet werden können. Dieses Verhalten kann von Portable Operating System Interface (POSIX) kompatiblen Prozessen auch im Allgemeinen erwartet werden.

Das Renew `Console` Plugin verhält sich aber nicht wie erwartet. Wird der Standardeingabe-Datenstrom geschlossen, bleibt der Renew Prozess bestehen und wartet weiter auf Eingaben. Das führt dazu, dass aus Elixir heraus gestartete Java Renew Prozesse nicht zuverlässig wieder beendet werden und sich mit der Zeit ansammeln.

Im Rahmen dieser Arbeit wurde entschieden, das Verhalten in Renew nicht zu korrigieren. Zum einen, weil laut Anforderungen Renew nicht verändert werden soll. Zum anderen, weil nicht ermittelt werden konnte, ob das derzeitige Verhalten von Renew von anderen Systemen vorausgesetzt wird.

Stattdessen wurde ein Adapter in Java entwickelt, der als Vermittler zwischen dem Elixir Web-Editor und dem Java Renew Prozess geschaltet wird. Dieser Adapter leitet lediglich die Ein- und Ausgabe-Datenströme zwischen dem Elixir und Renew Java Prozess weiter. Außerdem reagiert er auf das Schließen des Eingabe-Datenstroms mit einer sofortigen Beendigung des Renew Prozesses. Quellcode 20.2 zeigt die Implementation dieses Adapters in Java.

Damit stellt dieser Adapter sicher, dass alle durch Elixir gesteuerten Java-Prozesse korrekt terminiert werden.

```java
import java.io.*;
import java.util.concurrent.*;
import java.util.Map;
import java.util.Set;

public class Interceptor {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("Usage: java Interceptor <command> [args...]");
            System.exit(1);
        }

        Process childProcess = null;
        final ProcessBuilder processBuilder = new ProcessBuilder(args);

        final Set<String> variablesToKeep = Set.of("DISPLAY");
        final Map<String, String> env = processBuilder.environment();
        final Map<String, String> filteredEnv = env.entrySet().stream()
            .filter(entry -> variablesToKeep.contains(entry.getKey()))
            .collect(java.util.stream.Collectors.toMap(Map.Entry::getKey, Map.
                Entry::getValue));
        env.clear();
        env.putAll(filteredEnv);

        try {
            final Process process = processBuilder.start();
            childProcess = process;

            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                process.destroyForcibly();
            }));

            CompletableFuture<Void> input = CompletableFuture.runAsync(() -> {
                try (OutputStream processInput = process.getOutputStream();
                     InputStream stdin = System.in) {
                    byte[] buffer = new byte[1024];
                    int bytesRead;
                    while ((bytesRead = stdin.read(buffer)) != -1) {
                        processInput.write(buffer, 0, bytesRead);
                        processInput.flush();
                    }
                } catch (IOException e) {
                    e.printStackTrace(System.err);
                }
            });

            CompletableFuture<Void> output = CompletableFuture.runAsync(() -> {
                try (InputStream processOutput = process.getInputStream();
                     OutputStream stdout = System.out) {
                    byte[] buffer = new byte[1024];
```

```
50                      int bytesRead;
51                      while ((bytesRead = processOutput.read(buffer)) != -1) {
52                          stdout.write(buffer, 0, bytesRead);
53                          stdout.flush();
54                      }
55                  } catch (IOException e) {
56                      e.printStackTrace(System.err);
57                  }
58              });
59
60              CompletableFuture<Void> error = CompletableFuture.runAsync(() -> {
61                  try (InputStream processError = process.getErrorStream();
62                       OutputStream stderr = System.err) {
63                      byte[] buffer = new byte[1024];
64                      int bytesRead;
65                      while ((bytesRead = processError.read(buffer)) != -1) {
66                          stderr.write(buffer, 0, bytesRead);
67                          stderr.flush();
68                      }
69                  } catch (IOException e) {
70                      e.printStackTrace(System.err);
71                  }
72              });
73
74              CompletableFuture.anyOf(input, output, error).join();
75
76              process.destroyForcibly();
77
78              process.waitFor();
79              System.exit(0);
80          } catch (IOException | InterruptedException e) {
81              e.printStackTrace(System.err);
82              System.exit(1);
83          } finally {
84              if (childProcess != null) {
85                  childProcess.destroyForcibly();
86              }
87          }
88      }
89  }
```

Quelltext 20.2: Adapter zur Verwaltung von Ein- und Ausgabedatenströmen

20.2. Erzeugen von Shadow-Net-Systems

Für die Simulation verwendet Renew eine gegenüber den rnw-Dateien vereinfachte und kompakte Repräsentation von Netzen. Die sogenannten Shadow Net System (SNS) enthalten die topologische Struktur der zu simulierenden Netze und die Textanschriften der einzelnen Netzelemente. Anders als eine rnw-Datei enthält ein SNS keine Informationen zur grafischen Darstellung der Elemente — also keine Farben, Positionen oder grafischen Symbole. Dafür kann eine SNS aber alle für eine Simulation benötigten Netze zu einem sogenannten Netzsystem zusammengefasst in einer Datei speichern — daher der Name SNS.

Der Renew-Simulator erwartet als Eingabe eine SNS-Datei und den Namen eines darin enthaltenen Netzes, welches als Einstiegspunkt in die Simulation verwendet werden soll. Um ein im Web-Editor gezeichnetes Netz im Renew-Simulator simulieren zu können, muss das Netz also zunächst in ein SNS konvertiert werden.

20.2.1. Renew rnw-Dateien in SNS kompilieren

In Kapitel 6 „Import von rnw-Dateien“ und Kapitel 12 „Export von rnw-Dateien“ wurde bereits erklärt, wie zwischen dem rnw-Dateiformat und dem Datenformat des Web-Editor übersetzt werden kann. In Renew ist ein SNS-Compiler zur Übersetzung von rnw-Dateien in das SNS-Format enthalten. Dieser Compiler lässt sich über die Befehlszeile von Renew aufrufen.

Um im Web-Editor erstellte Netze für die Simulation vorzubereiten, werden sie zunächst als rnw-Dateien serialisiert. Diese rnw-Dateien werden dann über die Befehlszeilenschnittstelle an den SNS-Compiler von Renew übergeben. Als Ergebnis entsteht eine SNS-Datei, die vom Web-Editor wieder eingelesen für die spätere Simulation abgespeichert wird.

20.2.2. Synthetische Identifikatoren

Die vom Renew-Simulator zurückgemeldeten Simulationsergebnisse und -Ereignisse beziehen sich auf die Struktur und die Elemente des simulierten Netzes. Wenn im Verlauf der Simulation beispielsweise eine Transition in einer Netzinstanz schaltet, so wird dies vom Simulator protokolliert. Für die Interpretation des Protokolls müssen alle Netze, Plätze und Transitionen eindeutig benannt sein.

In Renew können Plätze und Transitionen optional mit manuell vergebenen Namen versehen werden. Dies kann das manuelle Lesen des Simulationsprotokolls erleichtern. Nicht manuell benannte Elemente werden von Renew automatisch durch eine aufsteigende Sequenz an Zahlen benannt.

Für die grafische Darstellung des Simulationsverlaufs im Web-Editor müssen alle Elemente und Ereignisse im Simulationsprotokoll vom Web-Editor automatisch verarbeitet, interpretiert und den zugehörigen Elementen im Dokument zugeordnet werden. Um

dies zu ermöglichen, werden beim Serialisieren der für die Simulation verwendeten rnw-Dateien automatisch zusätzliche eindeutige Namen für alle Netzelemente vergeben. Als Namen werden die UUIDs Primärschlüssel des in Kapitel 8 „Relationales Datenschema“ diskutierten relationalen Schemas verwendet.

20.2.3. Renew nicht ohne AWT verwendbar

In Renew wird die Java AWT Programmbibliothek für die grafische Benutzeroberfläche verwendet. AWT steht in Java-Installationen auf Servern ohne grafische Desktopumgebung aber oft nicht zur Verfügung. Für die Verwendung des Simulators kann Renew auch ohne grafische Benutzeroberfläche gestartet werden. Dies funktioniert auch auf Systemen ohne AWT Simulation. Allerdings benötigen einige Renew Plugins selbst dann Zugriff auf AWT, wenn keine grafische Benutzeroberfläche gestartet wird. Eines dieser Plugins ist das `Formalism` Plugin, welches für das Kompilieren von SNS benötigt wird. Für den Einsatz von Renew zur Simulation im Hintergrund des Web-Editors wird also eine vollständige AWT-Umgebung benötigt, obwohl Renew nur im Headless-Modus ausgeführt wird. Hierfür wird unter Linux das Paket X Window Virtual Framebuffer (XVFB) verwendet [Wig12].

20.3. Simulationsteuerung per Befehlszeile

Nachdem eine SNS-Datei erzeugt wurde, kann diese über die Befehlszeilenschnittstelle an den Renew-Simulator übergeben werden, um einen Simulationsprozess zu starten. Der Simulator unterstützt nur eine Simulationsinstanz zur Zeit. Für mehrere nebenläufige Simulationen müssen also mehrere Instanzen von Renew gestartet werden. Über das in Abschnitt 20.1.1 vorgestellte Elixir Port Modul ist der Web-Editor aber in der Lage, mehrere Instanzen von Renew zu starten und zu verwalten.

20.3.1. Konfiguration von Log4J

Log4j ist eine JavaBibliothek zum Protokollieren von Ereignissen in Java-Anwendungen. In Renew wird Log4J unter anderem verwendet, um die Ereignisse eines Simulationsprozesses für andere Komponenten des Systems zur Verfügung zu stellen [Per09]. Neben den Simulationsereignissen werden in Renew aber auch viele andere Datenströme protokolliert. Für die Verarbeitung der Simulation im Web-Editor sind nur die Informationen aus dem Simulationsprozess von Interesse. Log4j bietet viele Optionen, um die Protokolleinträge zu filtern und in verschiedene Datenströme aufzuteilen.

Für den Einsatz des Simulators in dieser Arbeit wurde die Log4j-Konfiguration von Renew so angepasst, dass ausschließlich die Simulationsereignisse in den Standardausgabestrom geleitet werden. Dieser wird vom Web-Editor entgegengenommen und verar-

```
log4j.logger.de=ERROR, RenewConLog
log4j.logger.CH=ERROR, RenewConLog
log4j.logger.jamr=ERROR, RenewConLog
log4j.logger.net=ERROR, RenewConLog
log4j.appender.RenewConLog=org.apache.log4j.ConsoleAppender
log4j.appender.RenewConLog.layout=org.apache.log4j.PatternLayout
log4j.appender.RenewConLog.layout.ConversionPattern=%p: %m%n
log4j.logger.simulation=TRACE, SimConLog
log4j.appender.SimConLog=org.apache.log4j.ConsoleAppender
log4j.appender.SimConLog.layout=org.apache.log4j.PatternLayout
log4j.appender.SimConLog.layout.ConversionPattern= %m%n
```

Quelltext 20.3: Log4J Konfiguration zum Auslesen des Simulatorprotokolls

beitet. Die vom Web-Editor für die Simulation verwendete Log4J-Konfiguration wird in Listing 20.3 dargestellt.

20.3.2. Renew-Simulator per Console Plugin steuern

Der Renew-Simulator verwendet ein diskretes Zeitmodell für die Simulation von Petrinetzen. Der Simulationsprozess startet mit einem Startzustand und überführt mit jedem Zeitschritt den jeweils aktuellen Zustand anhand der im Netz definierten Transitionen in einen Folgezustand. Abhängig vom aktuellen Zustand und der Struktur und Semantik des simulierten Netzes ist der jeweilige Folgezustand nicht eindeutig. Je nach konfigurierter Semantik wählt der Renew-Simulator aus der Menge der möglichen Folgezustände.

Mittels verschiedener Operationen kann Einfluss auf diesen Prozess genommen werden. Ein Simulationsprozess kann pausiert und fortgesetzt werden. In einer pausierten Simulation verweilt der Prozess in dem aktuellen Zustand. Wird ein Simulationsprozess fortgesetzt, berechnet der Simulator eigenständig so schnell wie möglich den jeweils nächsten aktuellen Zustand.

Eine pausierte Simulation kann aber auch manuell in den Folgezustand überführt werden. In der grafischen Benutzeroberfläche von Renew gibt es zwei Möglichkeiten, einen manuellen Simulationsschritt durchzuführen. Entweder es wird per Mausklick eine bestimmte Transition ausgewählt, die mit einer bestimmten Belegung geschaltet werden soll, oder man lässt den Simulator automatisch eine beliebige aktive Transition schalten. Für die manuelle Auswahl der zu schaltenden Transition lässt sich in der grafischen Benutzeroberfläche von Renew auch eine Liste aller möglichen Bindungen einer gewählten Transition anzeigen.

Über die Befehlszeile steht die manuelle Auswahl der zu schaltenden Transitionen aber nicht zur Verfügung. Daher kann diese Auswahl auch nicht vom Web-Editor angesteuert werden. Nur die Operationen zum Pausieren, Fortsetzen und Beenden werden von Renew auf der Befehlszeile angeboten. Somit sind es auch nur diese 3 Operationen, die im Rahmen dieser Arbeit im Web-Editor zur Verfügung stehen.

21. Speicherung des Simulationszustandes

Der Renew-Simulator stellt den Fortschritt und die Ergebnisse einer Simulation als Protokoll von Ereignissen bereit — welche Marken bewegen sich im Netz von welchem Platz zu welchem Platz. Während der Ausführung einer Simulation soll den Benutzer:innen aber der jeweils aktuelle Zustand der Simulation dargestellt werden — welche Marke liegt zum aktuellen Zeitpunkt auf welchem Platz. Für den Web-Editors muss also ein Modell zur Repräsentation des aktuellen Simulationszustandes erstellt werden.

Dafür wird zunächst ein relationales Datenmodell entwickelt. Also nächstes wird das von Renew sequenziell ausgegebene Simulationsprotokoll in einen Zustand akkumuliert, welcher dann in der relationalen Datenbank gespeichert wird.

21.1. Relationales Schema für Simulationen

In Kapitel 8 „Relationales Datenschema“ wurde die Verwendung eines relationalen Datenmodells für die Speicherung und Bearbeitung von Dokumenten motiviert. Die gleiche Motivation wird nun verwendet, um auch den Zustand einer Simulation in einem relationalen Modell zu erfassen.

Der Zustand einer Simulation setzt sich aus verschiedenen Entitäten und Attributen zusammen. Abb. 21.1 zeigt das erarbeitete Datenmodell in einem ER-Diagramm. Ein SNS besteht aus mehreren Shadow-Nets, für die jeweils eine Kopie des zugrundeliegenden Dokuments als JSON für die grafische Darstellung gespeichert ist. Für ein SNS können mehrere Simulationen durchgeführt werden. Für jede Simulation wird eine Liste an Protokolleinträgen gespeichert. Eine Simulation enthält mehrere Netzinstanzen. Diese sind jeweils auch dem zugrundeliegenden Netz aus dem SNS zugeordnet. Jede Netzinstanz enthält eine Menge an Marken (Token), die jeweils einem Platz zugeordnet sind. Außerdem enthält jede Netzinstanz die Liste aller bisher geschalteten Transitionen. Die Attribute `Place Id`, `Net Name`, `Instance Index` und `Transition Id` werden zur Zuordnung von Elementen zwischen dem Web-Editor und dem Renew -Simulator verwendet.

Eine Simulation basiert auf einem SNS. Ein SNS besteht aus mehreren Netzdefinitionen, wovon eine das initiale Netz ist. Während einer Simulation können mehrere Instanzen eines definierten Netzes erzeugt werden. Eine Marke (Token) befindet sich auf einem Platz einer Netzinstanz. Die Plätze der Marken und Netzdefinitionen haben universell eindeutige Namen, um Dokumenten und Grafikelemente zugeordnet werden zu können. Transitionen können zu einem Zeitpunkt innerhalb einer Netzinstanz schalten. Die geschalteten Transitionen sind über eindeutige Namen zur grafischen Darstellung einem Element in einem Dokument zugeordnet.

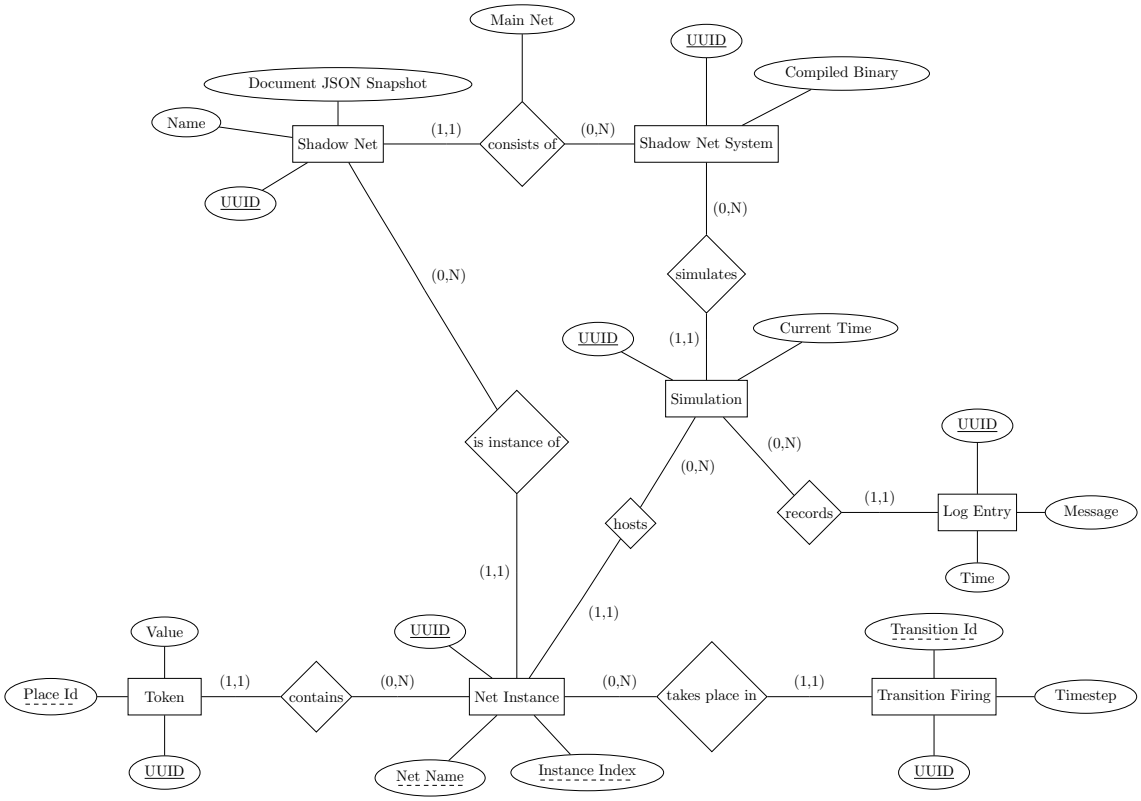


Abbildung 21.1.: ER-Diagramm zur Modellierung des gespeicherten Simulationszustandes im relationalen Datenmodell

Dieses relationale Modell wird für den Web-Editor in einer SQLite-Datenbank umgesetzt. Damit besitzt der Web-Editor für jede durchgeführte Simulation eine aktuelle Kopie des derzeitigen Simulationszustandes. Dieser Zustand kann unabhängig vom in Renew durchgeführten Simulationsprozess abgefragt und dargestellt werden.

21.2. Simulationszustand akkumulieren

Um das relationale Modell des Simulationszustandes mit Daten zu füllen, muss das von Renew bereitgestellte Simulationsprotokoll interpretiert und akkumuliert werden. Hierfür wird das in Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ behandelte Befehls-Entwurfsmuster verwendet. Jedes von Renew protokollierte Ereignis wird als Befehl interpretiert. Für jeden vom Simulator protokollierten Zeitschritt wird eine Transaktion in der relationalen Datenbank durchgeführt, um den gespeicherten Zustand auf den neuesten Stand der Simulation zu bringen.

Der Renew-Simulator protokolliert insgesamt sieben verschiedene Ereignisse. Listing 21.1 zeigt einen Ausschnitt eines von Renew generierten Simulationsprotokolls. Jede Zeile beginnt mit dem aktuellen ganzzahligen Zeitstempel der Simulation. *Simulation set up* signalisiert das erfolgreiche Starten einer neuen Simulation. *New net instance* signalisiert das erfolgreiche Erzeugen einer neuen Netzinstanz. *Initializing* signalisiert das Produzie-

```
(1)New net instance supernet[0] created.
(1)Initializing [] into supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
(1)Initializing [] into supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
(1)Initializing [] into supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
(1)Initializing [] into supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
Simulation set up, created net instance supernet[0].
Renew > (2)Removing [] in supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
(2)Firing subnet[2].c8f856b9-1903-4ea0-b555-a5e9bb3417d4
(2)Firing supernet[0].5aa33098-e47a-4c43-be85-c19fb5c9ce0c
(2)New net instance subnet[2] created.
(2)Putting int(5) into subnet[2].2f341fe8-9754-413e-93e6-26e168ce656e
(2)Putting subnet[2] into supernet[0].6ba39775-9304-4c2e-ad97-c637409c128a
(2)-------- Synchronously --------
Renew > (3)Removing [] in supernet[0].a69ebb34-c322-40ec-85c4-bfd6ade37684
(3)Firing subnet[5].c8f856b9-1903-4ea0-b555-a5e9bb3417d4
(3)Firing supernet[0].5aa33098-e47a-4c43-be85-c19fb5c9ce0c
(3)New net instance subnet[5] created.
(3)Putting int(5) into subnet[5].2f341fe8-9754-413e-93e6-26e168ce656e
(3)Putting subnet[5] into supernet[0].6ba39775-9304-4c2e-ad97-c637409c128a
(3)-------- Synchronously --------
```

Quelltext 21.1: Ausschnitt aus dem Renew Simulationsprotokoll.

ren einer initialen Belegung für eine zuvor neu erzeugte Netzinstanz. *Firing* signalisiert das Schalten einer Transition in einer Netzinstanz. *Removing* und *Putting* signalisieren das Konsumieren und Produzieren von Marken von und auf Plätzen einer Netzinstanz beim Schalten einer Transition. *Synchronously* signalisiert die Erhöhung des Zeitzählers der Simulation.

Jedes Ereignis im Protokoll ist mit einem monoton wachsenden Zeitstempel versehen. Aufeinanderfolgende Protokolleinträge können anhand des gemeinsamen Zeitstempels zu einem Zeitschritt gruppiert werden.

Der Web-Editor aktualisiert den akkumuliert gespeicherten Zustand der Simulation erst, sobald ein Zeitschritt des Simulators abgeschlossen ist, also wenn der Zeitzähler erhöht wird. Daraus ergibt sich, dass für die Benutzer:innen im Web-Editor stets ein konsistenter Zustand der Simulation präsentiert wird.

21.3. Drosselung der Simulation

Renew kann viele tausend Zeitschritte pro Sekunde simulieren. Dabei entsteht ein großer Durchsatz an Protokolleinträgen, die verarbeitet werden müssen, um den daraus akkumulierten Simulationszustand aktuell zu halten. Die Akkumulation der protokollierten Ereignisse in Echtzeit stellt eine Herausforderung dar, die im Rahmen dieser Arbeit nicht gelöst werden konnte.

Das Durchführen von Simulationen durch Renew in automatisch höchster Simulationsgeschwindigkeit führt zu einem Anhäufen von unverarbeiteten Protokolleinträgen im Standardausgabestrom. Dadurch wird sogar verhindert, dass eingehende Befehle zum

```
1 (1)Initializing into into into into into[0].0941982f
2 Simulation set up, created net instance into into[0].
3 (2)Removing into into in into into[0].0941982f
4 (2)Firing into into[0].8edabc14
5 (2)Putting into into into into into[0].5a031acd
```

Quelltext 21.2: Ein durch Renew erzeugtes, mehrdeutiges Simulationsprotokoll.

Steuern des Simulationsprozesses verarbeitet werden. Eine Simulation, deren Protokoll den Ausgabestrom überlastet, kann also nicht einmal pausiert werden.

Um so eine Überlastung zu verhindern, wird für die Simulation im Web-Editor nur die Simulation in Einzelschritten in Renew verwendet. Im Web-Editor kann ein einzelner Simulationsschritt durch Benutzer:innen ausgelöst werden. Wird im Web-Editor die automatische Simulation aktiviert, wird das automatische Voranschreiten der Simulation in Renew durch wiederholte, vom Web-Editor ausgelöste Einzelschritte nachgebildet. Damit kann der Web-Editor die Simulationsgeschwindigkeit daran anpassen, wie schnell das Ereignisprotokoll verarbeitet werden kann.

21.4. Eindeutige Interpretation des Ereignisprotokolls

Das von Renew erzeugte Simulationsprotokoll ist nicht darauf ausgelegt, maschinell weiterverarbeitet zu werden, sondern von Menschen lesbar zu sein. Ein einzelner Protokolleintrag kann durch Menschen vergebene Namen von Netzen und serialisierte Markenliterale enthalten. Diese wiederum können Leerzeichen und Sonderzeichen enthalten, die auch als Trennzeichen innerhalb eines Protokolleintrags verwendet werden. Daher lässt sich ein Protokolleintrag nicht immer eindeutig lesen. Ein Protokolleintrag mit mehrdeutigen Lesarten kann von Web-Editor nicht verarbeitet werden und führt zu einer fehlerhaften Akkumulation des Simulationszustandes. Listing 21.2 zeigt ein Beispiel für ein nicht verarbeitbares Protokoll. Dieses entstand daraus, dass sowohl das simulierte Netz den Namen *into into* trägt, als auch die initiale Marke auf einem Platz den String?-Wert `into into"` hat. Die daraus resultierenden Protokolleinträge *Removing into into in into into[o]* (Zeile 3) und *Putting into into into into into[o]* (Zeile 5) können mangels Trennzeichen nicht eindeutig entziffert werden.

Im Rahmen dieser Arbeit wurde dieses Problem nicht weiter behandelt. Eine Lösung bestünde darin, das Ausgabeformat des Simulationsprotokolls von Renew zu überarbeiten und zu vereinheitlichen. Bei einer solchen Überarbeitung kann auch in Betracht gezogen werden, das zuvor genannte Problem der Drosselung zu lösen. Anstatt alle Simulationsereignisse zu protokollieren, könnte Renew auch anbieten, den bereits akkumulierten Gesamtzustand einer Simulation in einem maschinenlesbaren Format über die Befehlszeilenschnittstelle abzufragen.

21.5. Fehlerbehandlung

Während einer Simulation können auch Fehler auftreten. Beispielsweise kann das zu simulierende Netz fehlerhaft sein, der Simulator kann abstürzen oder Transitionen können aus einem bestimmten Grund nicht schalten.

Die verschiedenen Fehlerfälle werden vom Renew-Simulator unterschiedlich ausführlich protokolliert. Im Falle von schwerwiegenden Fehlern wird der Stacktrace einer passenden Java-Exception ausgegeben. Die Fehlermeldung solcher Exceptions kann für erfahrene Anwender:innen aufschlussreich sein. Unter Verweis auf die aufgestellten Anforderungen wird im Rahmen dieser Arbeit aber darauf verzichtet, solche Fehlerfälle zu behandeln. Daher bekommen Anwender des Web-Editors diese Java-Exceptions nicht angezeigt, sondern können nur feststellen, dass der Simulationsprozess beendet wurde.

Der Fall, dass die Simulation einen Endzustand erreicht hat, ist für Benutzer grundsätzlich interessant. In diesem Fall lässt sich keine weitere Transition schalten und die Simulation kann als abgeschlossen betrachtet werden. Allerdings gibt der Renew-Simulator hierüber keine ausdrücklich Auskunft mittels Protokolleintrag. Stattdessen kann nur durch das Ausbleiben weiterer Protokolleinträge implizit darauf geschlossen werden, dass keine weiteren Simulationsschritte durchführbar sind. Ohne Renew diesbezüglich zu erweitern, kann auch der Web-Editor den erfolgreichen Abschluss einer Simulation nicht ausdrücklich signalisieren. Stattdessen kann nur das Stehenbleiben des Zeitzählers als Hinweis für die Benutzer:innen verwendet werden.

22. Web-Schnittstelle des Simulators

In diesem Kapitel wird die Schnittstelle zur Bereitstellung des Simulationszustands an eine Client-Anwendung behandelt. Um den aktuellen Zustand einer Simulation in der GUI darzustellen, muss der auf dem Editor-Server akkumulierte Zustand an den Client übertragen werden. Damit Benutzer Einfluss auf den Simulationsprozess nehmen können, muss der Client Steuerungsbefehle über den Server an den Simulator schicken können.

Für die Abfrage von Netzinstanzen vom Server durch den Client wird eine HTTP-Schnittstelle angeboten. Über eine WebSocket-Schnittstelle kann der Server die Clients aber auch proaktiv über geschaltete Transitionen und geänderte Markierungen von Plätzen informieren.

22.1. HTTP-Schnittstelle

Die für den Simulator entwickelte Schnittstelle entspricht derselben Struktur, die auch in Kapitel 13 „Web-Schnittstelle des Editor Servers“ für die Bereitstellung der Dokumenteninhalte vorgestellt wurde. Jeder laufenden und jeder abgeschlossenen Simulation ist eine URL zugeordnet. Über die URL kann per HTTP der aktuelle Zustand der Simulation abgefragt werden. Der Zustand wird als JSON-Struktur bereitgestellt und enthält Informationen darüber, welche Instanzen von welchen Netzen im Simulationsprozess erzeugt wurden.

Für jede Netzinstanz kann über eine eigene URL die aktuelle Belegung der Marken und die zuletzt geschaltete Transition abgefragt werden. Diese Informationen werden ebenfalls in Form einer JSON-Struktur bereitgestellt.

Außerdem lässt sich für jedes Netz auch das zugehörige Dokument vom Editor-Server abfragen. Das Dokument enthält alle Informationen, die nötig sind, um die Struktur eines Netzes grafisch darzustellen.

Tabelle 22.1 zeigt die HTTP-Adressen über die der Zustand von Simulationen abgefragt und die Simulationen gesteuert werden können. Das *:id*-Fragment in den Pfaden steht dabei jeweils für die UUID einer jeweiligen Simulation. Der Server antwortet auf jede Anfrage an diese Endpunkte mit einem JSON-Dokument, welches die passenden Informationen und gegebenenfalls URLs zu anderen Endpunkten enthält.

Die exakte Struktur der JSON-Antworten wird in diesem schriftlichen Teil dieser Arbeit nicht vollständig dokumentiert, sondern kann der konkreten Implementierung entnommen werden. Ein Beispiel für eine Antwort auf die Abfrage des Zustands einer einzelnen Netzinstanz ist aber im Listing 22.1 dargestellt. Zu erkennen ist, welche Marken auf welchen Plätzen in dieser Netzinstanz liegen (`tokens"`), welche Transition zuletzt geschaltet hat (`firings"`), über welche Adresse zusätzliche Informationen über die Simulation ab-

gefragt werden können (`links"`) und zu welchem Shadownet diese Netzinstanz gehört (`"7ebe1cae-3a0b"`). Außerdem ist ein `topic` angegeben, über welches per WebSocket-Verbindung Änderungen am Zustand der Netzinstanz empfangen werden können.

```
{
  "id": "6dc5d1fb-cbed",
  "links": {
    "simulation": {
      "href": "http://localhost:4000/api/simulations/500e8bb7-a6f7-430a"
    }
  },
  "href": "http://localhost:4000/api/simulations/500e8bb7-a6f7-430a/instance/
      foo/0",
  "topic": "redux_net_instance:ee95ed06-1de0-4427-93f3-12e1f4d52c4d",
  "content": {
    "id": "6dc5d1fb-cbed",
    "label": "foo[0]",
    "tokens": [
      {
        "id": "ee751992-3ccb",
        "value": "bar[10]",
        "place_id": "813f5756-2e17"
      }
    ],
    "firings": [
      {
        "id": "1243c1bc-b80e",
        "timestep": 7,
        "transition_id": "8d94e4ff-434b"
      }
    ],
    "integer_id": 0,
    "shadow_net_id": "7ebe1cae-3a0b"
  }
}
```

Quelltext 22.1: Antwort auf Abfrage des Simulationszustands einer Netzinstanz als JSON-Struktur

22.2. WebSocket-Schnittstelle

Wie schon in Kapitel 13 „Web-Schnittstelle des Editor Servers“ beschrieben, kann über das HTTP-Protokoll durch den Client nur proaktiv ein Zustand abgefragt werden. Im Laufe einer Simulation ändert sich die Menge an aktiven Netzinstanzen und die Belegung der Marken auf den Plätzen aber ständig. Daher muss der Server in der Lage sein, alle Clients nach jedem verarbeiteten Simulationsschritt über den neuen Zustand

Methode	Pfad	Beschreibung
GET	`/simul`	Liste aller Simulation
GET	`/simul/:id`	Aktueller Simulationszustand
POST	`/simul`	Simulation anlegen
PUT	`/simul/:id`	Simulation ändern
DELETE	`/simul/:id`	Simulation löschen
GET	`/simul/:id/inst/:name/:num`	Status einzelner Netzinstanz
POST	`/simul/:id/step`	Zum nächsten Zeitschritt schalten
DELETE	`/simul/:id/process`	Simulation beenden
GET	`/sns/:id`	Alle zum SNS gehörenden Netze
GET	`/sns/:id/download`	SNS-Datei herunterladen

Tabelle 22.1.: Endpunkte der HTTP-Schnittstelle zur Simulatorsteuerung

zu informieren. Hierfür wird genau wie für die Bearbeitung der Dokumente in Kapitel 13 „Web-Schnittstelle des Editor Servers“ das WebSocket-Protokoll in Kombination mit Phoenix Channels und dem JSON-Patch-Format verwendet.

Ein Client, der den Zustand einer Simulation als JSON per HTTP abgefragt hat, kann eine WebSocketVerbindung aufbauen, um zukünftige Änderungen des Simulationszustandes in Form von JSON-Patch-Anweisungen zu erhalten. Für jede Netzinstanz wird ein eigener Kommunikationskanal bereitgestellt, sodass ein Client nur Zustandsänderungen von den Netzinstanzen verarbeiten muss, die aktuell dargestellt werden.

Des weiteren können über die WebSocket-Verbindung auch Befehle zum Steuern der Simulation an den Server geschickt werden. Diese sind aber auf das Pausieren und Fortsetzen der Simulation beschränkt, da, wie zuvor erklärt, der Renew-Simulator keine weiteren Möglichkeiten anbietet. Tabelle 22.2 enthält eine Übersicht der Befehle, die über die WebSocket-Verbindung an den Server geschickt werden können, um die Simulation zu steuern.

Listing 22.3 zeigt zwei JSON-Patches, die nach dem Schalten einer Transition vom Server an die Clients geschickt wurden. Der eine Patch in Zeile 4 bis Zeile 9 erhöht den Zeitstempel der Simulation auf 3. Der andere Patch von Zeile 13 bis Zeile 33 ändert die Markenbelegung und markiert, welche Transition zuletzt geschaltet hat.

In Listing 22.2 wird die Verwendung der WebSocket-Schnittstelle in JavaScript demonstriert. In Zeile 3 bis Zeile 5 wird eine WebSocket-Verbindung über das Phoenix-Channel-Protokoll aufgebaut. Zeile 6 wählt sich über diese WebSocket-Verbindung in den Kommunikationskanal einer einzelnen Simulation ein. Zeile 7 bis Zeile 8 empfangen den aktuellen Simulationszustand und Zeile 10 sendet den `step`-Befehl, damit der nächste Zeitschritt der Simulation berechnet wird.

Der Funktionsumfang vom Simulator des WebEditors in dieser Arbeit ist durch die von Renew bereitgestellte Schnittstelle begrenzt. Die Auflistung von aktiven Transitionen oder das Schalten von ausgewählten Transitionen wäre denkbar, wenn Renew die entsprechende Schnittstelle bietet.

```javascript
import { Socket } from "phoenix";
import { LiveState } from "./livestate";
const socket = new Socket("https://api.webeditor.example/socket", {
  params: { token: cookie.auth_token }
});
const livestate = new LiveState(socket, { topic: "simulation:500e8bb7-a6f7" });
livestate.subscribe((currentState) => {
  console.log(currentState)
})
livestate.sendAction("step")
```

Quelltext 22.2: Verwendung der WebSocket-Schnittstelle zur Steuerung der Simulation in JavaScript

```json
{
  "redux_simulation:7dd61a32-5219-4d77-8477-fc0523db7501": {
    "patch": [
      {
        "op": "replace", "path": "/timestep",
        "value": 3
      }
    ]
  },
  "redux_net_instance:f1daf464-b796-4b4e-83f6-18488e8517a7": {
    "patch": [
      {
        "op": "replace", "path": "/firings/0/transition_id",
        "value": "a3991ad6-2edd-4145-ae54-b05b180c55b8"
      },
      {
        "op": "replace", "path": "/firings/0/timestep",
        "value": 3
      },
      {
        "op": "replace", "path": "/firings/0/id",
        "value": "1e7f05d5-9e10-49b1-b3b8-acd5b51ab686"
      },
      {
        "op": "remove", "path": "/tokens/0"
      }
    ]
  }
}
```

Quelltext 22.3: JSON-Patches zum aktualisieren des Simulationszustands nach Schalten einer Transition

Befehl	Beschreibung
`step`	Simulation zum nächsten Zeitschritt schalten
`play`	Simulation automatisch abspielen
`pause`	Automatisches Abspielen pausieren
`terminate`	Simulation beenden
`init`	Simulation zurücksetzen

Tabelle 22.2.: Steuerungsbefehle der WebSocket-Schnittstelle des Simulators.

23. Darstellung des Simulationszustandes

Dieses Kapitel behandelt die grafische Darstellung einer Simulation. Hierfür können die gleichen Techniken verwendet werden, die auch für die Darstellung von Dokumenten in Kapitel 15 „Grafische Darstellung“ behandelt wurden.

Netzinstanzen und Markenbelegungen können genau wie die Dokumente als Vektorgrafik dargestellt werden. Außerdem kann der Simulationsprozess ebenfalls kollaborativ beobachtet und gesteuert werden.

23.1. Darstellung als Vektorgrafik

Der im Webbrowser ausgeführte Editor-Client kann den Zustand einer Simulation über die in Kapitel 22 „Web-Schnittstelle des Simulators“ vorgestellte Schnittstelle abfragen. Außerdem können die zur Simulation gehörenden Dokumente ebenfalls vom Server abgerufen und, wie in Kapitel 15 „Grafische Darstellung“ behandelt, dargestellt werden. Über die eindeutige Benennung von Transitionen, Plätzen und Netzen können all durch den Simulationszustand ausgedrückten Markenbelegungen den zugehörigen Plätzen und deren Positionen in der grafischen Darstellung zugeordnet werden.

Mittels der in Kapitel 17 „Kamera-Navigation“ erarbeiteten Kameranavigation lässt sich auch während der Simulation in einem Dokument navigieren. Eine Simulation kann aber auch aus mehreren Netzen und mehreren Instanzen dieser Netze entstehen. Entsprechend kann in der grafischen Darstellung einer Simulation nicht nur in einem Dokument, sondern zwischen mehreren Dokumenten navigiert werden. Hierfür kann die Listenansicht aller Netzinstanzen verwendet werden. Abb. 23.1 zeigt die Simulationsansicht der Benutzeroberfläche des Web-Editors. Auf der rechten Seite ist die Listendarstellung der erzeugten Netzinstanzen zu erkennen. Der unterste Eintrag in der Liste ist die aktuell ausgewählte Netzinstanz, die auf der Zeichenfläche dargestellt wird. Oben links befinden sich die Schaltflächen zur Steuerung der Simulation und die Anzeige der aktuellen Simulationszeit.

Die Referenznetzsemantik von Renew erlaubt aber auch, dass Netzinstanzen sich selbst in Form von Marken auf einem Platz einer Netzinstanz befinden. Im Editor-Client werden diese Marken unterstrichen als Hyperlinks dargestellt. Ein Klick auf die jeweilige Marke navigiert die Ansicht in die zugehörige Netzinstanz. Unten in Abb. 23.1 ist zu erkennen, dass 3 Instanzen von `subnet` auf dem unteren Platz liegen. Diese sind jeweils unterstrichen dargestellt und lassen sich anklicken.

Ebenfalls in Abb. 23.1 zu erkennen ist, dass auf dem oberen linken Platz eine dicke große Eins dargestellt wird. Diese große Schreibweise gibt die Kardinalität, als die Gesamtzahl der Marken auf dem Platz, an. Per Mausklick auf einen Platz lässt sich, genau wie in

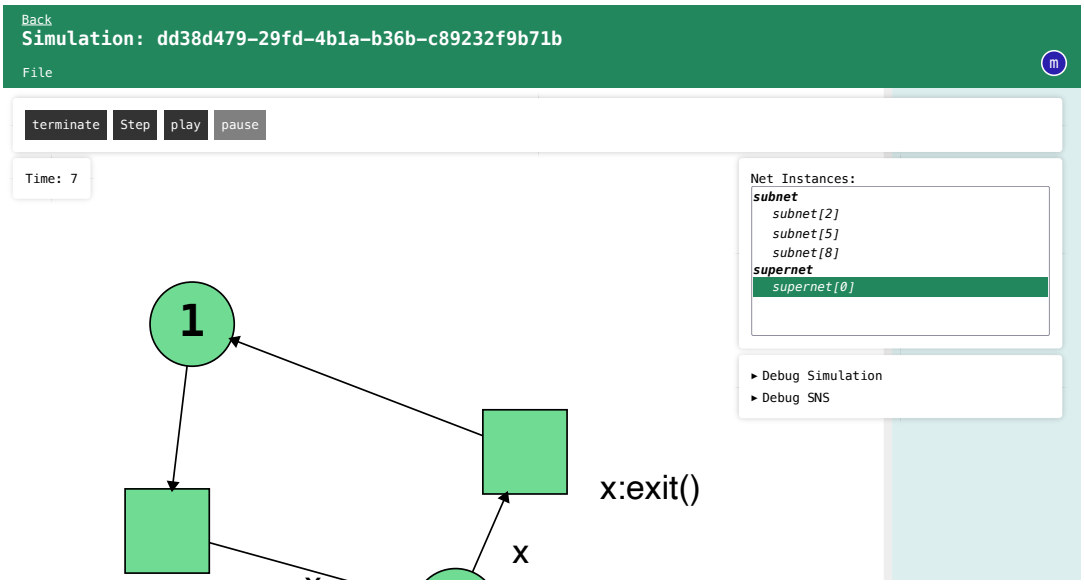


Abbildung 23.1.: Screenshot der Simulationsansicht des Web-Editors

Renew, zwischen der Darstellung der Kardinalität und der Auflistung der einzelnen Marken wechseln.

In der Benutzeroberfläche von Renew leuchten die schaltenden Transitionen während der Simulation kurz auf. Diese Darstellungsform wurde auch für die Darstellung von Simulationen im Web-Editor übernommen. Über den vom Server mitgeteilten aktuellen Simulationszustand hat der Client Kenntnis über die zuletzt geschaltete Transition. Diese wird mittels einer SVG-Animation farblich hervorgehoben und aufleuchten lassen. In Darstellung 23.2 ist zu erkennen, wie die schaltende Transition in hellgrün umrandet hervorgehoben wird.

23.2. Kollaborative Simulation

Aus der gewählten Architektur ergeben sich für die Simulation von Netzen im Web-Editor die gleichen Möglichkeiten zur Kollaboration wie für das Bearbeiten von Dokumenten. Der Renew-Simulator wird serverseitig ausgeführt. Der aktuelle Simulationszustand ist zentral auf dem Server gespeichert. Alle Änderungen am Zustand werden durch den Server verwaltet.

Seitens des Clients wird der aktuelle Zustand einer Simulation abgefragt und in eine SVG Darstellung umgewandelt. Steuerbefehle zum Pausieren und Fortsetzen einer Simulation können an den Server gesendet werden. Die jeweilige Auswirkung ist für alle Clients sichtbar.

In Kapitel 19 „Kollaboratives Arbeiten“ wurden verschiedene kollaborative Werkzeuge zum gemeinsamen Arbeiten in einem Dokument diskutiert — etwa die Darstellung farbiger Mauszeiger anderer Benutzer. Diese kollaborativen Werkzeuge lassen sich im Prinzip auch für die Darstellung von Simulationen umsetzen. Aus Zeitgründen wurde in dieser Arbeit aber darauf verzichtet.

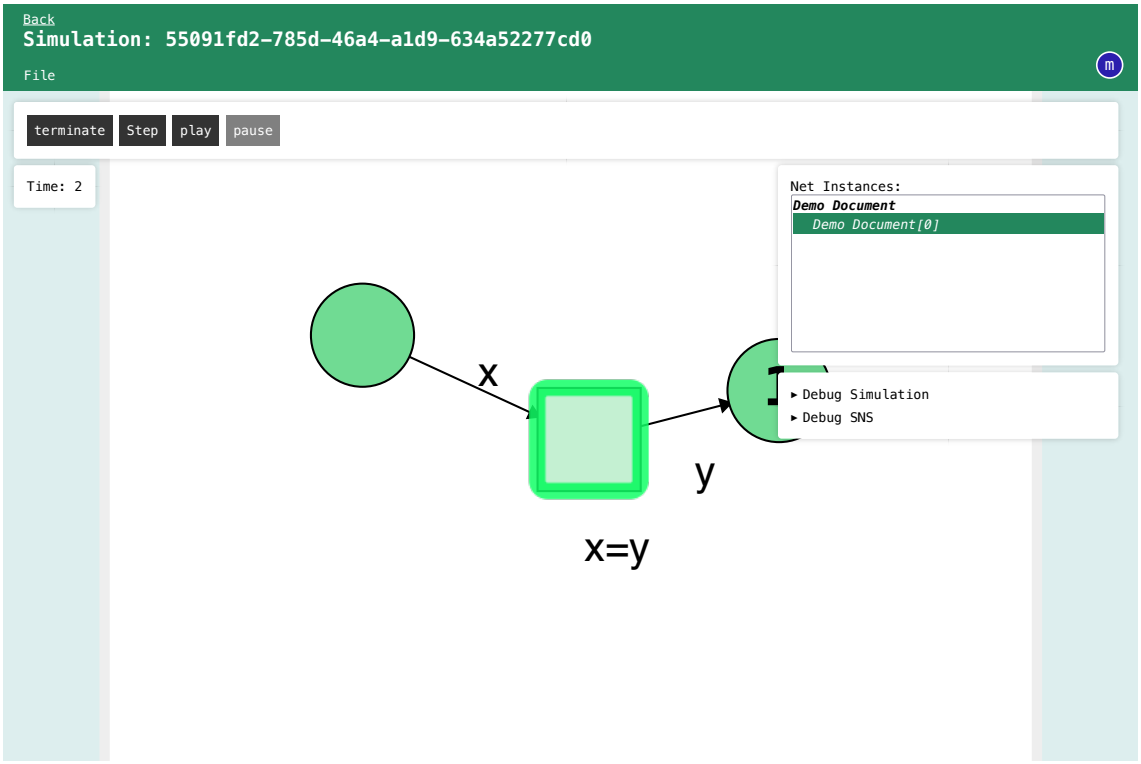


Abbildung 23.2.: Darstellung einer schaltenden Transition in der Simulationsansicht.

Teil VI.

Evaluation und Diskussion

24. Zusammenfassung der Architektur

In bisherigen Kapiteln wurden die einzelnen Komponenten, aus denen sich der in dieser Arbeit entwickelte Web-Editor zusammensetzt, entwickelt und vorgestellt. Bevor in den nächsten beiden Kapiteln das Ergebnis diskutiert und die Erkenntnisse ausgewertet werden, wird im Folgenden ein zusammenfassender Überblick über die gesamte Architektur des Web-Editors gegeben.

In Abb. 24.1 wird der Zusammenhang der einzelnen Komponenten grafisch dargestellt. Auf der linken Seite ist in gelb der in JavaScript entwickelte Editor-Client dargestellt. Auf der rechten Seite ist in lila der in Elixir entwickelte Editor-Server dargestellt. Die Kommunikation zwischen Client und Server findet über HTTP- und WebSocket-Verbindungen statt. Am unteren Rand sind die Abhängigkeiten zu fremden Softwarebibliotheken zu erkennen. Im Editor-Client wird vor allem Svelte und die `partial.lenses`-Bibliothek verwendet, um ein reaktives Datenmodell zu konstruieren. Für die Netzwerkkommunikation und die Darstellung der Benutzeroberfläche werden die HTML5-Schnittstellen des Web-Browsers verwendet.

Der Client lässt sich konzeptionell in verschiedene Module einteilen. Nur das mit dem Asterisk ($*$) markierte Modul wurde als eigenständiges Softwarepaket entwickelt. Die mit einem Stern ($\star$) markierten Module bieten sich aber auch zur Extraktion in eigene Softwarepakete an, da sie einen Aufgabenbereich mit klar abgrenzbarer Schnittstelle abdecken.

Im Editor-Server wurde das Phoenix Web Framework für die Umsetzung der Web-Schnittstelle verwendet. Viele der serverseitigen Module wurden als eigene Softwarepakete veröffentlicht. Diese wurden in der Darstellung mit einem Asterisk ($*$) markiert. Die mit einem Stern ($\star$) markierte Simulator-Komponente stellt die Schnittstelle zum Java Renew-Simulator dar. Sie wurde aus Zeitgründen nicht als eigenes Softwarepaket veröffentlicht, obwohl sich dies angeboten hätte. Im Editor-Server wird SQLite zur Speicherung von Dokumenten und Simulationsergebnissen verwendet. Alternativ lassen sich aber auch PostgreSQL oder MySQL als relationale Datenbank verwenden.

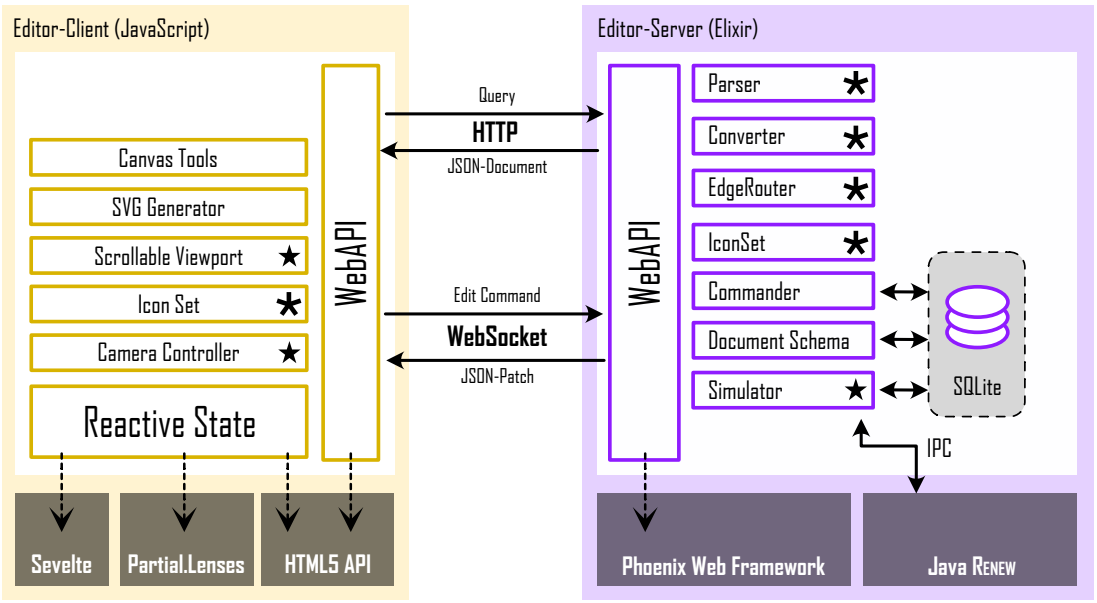


Abbildung 24.1.: Übersicht der Komponenten des Web-Editors. Die mit mit einem Asterisk (∗) markierten Module sind als eigenständige Softwarepaket veröffentlicht. Die mit einem Stern (⋆) markierten Module, könnten als eigenständige Pakete extrahiert werden.

25. Diskussion der erfüllten Anforderungen

In Kapitel 4 „Anforderungen“ wurden acht zentrale Anforderungen aufgestellt, die der Web-Editor erfüllen solle. In diesem Kapitel wird diskutiert, inwiefern die aufgestellten Anforderungen im Rahmen dieser Arbeit erfüllt werden konnten. Dafür wird im Folgenden jede Anforderung noch einmal genannt und dann begründet, dass der entwickelte Web-Editor sie jeweils erfüllt.

Die Anforderungen waren das Lesen und Schreiben von .rnw-Dateien, die Benutzbarkeit auf Desktop- und Mobilgeräten, kollaboratives Arbeiten, Kompatibilität mit hierarchischen Dokumenten, eine Undo/Redo-Historie, Navigation innerhalb eines Dokuments und die Simulation von Referenznetzen.

25.1. Renew Dateien lesen und schreiben

Das Einlesen von bestehenden Renew Dateien in den Web-Editor konnte erfolgreich umgesetzt werden. Das für den Web-Editor verwendete relationale Datenmodell konnte besonders darauf ausgelegt werden, dass das Einlesen bestehender rnw-Dateien verlustfrei möglich ist. Die für das Einlesen entwickelten Module wurden als eigenständige Elixir-Pakete veröffentlicht und lassen sich somit auch unabhängig vom Web-Editor verwenden.

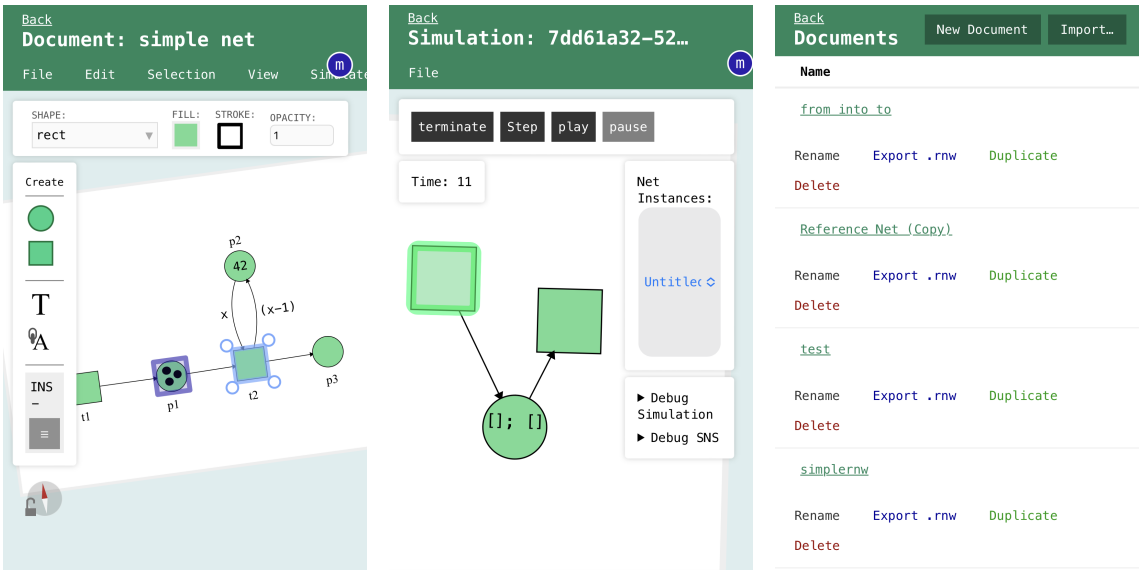
Das Exportieren von Dokumenten aus dem Web-Editor als Renew Dateien konnte grundsätzlich auch umgesetzt werden. Es funktioniert aber nur für eine begrenzte Teilmenge aller Dokumente. Das liegt daran, dass sich im Datenmodell des Web-Editors Dokumente darstellen lassen, für die es keine offensichtliche und eindeutige Abbildung in die von Renew erwartete Struktur gibt.

Für einfache im Web-Editor gezeichnete Netze stellt dies aber kein Problem dar. Alle einfachen Referenznetze, die im Web-Editor gezeichnet werden, lassen sich in gültige rnw-Datei umwandeln. Der Exportprozess lässt sich zukünftig weiter ausarbeiten, um die Menge an exportierbaren Dokumenten zu vergrößern.

25.2. Benutzbarkeit auf Desktop- und auf Mobilgeräten

Eine grafische Benutzeroberfläche für Desktop und Mobilgeräte konnte erfolgreich umgesetzt werden. Die im Webbrowser ausgeführte Client-Anwendung lässt sich in allen gängigen Browsern (Google Chrome, Mozilla Firefox, Microsoft Edge) verwenden. Sowohl per Maus und Tastatur als auch per Touchscreen sind alle Funktionen in der Benutzeroberfläche einfach erreichbar und zu bedienen.

Die gewählte Darstellung von Dokumenten als SVG hat sich auch auf Mobilgeräten als geeignetes Format für mittelgroße Dokumente bewährt. Da die grafische Darstellung der



(a) Editor (b) Simulator (c) Listenansicht

Abbildung 25.1.: Screenshots des Web-Editor-Clients auf einem Mobilgerät

Dokumente im Client von der Bearbeitungslogik des Servers entkoppelt ist, lassen sich aber auch alternative Client-Anwendungen für zusätzliche Endgeräte entwickeln.

Abb. 25.1 zeigt drei Bildschirmaufnahmen des Web-Editors auf einem Mobilgerät. In Abb. 25.1(a) ist zu erkennen, dass ein Dokument auf einem kleinen Bildschirm dargestellt werden kann und die Schaltflächen für die Verwendung per Fingergesten sinnvoll angeordnet sind. Abb. 25.1(b) zeigt auch die sinnvolle Darstellung der Simulationsansicht auf Mobilgeräten. Auch die in Abb. 25.1(c) gezeigte Listenansicht passt sich an die kleine Bildschirmgröße an.

25.3. Kollaboratives Arbeiten

Das kollaborative Arbeiten an Dokumenten im Web-Editor ist möglich. Mehrere Benutzer:innen können ein gemeinsames Dokument im Web-Editor auf ihrem jeweiligen Endgerät öffnen und zeitgleich verschiedene Änderungen vornehmen. Eigene und von anderen Benutzerinnen und Benutzern durchgeführte Änderungen sind ohne stark wahrnehmbare Latenz im Dokument zu beobachten.

Auch Simulationen lassen sich kollaborativ im Web-Editor durchführen. Mehrere Benutzer:innen können gemeinsam den Fortschritt einer auf dem Server durchgeführten Simulation beobachten und die Simulation mittels verfügbarer Steuerbefehle beeinflussen.

Außerdem konnten verschiedene Werkzeuge zum Unterstützen der kollaborativen Arbeit umgesetzt werden. Während der gemeinsamen Arbeit in einem Dokument werden die Mauszeigerpositionen anderer Benutzer:innen farbig dargestellt. Von anderen Benutzerinnen und Benutzern ausgewählte Elemente werden farbig hervorgehoben. Eine Liste aller anderen aktiven Benutzer:innen wird dargestellt.

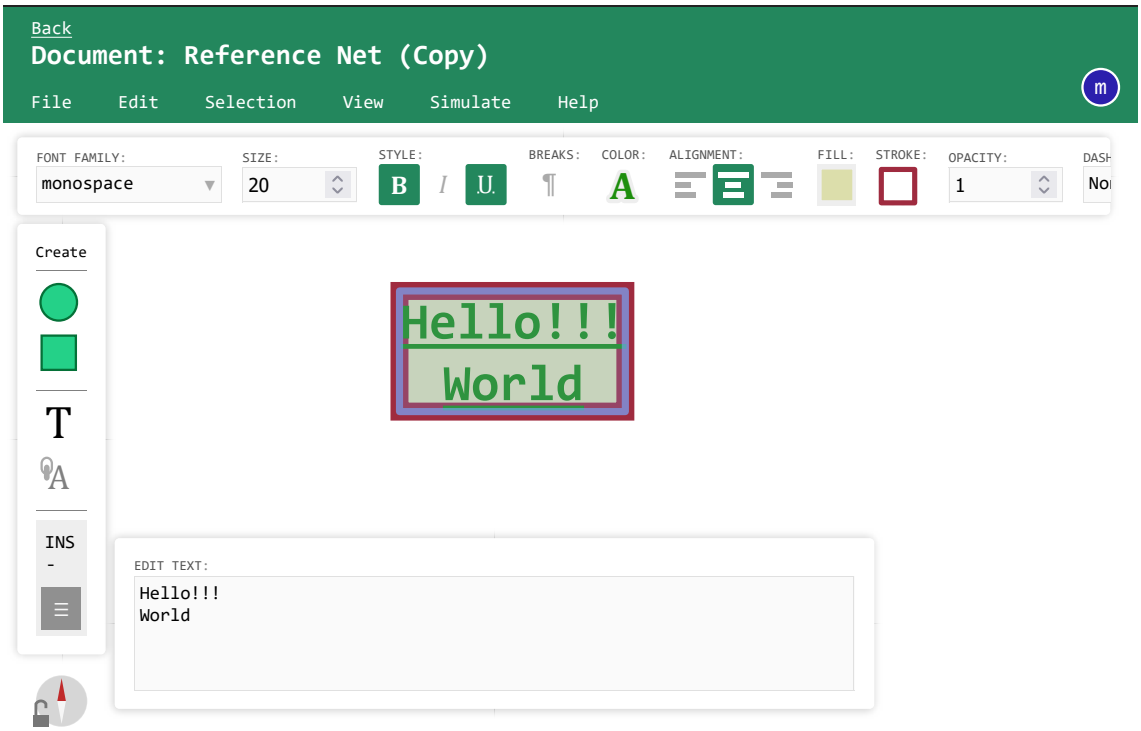


Abbildung 25.2.: Screenshot der Kontrollfelder zum Bearbeiten von Textattributen

25.4. Elemente im Dokument Bearbeiten

Der Web-Editor erlaubt das Erstellen und Bearbeiten von verschiedenen Elementen und deren Attributen. Zum Zeichnen von Netzen können Plätze und Transitionen auf der Zeichenfläche platziert werden. Die Positionen von Elementen können verändert werden. Elemente können durch Kanten miteinander verbunden werden. Textanschriften lassen sich in einem Dokument platzieren und mit anderen Elementen verknüpfen. Darstellungsoptionen wie Farbe, Strichstärke und Schriftart können über zugehörige Schaltflächen manipuliert werden. Elemente lassen sich auch wieder löschen. Dabei werden zusammenhängende Elemente gemeinsam gelöscht, sodass die Konsistenz eines Dokuments stets aufrechterhalten wird.

In Abb. 25.2 sind die Werkzeuge zum Bearbeiten von Textelementen zu erkennen. Mittig im Bild ist das ausgewählte und formatierte Textelement zu sehen. Am unteren Rand ist ein mehrzeiliges Eingabefeld platziert, über welches sich der Textinhalt bearbeiten lässt. Die horizontale Werkzeugleiste am oberen Rand der Zeichenfläche bietet Schaltflächen zum Bearbeiten der Darstellungsattribute. Hierüber können die Schriftart, Schriftgröße, Textfarbe, Textausrichtung, Deckkraft, Hintergrund- und Rahmenfarbe des ausgewählten Elements verändert werden.

Abb. 25.3 zeigt die Bearbeitungsoptionen von Kanten und Linienzügen. Mittig auf der Zeichenfläche ist eine ausgewählte gestrichelte Linie mit zwei verschiedenen Pfeilspitzen dargestellt. Entlang der Linie befinden sich blau umkreiste Wegpunkte. Diese können per Maus oder Finger verschoben werden, um den Verlauf der Linie zu steuern. In der

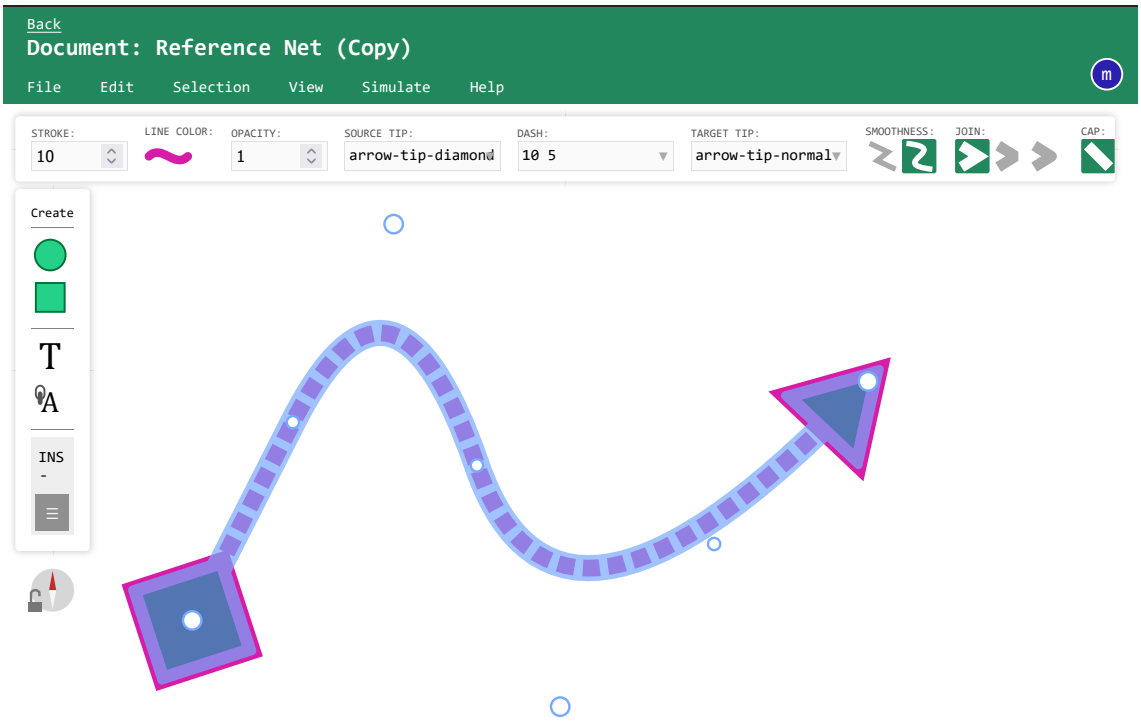


Abbildung 25.3.: Screenshot der Kontrollfelder zum Bearbeiten von Kantenattributen

horizontalen Werkzeugleiste am oberen Rand können die Darstellungsoptionen über verschiedene Schaltflächen gesteuert werden. Die Dicke, Farbe, Deckkraft und Strichelung der Linie lässt sich konfigurieren. Außerdem kann für den Start- und für den Endpunkt jeweils ein Pfeilspitzensymbol gewählt werden. Es kann zwischen Bézier- und linearer Interpolation gewählt werden, um entweder weiche oder eckige Linienzüge zu zeichnen.

Abb. 25.4 zeigt die Werkzeugleiste zum Bearbeiten der Knotendarstellung. Zu sehen ist ein ausgewählter Knoten in Form einer Raute mit roter Füllfarbe und lila gestrichelter Umrandung. Die Farben, Strichstärke, Deckkraft und die geometrische Form lassen sich über die jeweiligen Schaltflächen verändern.

25.5. Hierarchische Dokumente bearbeiten

Das für den Web-Editor erarbeitete Datenmodell erlaubt auch die Darstellung und das Bearbeiten von hierarchisch verschachtelten Dokumenten. Im Editor-Server wurden die dafür nötigen Datenstrukturen und Operationen umgesetzt. Im Editor-Client stehen zugehörige Werkzeuge bereit, um in der Hierarchie eines Dokuments zu navigieren und Gruppen von Elementen als Einheit zu verschieben.

Abb. 25.5 zeigt die Darstellung der Ebenenhierarchie in der Benutzeroberfläche des Editor-Clients. In der Listendarstellung sind die Elemente unterschiedlich tief eingerückt, um die Baumstruktur auszudrücken. Ein ausgewähltes Element ist in der Liste etwas dunkler in Grün hervorgehoben. Über die Kontrollkästchen in der Liste lassen sich Elemente ein-

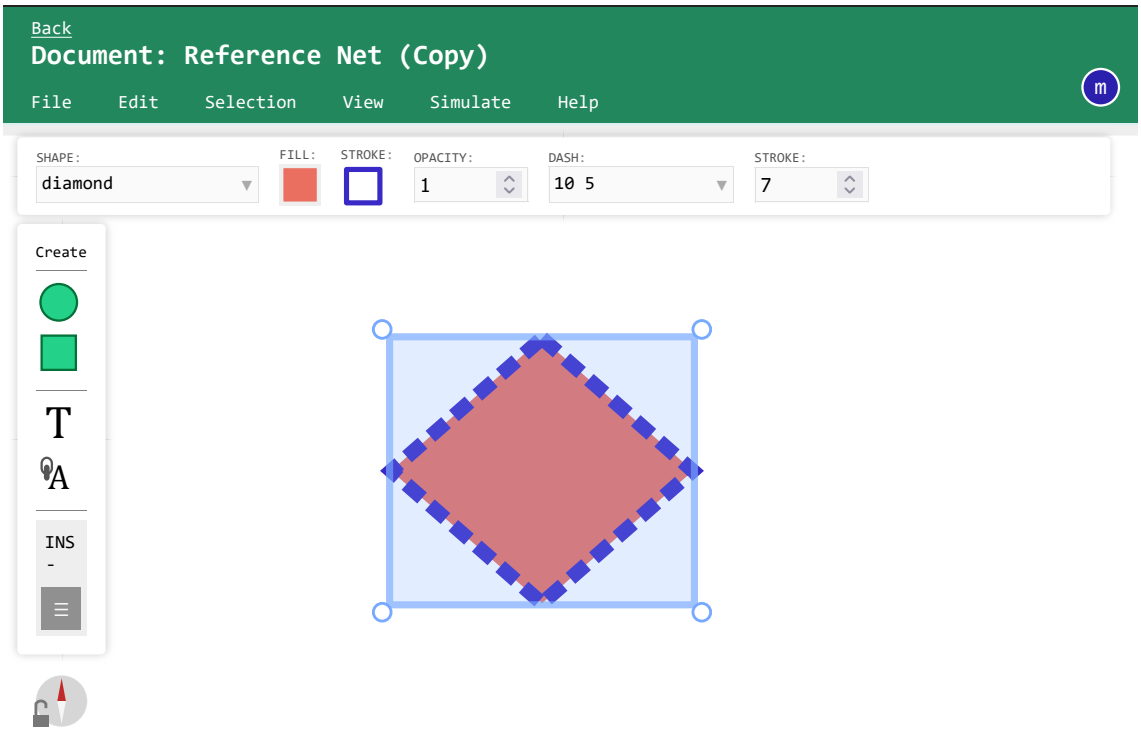


Abbildung 25.4.: Screenshot der Kontrollfelder zum Bearbeiten von Boxattributen

und ausblenden. Die dunklen Quadrate mit den drei hellen Strichen lassen sich mit der Maus greifen, um das jeweilige Element in der Liste zu verschieben. Darüber kann sowohl die Reihenfolge als auch die Verschachtlung der Elemente verändert werden. Unterhalb der Liste befinden sich drei weitere Schaltflächen, über die sich das ausgewählte Element bearbeiten lässt. Das gewählte *Interface* bestimmt, welche Verbindungspunkte ein Element für eingehende und ausgehende Kantenverbindungen bereitstellt. Das Feld *Semantic Tag* bestimmt für den Renew-Export, in der Rolle welcher Java-Klasse das Element exportiert werden soll.

Abb. 25.6 zeigt eine Auswahl an geometrischen Formen, die mit einer Menge an Verbindungspunkten kombiniert wurden. Geometrische Formen und Verbindungspunktmenge lassen sich durch Benutzer:innen beliebig kombinieren. Eine Kante kann zwischen genau zwei beliebigen Verbindungspunkten gezogen werden. Im Datenmodell ist auch die Verknüpfung einer Kante mit nur einem Verbindungspunkt erlaubt. In der Benutzeroberfläche ist hierfür aber keine Interaktion definiert.

25.6. Undo/Redo-Historie

Alle an einem Dokument im Web-Editor vorgenommene Änderungen werden in einer Historie aufgezeichnet. Durch die Wiederherstellung alter Zwischenstände lassen sich Änderungen rückgängig machen. Die Aufzeichnung der Historie funktioniert auch beim kollaborativen Arbeiten.

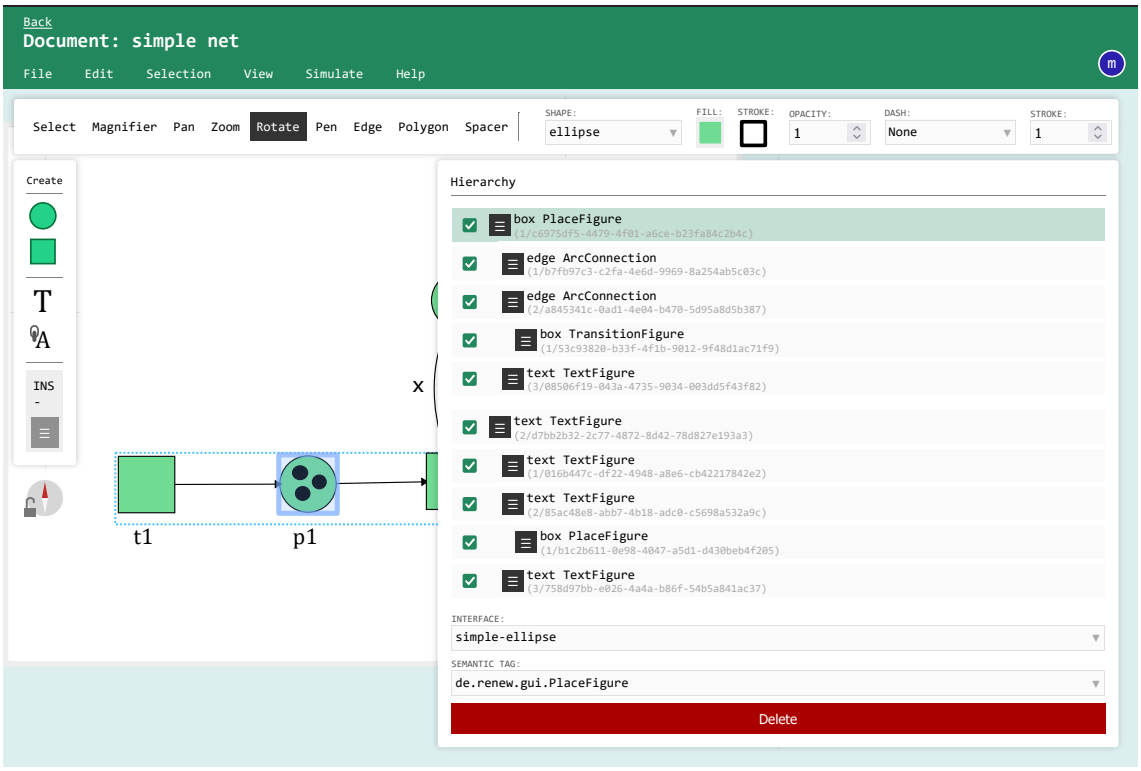


Abbildung 25.5.: Screenshot der Ebenenhierarchie im Web-Editor-Client

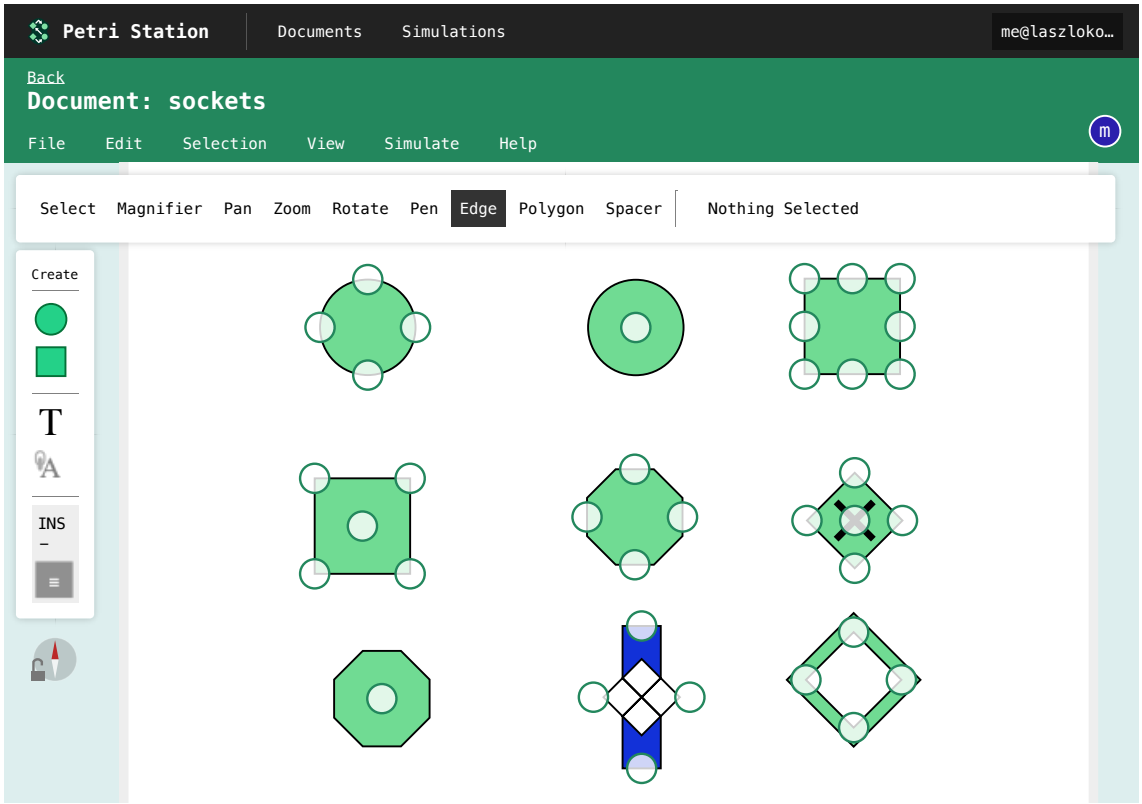


Abbildung 25.6.: Im Web-Editor definierte Verbindungspunkte von Knoten für die Verbindung durch Kanten

25.7. Grafische Navigation im Dokument

Ein besonderer Fokus wurde auf die Entwicklung der Kamera-Navigation gelegt. Sowohl per Maus als auch per Zweifingergeste lässt sich die Ansicht in einem Dokument zoomen, drehen und verschieben. Über zusätzliche Werkzeuge kann auch einhändig oder mit nur einem Finger im Dokument navigiert werden. Auch per Scrollbalken lässt sich flüssig im Dokument navigieren. Eine Minimap für die Navigation und den Überblick in großen Dokumenten konnte ebenfalls umgesetzt werden.

25.8. Simulation von Referenznetzen

Der Renew Simulator konnte erfolgreich an Web-Editor angeschlossen werden. Im Web-Editor gezeichnete Netze können mit vollwertiger Referenznetzsemantik simuliert werden. Der aktuelle Zustand einer Simulation kann im Web-Editor grafisch auf Desktop- und mobilen Endgeräten dargestellt werden.

Abb. 25.7 zeigt die Simulationsansicht der Web-Editors. Über die vier Schaltflächen oben links lässt sich die Simulation beenden, schrittweise weiterschalten, automatisch abspielen und pausieren. Oben links wird die aktuelle Simulationszeit angezeigt. Auf der rechten Seite werden alle durch die Simulation erzeugten Netzinstanzen gruppiert nach ihrer Netzstruktur aufgelistet. Die ausgewählte Instanz ist farbig markiert und wird grafisch dargestellt. In der grafischen Darstellung wird auf den einzelnen Plätzen die Markenbelegung angezeigt. Auf dem oberen linken Platz wird die Anzahl der Marken in groß und fett angezeigt. Auf dem unteren rechten Platz werden die einzelnen Marken auf dem Platz aufgelistet. Per Mausklick auf einen einzelnen Platz lässt sich zwischen den beiden Darstellungsformen wechseln.

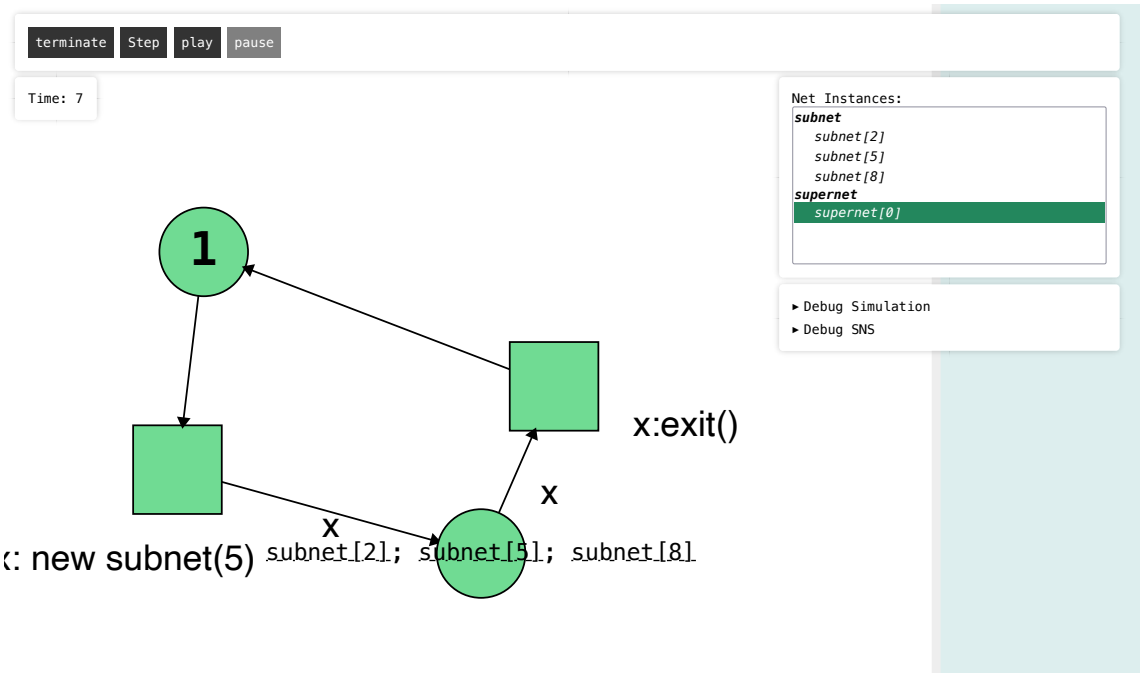


Abbildung 25.7.: Screenshot der der Benutzeroberfläche zur Simulation von Referenznetzen

26. Diskussion der gewählten Strategie

In diesem Kapitel wird diskutiert, inwiefern die gewählten Strategien sich auf die Umsetzung dieser Arbeit ausgewirkt haben. Allgemein haben die Strategien das erfolgreiche Umsetzen eines großen Funktionsumfangs ermöglicht. Allerdings wurde die Umsetzung einiger weniger Funktionen durch einige Strategien auch verhindert.

Die Schwerpunkte der gewählten Strategie waren der Verzicht darauf, Änderungen an Renew vorzunehmen, den Import und Export von Renew-Dokumenten zu erlauben, nur über die Befehlszeilenschnittstelle mit Renew zu kommunizieren, auf eine Client/Server-Architektur zu setzen, eine relationale Modellierung für die Speicherung von Dokumenten zu verwenden, und die Geschäftslogik und Benutzeroberfläche mittels funktionaler Programmierung zu modellieren.

26.1. Keine Änderungen an Java Renew

Der Vorsatz, keine Änderungen an Renew vorzunehmen, hat es ermöglicht, fokussiert nur an dem Web-Editor zu arbeiten, ohne dass eine Koordination mit dem Entwicklungsprozess von Renew nötig war. Es musste nicht auf die Behebung von Fehlern in Renew oder auf die Veröffentlichung einer neuen Version gewartet werden. Es mussten auch keine Merge-Anfragen gegen den Renew Entwicklungsstand durchgeführt werden. Außerdem hat diese Einschränkung die Menge an verfolgbaren Zielen sinnvoll eingeschränkt.

Einige Funktionen des WebEditors sind durch das gegebene Verhalten von Renew aber nicht umsetzbar gewesen. Im Web-Editor können während einer Simulation keine einzelnen Transitionen manuell geschaltet werden. Die Simulationsgeschwindigkeit muss künstlich gedrosselt werden, weil der Datendurchsatz im Ereignisprotokoll von Renew ansonsten zu groß ist. Die Robustheit des Web-Editor-Simulators ist durch das nicht eindeutige Format des Renew Simulationsprotokolls eingeschränkt.

26.2. Import und Export von rnw-Dateien

Die Wahl des rnw-Dateiformats als Schnittstelle zu Renew hat sich von Beginn der Arbeit an bewährt. Besonders das Einlesen von rnw-Dateien hat es ermöglicht, alle Funktionen des Web-Editors mit bestehenden Renew Dokumenten zu testen. Die grafische Darstellung bestehender Dokumente hat als sinnvoller Maßstab während der Entwicklung des Web-Editors gedient. Das von Renew vorgebende Datenmodell hat den potenziellen Umfang des für den Web-Editor entwickelten Datenmodells sinnvoll eingeschränkt.

Der Export von rnw-Dateien hat sich als anspruchsvoller als der Import herausgestellt. Allerdings konnte die entwickelte Exportschnittstelle erfolgreich auch für das Durchführen von Simulationen mittels Renew eingesetzt werden.

26.3. Anbindung an den Renew Simulator

Die Anbindung des Renew Simulators konnte erfolgreich mittels Befehlszeilenschnittstelle umgesetzt werden. Durch die gewählte Schnittstelle ist der im Web-Editor verfügbare Funktionsumfang des Simulators beschränkt. Dafür hat die gewählte Schnittstelle im positiven Sinne für eine klare Begrenzung des Umfangs dieser Arbeit gesorgt.

26.4. Webanwendung mittels Client/Server-Architektur

Die gewählte Client/Server-Architektur mit der HTTP und WebSocket-Schnittstelle zwischen Server und JS-Client hat für die Umsetzung des Web-Editors sehr gut funktioniert.

Serverseitig konnte ein robustes Datenmodell entwickelt werden. Dieses sorgt dafür, dass alle gespeicherten Dokumente sich stets in einem konsistenten Zustand befinden. Dank der zentralen Koordination aller Änderungsoperationen konnte eine Versionshistorie, die zentrale Simulation von Netzen und kollaborative Werkzeuge zum Bearbeiten der Dokumente problemlos umgesetzt werden.

Die grafische Benutzeroberfläche als JS Anwendung im Webbrowser konnte unabhängig gegen eine einfache Schnittstelle entwickelt werden. Es konnte gezeigt werden, wie Dokumente sich sowohl mittels WebGL als auch als SVG darstellen lassen. Die Entwicklung des Clients als eigenständige Webanwendung hat es erlaubt, dass der Web-Editor sich auf verschiedenen Endgeräten einsetzen lässt und dass sich bei Bedarf auch zusätzliche Clients für andere Geräte entwickeln lassen.

Die Verwendung der WebPlattform für die Gestaltung der Benutzeroberfläche bringt zusätzliche unerwartete Vorteile mit sich. Über die Zoom-Funktion des Webbrowsers kann die gesamte Benutzeroberfläche unabhängig vom Kamerazoom im Dokument frei skaliert werden. Die Darstellung der Dokumente als SVG ist mit der Suchfunktion von Webbrowsern kompatibel, sodass die Textelemente in einem Dokument sich durchsuchen lassen, ohne dass eine eigene Suchfunktion programmiert werden musste. Mittels User Stylesheets können Anwender auf ihrem Endgerät eigenständig die gesamte Benutzeroberfläche umgestalten [MDN24]. Da jedes Dokument sowieso als SVG angezeigt wird, lässt es sich aus dem Webbrowser auch direkt als solches exportieren, ohne dass ein Grafikexport im Web-Editor umgesetzt werden musste.

26.5. Relationale Modellierung

Beim Entwickeln der Server-Komponente des Web-Editors hat die Verwendung des relationalen Datenmodells dafür gesorgt, dass eine konsistente und fehlerfreie Bearbeitung von Dokumenten umgesetzt werden konnte. Änderungsoperationen konnten auf hoher Abstraktionsebene mittels SQL definiert werden. Alle kollaborativen Änderungen an Dokumenten und Simulationszuständen konnten transaktional umgesetzt werden, sodass sich alle Datensätze stets in einem konsistenten Zustand befinden.

Im Rahmen dieser Arbeit wurde der Einsatz von SQLite für die Umsetzung des relationalen Datenmodells um die Speicherung aller Daten motiviert und umgesetzt. Es hat sich aber gezeigt, dass der entwickelte Web-Editor sich alternativ auch in Verbindung mit MySQL oder PostgreSQL ausführen lässt. Es wurde darauf geachtet, dass alle verwendeten SQL-Sprachkonstrukte standardkonform und mit allen drei Datenbanken kompatibel sind.

26.6. Funktionale Programmierung

Für die Programmierung der Server-Komponente war der Einsatz der funktionalen Programmiersprache Elixir ein großer Erfolg. Der entwickelte Kontrollfluss für die Bearbeitung von Dokumenten ist sehr einfach und gut nachvollziehbar. Während der Entwicklung sind keine umständlich zu behebenden Programmierfehler entstanden.

Die Client-Server-Architektur hat sich besonders gut dafür angeboten, einen linearen Kontrollfluss zwischen Client, Server und Datenbank zu definieren. Dieser ließ sich problemlos im funktionalen Paradigma in Elixir unter Verwendung des Phoenix-Web-Frameworks definieren und umsetzen.

Für die Umsetzung der Client-Komponente war der Einsatz von funktionaler Programmierung erfolgreich, um ein konsistentes Modell aller Zustände der Benutzeroberfläche zu definieren. Durch den Einsatz von reaktiven Datenstrukturen in Verbindung mit dem Optik- und CalmmJS-Entwurfsmuster konnte eine konsistente Benutzeroberfläche aus einzelnen Modulen zusammengebaut werden. Im begrenzten Zeitraum dieser Arbeit konnte der Quellcode der Client-Anwendung nicht so sauber strukturiert und gegliedert werden wie der der Server-Anwendung. Allerdings ist es als ein umso stärkerer Erfolg zu betrachten, dass unter Einsatz der genannten Techniken trotz knapp bemessener Zeit ein sehr hoher Funktionsumfang fehlerfrei umgesetzt werden konnte.

26.7. Verwendete Werkzeuge

SQLite wurde zu Beginn der Arbeit als einfach zu integrierende Programmbibliothek zur Umsetzung der relationalen Modellierung gewählt. SQLite hat sich im Laufe der Arbeit als ein stabiles und leistungsstarkes System bewährt. Die Latenz beim Bearbeiten von Dokumenten ist nicht wahrnehmbar. Während der Entwicklung des Web-Editors musste kein vollständiger Datenbankserver gepflegt werden. Die automatisierten Unittests, die für den Web-Editor definiert wurden, lassen sich zügig gegen eine In-Memory-Instanz von SQLite ausführen.

Im Laufe dieser Arbeit wurde auch versucht, SQLite gegen MySQL und PostgreSQL auszutauschen. Ohne irgendwelche Optimierungen in der Konfiguration der jeweiligen Datenbank vorzunehmen, hat sich gezeigt, dass SQLite beim Ausführen von Änderungsoperationen an Dokumenten deutlich schneller ist als MySQL. Zwischen SQLite und

PostgreSQL hingegen konnte kein wahrnehmbarer Unterschied festgestellt werden. Für den Einsatz des Web-Editors in der Praxis lässt sich wahlweise SQLite, MySQL oder PostgreSQL verwenden. Hierfür wurden auch entsprechende Konfigurationen als Container vorbereitet. Beim Einsatz von SQLite konnte eine Leistungssteigerung erreicht werden, indem die Speicherung von Dokumenten und Simulationszuständen in unabhängige SQLite-Datenbanken aufgetrennt wurde.

Wie schon in Abschnitt 26.6 erklärt, hat sich der Einsatz von Elixir für die Programmierung der Server-Komponente sehr positiv ausgewirkt. Seit Beginn der Entwicklung konnte der jeweils aktuelle Entwicklungsstand der Server-Komponente bereitgestellt und im Internet zugänglich gemacht werden. Während der gesamten Zeitdauer dieser Arbeit konnten keine Systemabstürze des Editor-Server verzeichnet werden.

Das von Elixir und dem Phoenix Web Framework bereitgestellte Abstraktionslevel hat es erlaubt, den Fokus der Entwicklung auf die Umsetzung von Geschäftsanforderungen zu legen. Das Phoenix Channel Protokoll war die Grundlage für eine stabile Kommunikation zwischen Client und Server mittels WebSocket. Zustandsänderungen können sehr einfach zwischen Server und Client kommuniziert werden, ohne dass händisch ein komplexes Protokoll entworfen oder implementiert werden musste.

Die Verwendung von JavaScript in Verbindung mit Svelte hat sich im Rahmen dieser Arbeit gut geeignet, um in kurzer Zeit eine umfangreiche Benutzeroberfläche zu entwickeln. Ob Svelte im Kontrast zu alternativen Werkzeugen wie ReactJS, SolidJS oder VueJS die langfristig sinnvolle Wahl ist, lässt sich im Rahmen dieser Arbeit nicht abschließend feststellen. Dank der gewählten Client-Server-Architektur lässt sich der Client bei Bedarf aber auch unter Verwendung anderer Frameworks neu entwickeln. Der Einsatz der CalmmJS-Architektur auf Basis von reaktiven Datenstrukturen und dem Option-Entwurfsmuster hat sich bewährt.

27. Zukünftige Erweiterungen und Verbesserungen

In diesem Kapitel werden mögliche zukünftige Verbesserungsvorschläge für den Web-Editor zusammengetragen. Einige Möglichkeiten für Verbesserungen wurden in den bisherigen Kapiteln im jeweiligen Kontext schon erwähnt.

Es wird diskutiert, inwiefern die Bereitstellung des Web-Editors weiter ausgearbeitet werden kann, wie sich Dokumente langfristig persistieren und archivieren lassen, wie die Simulatoranbindung weiter ausgearbeitet werden kann, und um welche Funktionen die Benutzeroberfläche noch erweitert werden kann. Außerdem wird diskutiert, inwiefern sich der Editor auf einzelne weitere Formalismen zuschneiden ließe und inwiefern hierfür Techniken der Metamodellierung eingesetzt werden könnten. Auch das Offline-Bearbeiten von Dokumenten und das Bearbeiten von Renew-fremden Elementtypen wird diskutiert.

27.1. Bereitstellung

Im Rahmen dieser Arbeit wurde nicht ausführlich untersucht, wie sich der entwickelte Web-Editor als Service für eine große Menge Anwenderinnen bereitstellen lässt. Es wurde zwar ein rudimentäres Authentifizierungssystem eingebaut, um den Zugriff auf den Web-Editor Server durch ein Passwort zu schützen, allerdings ist die zugrunde liegende Benutzerverwaltung nicht darauf ausgelegt, unbekannte Benutzer in das System aufzunehmen und zu verwalten.

Zum einen fehlen Arbeitsabläufe zum eigenständigen Registrieren neuer Benutzer, zum Zurücksetzen von Passwörtern oder auch zum Importieren von Benutzern aus anderen Systemen. Noch wichtiger ist aber die Einschränkung von Zugriffen einzelner Benutzer auf einzelne Dokumente. In der aktuellen Umsetzung des Web-Editors können alle Benutzer auf alle Dokumente und alle Simulationen zugreifen und diese bearbeiten. Dieser Arbeit hat sich auf das Erarbeiten eines Datenmodells zum kollaborativen Bearbeiten eines einzelnen Dokuments fokussiert. Dabei wurde aber ein Datenmodell für die Verwaltung mehrerer Dokumente nicht weiter beachtet. Für die öffentliche Bereitstellung des Web-Editors muss die Server-Komponente erweitert werden, sodass Benutzerinnen jeweils nur Zugriff auf ihre eigenen Dokumente haben und diese dann optional für andere Benutzer zum gemeinsamen Bearbeiten freigeben können.

27.2. Langfristige Persistierung

Für die langfristig zentrale Bereitstellung des Web-Editors für eine große Nutzergemeinschaft ist auch zu entscheiden, in welcher Form die bearbeiteten Dokumente und Simulationen langfristig gespeichert werden sollen. Aktuell werden Dokumente nur gelöscht, wenn dies manuell durch einen Benutzer veranlasst wird. Besonders die gespeicherte Versionshistorie der einzelnen Dokumente kann auf lange Sicht aber sehr viel Speicherplatz in Anspruch nehmen.

Außerdem lassen sich aktuell alle im Web-Editor gespeicherten Dokumente nur in Form einer langen Liste darstellen. Für die langfristige Speicherung von Dokumenten wäre es aber sinnvoll, diese in Ordnern, Gruppen oder Projektmappen organisieren zu können. Auch eine Suchfunktion wäre sinnvoll.

Auch wenn die Verwendung des relationalen Datenmodells in Verbindung einer SQL-Datenbank in dieser Arbeit dazu anregt, Dokumente langfristig zu speichern, kann es auch sinnvoll sein, die langfristige Speicherung aus dem Web-Editor auszulagern. Dann ließen sich die im Web-Editor gespeicherten Dokumente als vergängliche Arbeitskopien betrachten, die außerdem des Editors archiviert werden müssen. Dafür wäre es aber nötig, dass sich Dokumente verlustfrei aus dem Web-Editor exportieren und wieder einlesen lassen.

In dieser Arbeit wurde zwar der Import und Export von rnw-Dateien behandelt. Allerdings wurde keine Schnittstelle entwickelt, um die Web-Editor erstellten Dokumente in einem eigenen Format vollständig sichern und wieder einlesen zu können. Ein entsprechendes Datenformat wäre in jedem Fall sinnvoll.

27.3. Simulation

In den Kapiteln 20 bis 23 wurde bereits auf einige Einschränkungen der von Renew bereitgestellten Schnittstelle zum Simulator hingewiesen. Durch die Erweiterung der Schnittstelle kann der im Web-Editor bereitgestellte Funktionsumfang des Simulators erweitert werden.

Das manuelle Schalten einzelner Transitionen per Mausklick oder Fingergeste kann ermöglicht werden. Eine manuelle Auswahl der verwendeten Bindungen von Variablen beim Schalten einer Transition könnte im Web-Editor angeboten werden. Dafür ist es aber nötig, die Liste der aktiven Transitionen und jeweils die Liste der verfügbaren Bindungen vom Renew Simulator abfragen zu können.

Neben der in dieser Arbeit verwendeten sequentiellen Simulationssemantik unterstützt der Renew Simulator auch eine Semantik, in welcher mehrere Transitionen innerhalb eines Zeitschritts nebenläufig geschaltet werden können. Der Web-Editor könnte Benutzern die verschiedenen Simulationsmodi zur Auswahl stellen. Bei Verwendung der nebenläufigen Semantik könnte das manuelle Schalten von nebenläufigen Transitionen per Mehrfingergeste auf einem Touchscreen erlaubt werden.

Aktuell speichert der Web-Editor nur den akkumulierten Zustand einer Simulation zum aktuellsten Zeitpunkt. Ähnlich wie die Historie beim Bearbeiten eines Dokuments könnte der Web-Editor aber auch das gesamte Simulationsprotokoll in Form aufzeichnen und den Benutzern die Möglichkeit geben, rückwärts durch die Simulation zu navigieren.

Wenn vergangene Zustände einer Simulation aufgezeichnet wurden, könnten diese auch wiederhergestellt werden. Dafür müsste der Renew Simulator ebenfalls eine entsprechende Schnittstelle bereitstellen, damit der interne Simulationszustand von außen gesetzt werden kann. Dann hätten Benutzer die Wahl, den Schaltvorgang einer einzelnen Transition rückgängig zu machen.

Zusammengefasst lässt sich feststellen, dass die aktuelle Anbindung des WebEditors an den Renew Simulator sehr eingeschränkt ist und entsprechend sehr viel Potenzial für zukünftige Verbesserungen besteht.

27.4. Grafische Benutzeroberfläche

Im Rahmen dieser Arbeit wurde eine vollwertige Benutzeroberfläche entwickelt, die sich sowohl auf Desktop als auch auf Mobilgeräten einsetzen lässt. Allerdings konnten im begrenzten Zeitrahmen auch viele in Renew Funktionen, die auch im Web-Editor sinnvoll wären, nicht umgesetzt werden.

Im Rahmen dieser Arbeit wurde grundsätzlich auf die verständliche Darstellung von Fehlermeldungen verzichtet. Bei der Weiterentwicklung des Web-Editors sollte hier nachgebessert werden. Typische Fehlerfälle sollten eigenständig behandelt und verständlich in der Benutzeroberfläche dargestellt werden. Beispiele für solche Fehlerfälle sind das Fehlschlagen einer Simulation oder das erfolglose Exportieren einer rnw-Datei.

Aktuell kann im Web-Editor immer nur ein einzelnes Element ausgewählt und bearbeitet werden. Es ist beispielsweise nicht möglich, die Schriftfarbe von vielen verschiedenen Textelementen auf einmal zu verändern oder mehrere Elemente auf einmal zu löschen. Für die effizientere Arbeit in einem Dokument wäre es sinnvoll, alle Änderungsoperationen auch auf Mengen von Elementen statt auf einzelnen Elementen ausführen zu können. Die meisten Operationen können vermutlich relativ einfach entsprechend erweitert werden. Besonders für Operationen, die sich auf die Hierarchie von Ebenen in einem Dokument beziehen, muss aber herausgearbeitet werden, wie diese sich in Bezug auf eine Menge aus Elementen verhalten sollen.

Derzeit bietet der Web-Editor keine Operationen zum Kopieren, Ausschneiden und Einfügen von Elementen. Für die effiziente Arbeit sind diese Operationen aber nötig, da sie es erlauben, viele Arbeitsschritte beim Erstellen vieler ähnlicher Elemente zu sparen. Im Editor-Server ist bereits eine Operation zum Kopieren ganzer Dokumente definiert. Diese Operation kann angepasst werden, um auch das Kopieren von einzelnen Elementen zu erlauben. Das genaue Verhalten der Kopie- und Einfüge-Operationen muss aber besonders in Bezug auf die Ebenenhierarchie eines Dokuments noch ausführlich spezifiziert werden.

In den grafischen Benutzeroberflächen vieler Anwendungen ist es üblich, dass sich per rechter Maustaste ein sogenanntes Kontextmenü aufrufen lässt, welches Operationen anbietet, die sich auf Elemente in der Nähe des Mauszeigers beziehen. Im Web-Editor dieser Arbeit wurde so ein Kontextmenü nicht umgesetzt. Dies wäre aber eine mögliche Erweiterung für die Zukunft.

Die Umsetzung der Dokumentendarstellung als SVG ist auf natürliche Weise kompatibel mit der von Webbrowsern angebotenen Suchfunktion. Somit lässt sich ein Dokument im Web-Editor bereits jetzt nach enthaltenen Textelementen durchsuchen. Es kann aber auch sinnvoll sein, die Menge aller Dokumente nach verschiedenen Kriterien zu durchsuchen. Hierfür wäre eine gesonderte Suchfunktion in Bezug auf alle gespeicherten Dokumente sinnvoll.

In Renew können Elemente in einem Dokument mittels verschiedener Layout-Algorithmen automatisch im Dokument positioniert werden. Beispielsweise können mittels einer Masse-Feder-Simulation die Knoten eines gezeichneten Netzes auf einen Mindestabstand gebracht werden. Alternativ lassen sich Elemente auch an einem vordefinierten Gitternetz ausrichten. Im Rahmen dieser Arbeit wurde zwar ein Gitternetz umgesetzt, welches sich grafisch in der Zeichenfläche einblenden lässt, allerdings hat es keinen automatischen Einfluss auf die Positionen der Elemente. Für eine einfache Positionierung von Elementen im Dokument wäre es sinnvoll, den Web-Editor um solche automatischen Positionierungen zu erweitern. Dies könnte entweder im Client umgesetzt werden, sodass die berechneten Positionen an den Server geschickt werden, oder im Server, sodass die automatisch bestimmte Position an zentraler Stelle erzwungen wird.

In Kapitel 15 „Grafische Darstellung“ wurde bereits demonstriert, wie sich die Web-Editor Dokumente auch mittels WebGL darstellen lassen. Dies könnte als Experiment weiter verfolgt werden, um die gezeichneten Petrinetze dreidimensional in einer Virtual Reality (VR) oder Augmented Reality (AR) in einem Head-Mounted Display (HMD) darzustellen und eine zugehörige Benutzeroberfläche zu entwickeln. Zusätzlich könnte weiter untersucht werden, ob sich besonders große Dokumente effizienter mittels WebGL als mittels SVG darstellen lassen.

27.5. Spezialisierung auf einzelne Formalismen

Im Web-Editor ist es möglich, beliebige Graphen aus Symbolen, gerichteten und ungerichteten Kanten mit beliebigen Textanschriften zu zeichnen. Nur solche Dokumente, die der von Renew erwarteten Referenznetzstruktur entsprechen, lassen sich letztendlich simulieren. Konkret können im Web-Editor beispielsweise zwei Platzknoten erzeugt und durch eine Kante direkt miteinander verbunden werden. Das Resultat ist kein gültiges Petrinetz.

In der Benutzeroberfläche von Renew sind die Zeichenwerkzeuge direkt an die in Renew verfügbaren Formalismen gekoppelt. Entsprechend bietet Renew auch eine viel größere Menge an Zeichenwerkzeugen an, da für jeden von Renew unterstützten Formalismus spezifische Werkzeuge bereitgestellt werden.

In dieser Arbeit wurde der Fokus auf das Erstellen und Simulieren von Referenznetzen gelegt. Das Datenmodell des Web-Editor erlaubt aber prinzipiell das Erstellen von Modellen in allen durch Renew unterstützten Formalismen. Nur sind eben die bereitgestellten Werkzeuge nicht auf einzelne Formalismen zugeschnitten und erzwingen nicht die Einhaltung der jeweiligen Syntax.

Eine zukünftige Erweiterung des Web-Editors wäre also die Bereitstellung von auf einzelne Formalismen spezialisierten Werkzeugen in der Benutzeroberfläche.

27.6. Metamodellierung

Im Rahmen dieser Arbeit wurde das Thema Metamodellierung nicht behandelt. Ein Metamodell könnte allerdings verwendet werden, um zu beschreiben, aus welchen Elementen ein Dokument zusammengesetzt werden darf und in welchen Beziehungen die Elemente zueinander stehen müssen. Darüber könnten die Anforderungen der einzelnen Formalismen so formuliert werden, dass sich das Verhalten der Werkzeuge des Web-Editors an den jeweils gewählten Formalismus anpassen.

Das in dieser Arbeit vorgeschlagene relationale Datenmodell eignet sich vermutlich gut als Grundlage für die Umsetzung so eines solchen Metamodells. Besonders die in Kapitel 7 „Parametrische Grafiksymbole“ parametrisierten Grafiksymbole und die in Kapitel 8 „Relationales Datenschema“ durchgeführte Normalisierung der Menge Renew Figure-Elementen bieten eine gute Grundlage für die Konstruktion eines Metamodells.

27.7. Offline Bearbeitung

Im Rahmen dieser Arbeit wurde die Verwendung des entwickelten Web-Editors ohne verfügbare Netzwerkverbindung im Vorfeld ausgeschlossen. Es wäre aber interessant zu untersuchen, ob die in dieser Arbeit vorgeschlagene Schnittstelle zwischen Client und Server sich auch dazu eignet, Offline-Bearbeitungsoperationen zu ermöglichen.

Im Web-Editor Server wird eine SQL-Datenbank eingesetzt, um den Inhalt der bearbeiteten Dokumente zentral zu speichern. Das relationale Datenmodell ließe sich aber auch mittels JS SQLite direkt im Client umsetzen, sodass alle Änderungsoperationen vollständig lokal durchgeführt werden können. Für das kollaborative Arbeiten müssen die Änderungen verschiedener Clients natürlich noch synchronisiert werden.

27.8. Neue Elemente in Dokumenten

Zu Beginn dieser Arbeit wurde zur Einschränkung des Arbeitsumfangs festgelegt, dass sich im Web-Editor nur solche Dokumente bearbeiten lassen sollen, die auch in Renew erstellt werden können.

Die SVG basierte Darstellung der Dokumente bietet nun aber die Möglichkeit für einige zukünftige Erweiterungen, die auch für Renew schon auf der Agenda stehen. Ein Beispiel hierfür ist die Darstellung von freier Formatierung der Textelemente in einem Dokument. Aktuell lassen sich zwar Größe, Farbe und Schriftart eines einzelnen Textelementes verändern, allerdings können innerhalb eines Textelements keine einzelnen Wörter oder Zeichenfolgen unterschiedlich gestaltet werden. Auch die Darstellung von Mathematischen Ausdrücken in LaTeX-Syntax ist eine sinnvolle zukünftige Erweiterung. Die SVG-Darstellung würde sich auch dazu eignen, Animationen innerhalb eines Dokumentes oder zur Darstellung von Schaltvorgängen während der Simulation einzusetzen. Aktuell leuchten schaltende Transitionen zwar auf, aber die Markenbewegung wird schlagartig dargestellt. Um die Markenbewegung kontinuierlich animiert darzustellen, müsste aber auch die Simulationssemantik die eindeutige Zuordnung von konsumierten zu produzierten Marken erlauben.

28. Erkenntnisse für Java Renew

Die Motivation für diese Arbeit war nicht nur die Entwicklung eines neuen Editors zum Bearbeiten und Simulieren von Referenznetzen, sondern auch das Erkunden von möglichen Verbesserungen, die im bestehenden Renew umgesetzt werden können. In diesem Kapitel werden die gesammelten Erkenntnisse zusammengetragen.

Es wird vorgeschlagen, das .rnw-Dateiformat zu überarbeiten, das Renew-interne Datenmodell zu normalisieren, die Berechnung von Textmetriken aus dem Datenmodell zu entkoppeln, die Kamerasteuerung von Renew zu überarbeiten und die Befehlszeilenschnittstelle von Renew robuster zu gestalten.

28.1. Definition einer Grammatik für .rnw-Dateiformat

Zu Beginn dieser Arbeit wurde in Kapitel 6 „Import von rnw-Dateien“ zunächst eine Grammatik zum Lesen von rnw-Dateien erarbeitet. Eine Hürde hierbei war, dass das Datenformat von rnw-Dateien in Renew nirgendwo einheitlich definiert ist. Es ergibt sich nur implizit aus den Implementationen aller Java-Klassen, die das `Storable`-Interface implementieren. Die Menge dieser Klassen variiert zwischen verschiedenen Renew Versionen und hängt davon ab, welche Plugins geladen sind. Aufgrund fehlender Trennzeichen können rnw-Dateien im Allgemeinen nicht eingelesen werden, ohne die Definition all dieser Klassen zu kennen.

Es wäre sinnvoll, das Datenformat für rnw-Dateien so zu überarbeiten, dass eine einheitliche Grammatik definiert wird, auf die sich externe Anwendungen verlassen können, um mit Renew zu interagieren. Außerdem wäre es sinnvoll, dass diese Grammatik so strukturiert wird, dass sie unabhängig von der Menge der in Renew definierten Java-Klassen ist. Durch die Verwendung von öffnenden und schließenden Klammern könnte die Baumstruktur eines serialisierten Dokuments ohne mehrdeutige Präzedenzen ausgedrückt werden.

28.2. Normalisierung des Datenmodells

Die Menge an definierten `Figure`-Klassen in Renew ist sehr groß. Für jedes neu definierte Element, welches gezeichnet werden kann, muss eine eigene Klasse definiert werden. In Kapitel 8 „Relationales Datenschema“ wurde gezeigt, wie die Menge an einprogrammierten Elementen von über 200 auf unter 10 reduziert werden kann, ohne die Menge an darstellbaren Elementen zu reduzieren.

Die Reduktion wurde vor allem durch die parametrische Definition von Symbolgrafiken und die parametrisierte Entkopplung von Konrektoren erreicht. Die Menge an Java-Klassen in Renew könnte stark verringert werden, wenn die grafische Darstellung von `Figure`-Klassen und verschiedenen `Connector`-Klassen jeweils in eine vereinheitlichte Datenstruktur zusammengefasst wird.

28.3. Entkopplung des Text-Layouts

In Renew ist die Positionierung von Textelementen im Dokument stark an Java AWT gekoppelt. AWT wird benötigt, um die Breite und Höhe von Text-Elemente zu berechnen. Die Maße von Textelementen werden benötigt, um die Position anderer Elemente relativ zum Text zu bestimmen. Das führt dazu, dass die Bearbeitung von Dokumenten in Renew immer von der Verfügbarkeit von AWT abhängt.

Für die Entkopplung von AWT wäre es sinnvoll, diese Abhängigkeit zu lösen. Eine Möglichkeit wäre es, die Maße von Textelementen explizit zu persistieren und auch beim Speichern von rnw-Dateien zu serialisieren. Dann wäre die automatische Berechnung der Maße ein optionaler Vorgang, der durchgeführt werden kann, falls AWT verfügbar ist. Benutzer können die Maße dann auch manuell eingeben und andere Anwendungen können rnw-Dokumente grafisch darstellen, ohne die Textmaße eigenständig nachberechnen zu müssen.

28.4. Überarbeitung des Kameramodells

In dieser Arbeit konnte erfolgreich ein Kameramodell für die grafische Navigation in einem Dokument entwickelt werden. Es eignet sich zum Zoomen, Rotieren und Verschieben der Ansicht, ist kompatibel mit Scrollbalken, Fingergesten und Maussteuerung.

Es ist zwar in JS programmiert und lässt sich somit nicht direkt in die Java AWT Oberfläche von Renew übertragen, aber die grundlegende Datenstruktur ist auch für Renew verwendbar. Basierend auf den Kamerainteraktionen, die für den Web-Editor umgesetzt wurden, lässt sich auch die Kameranavigation von Renew überarbeiten.

28.5. Verbesserungen an der Befehlszeilenschnittstelle

In Kapitel 20 „Anbindung an Java Renew“ wurde in Bezug auf die Simulatoranbindung schon erklärt, wie der Funktionsumfang des Web-Editors durch die Befehlszeilenschnittstelle von Renew beschränkt ist. Es sind aber auch einige Hürden in der Benutzbarkeit der Schnittstelle aufgefallen. Es wird also vorgeschlagen, die Schnittstelle in einigen Aspekten zu erweitern.

Zunächst sollte die Verarbeitung des Standardeingabestroms im `Console` Plugin so angepasst werden, dass Renew vollständig beendet wird, wenn der Eingabestrom seitens des Betriebssystems geschlossen wird. Dadurch ließe sich Renew robuster im Zusammenspiel mit anderen Anwendungen wie dem Web-Editor verwenden. Sollte diese Änderung in Konflikt mit bestehenden Anwendungen stehen, sollte zumindest eine Option für das vorgeschlagene Verhalten angeboten werden.

Als nächstes sollte das `Formalism`-Plugin von der Abhängigkeit zu AWT befreit werden. Damit könnte der SNS Compiler von Renew einfacher auf Systemen ohne grafische Benutzeroberfläche verwendet werden, ohne einen virtuellen Framebuffer emulieren zu müssen.

Aktuell erwartet die Befehlszeilenschnittstelle sowohl die Befehle als auch die zu kompilierenden rnw-Dateien und die zu simulierenden SNS-Dateien in Form von Dateien, die im Dateisystem gespeichert sind. Um die Simulation für ein SNS zu starten, muss der entsprechende Dateiname an die Befehlszeilenschnittstelle gesendet werden. Zum Durchführen einer Simulation muss der Web-Editor diese Dateien also erzeugen, Renew starten und anschließend die Dateien wieder löschen. Die Schnittstelle wäre einfacher zu verwenden, wenn alle nötigen Daten direkt über den Eingabedatenstrom an Renew übergeben werden könnten. Daher wäre es sinnvoll, die Schnittstelle entsprechend zu erweitern.

Für die einfachere Verarbeitung der Simulationsergebnisse sollte das Ausgabeformat von Grund auf maschinenlesbar gestaltet werden. Es könnte eine Option hinzugefügt werden, um die Simulationsereignisse im JSON-, XML- oder Symbolic Expression (S-Expr)-Format auszugeben.

Anstatt einzelne Simulationsereignisse auszugeben, die nachträglich zu einem Zustand akkumuliert werden müssen, wäre eine alternative Schnittstelle sinnvoll, über die sich der gesamte Zustand einer Simulation auf einen Schlag ausgeben lässt. Andersherum wäre es auch sinnvoll, während einer pausierten Simulation den gesamten Simulationszustand von außen überschreiben zu können. Dann hätte der WebEditor mehr Freiheit, mit dem Simulator zu interagieren. Das Speichern und Laden des Zustands ist über die GUI bereits möglich.

Auch den Endzustand einer Simulation aktiv zu signalisieren, wäre eine sinnvolle Erweiterung. Wenn sich keine weiteren Transitionen schalten lassen, kann die Simulation als abgeschlossen betrachtet werden. Der Web-Editor könnte auf dieses Signal reagieren und den Renew Prozess beenden.

Für die interaktive Simulation, die in Renew über die grafische Benutzeroberfläche bereits verfügbar ist, wäre es sinnvoll, auch über die Befehlszeile eine Liste der aktiven Transitionen und verfügbaren Belegungen abfragen zu können. Darauf aufbauend wäre es dann auch sinnvoll, das Schalten einer Teilmenge von Transitionen über die Befehlszeile auslösen zu können.

Teil VII.

Abschluss

29. Zusammenfassung

Im Rahmen dieser Arbeit konnte erfolgreich ein Editor für das kollaborative Bearbeiten und Simulieren von Referenznetzen entwickelt werden. In diesem Kapitel werden die in vorherigen Kapiteln vorgestellten Arbeitsschritte und Ergebnisse noch einmal zusammengefasst.

29.1. Einleitung

Das Ziel dieser Arbeit, einen kollaborativen Web-Editor für Petrinetze zu entwickeln, wurde in Kapitel 1 „Einleitung“ motiviert. Dafür wurde in Kapitel 2 „Grundlagen“ ein Überblick gegeben, darüber, was Referenznetze sind und in welche Form die Renew Entwicklungsumgebung das Bearbeiten und Simulieren ermöglicht.

Außerdem wurde in Kapitel 3 „Verwandte Arbeiten und Stand der Forschung“ untersucht, welche Versuche es schon gab, Renew um kollaborative Werkzeuge zu erweitern und die grafische Benutzeroberfläche von Renew auf Mobilgeräten und im Webbrowser zur Verfügung zu stellen. Zudem wurde ein Überblick über die existierende Vielfalt an Werkzeugen für den Entwurf von Petrinetzen gegeben. Es wurde festgestellt, dass keines von ihnen die Ausdrucksstärke der Renew Formalismen mit einer modernen Benutzeroberfläche und kollaborativen Werkzeugen vereint.

29.2. Konzeption

In Kapitel 4 „Anforderungen“ wurden Anforderungen für den in dieser Arbeit zu erzielenden Editor aufgestellt. Dabei wurde der Fokus so gewählt, dass am Ende dieser Arbeit eine einsatzfähige Anwendung entsteht. Der Editor sollte sowohl auf Desktop- als auch auf Mobilgeräten verwendbar sein. Der Referenznetzformalismus von Renew sollte für die Simulation von Netzen im Editor verfügbar sein. Das gemeinsame Arbeiten mehrerer Anwenderinnen an einem Netz soll möglich sein. Um die Umsetzbarkeit im begrenzten Zeitrahmen dieser Arbeit zu ermöglichen, wurde der Funktionsumfang in einigen Punkten auch im Vorfeld eingegrenzt. Zusätzlich wurde gefordert, dass der entwickelte Editor im Datenformat kompatibel mit Renew ist, um so gut es geht von dem in Renew existierenden Funktionsumfang Gebrauch machen zu können.

In Kapitel 5 „Strategie zur Umsetzung“ wurde eine Strategie gewählt, die verwendet werden soll, um auf die Erfüllung der Anforderungen hinzuarbeiten. Als grobe Softwarearchitektur des Editors wurde eine Client-Server-Architektur gewählt. Für die Umsetzung der grafischen Benutzeroberfläche wurde eine im Webbrowser ausgeführte JS-Anwendung vorgeschlagen. Für die Programmierung des Servers wurde Elixir als funktionale Programmiersprache gewählt. Das Datenmodell sollte mittels relationaler Datenbank umgesetzt werden.

29.3. Umsetzung des Editor-Servers

In Kapitel 6 „Import von rnw-Dateien“ wurde der Entwurf einer Grammatik zum Einlesen bestehender Renew-Dokumente behandelt. Zum einen konnten die dabei gewonnenen Erkenntnisse eingesetzt werden, um später das Datenmodell des Editors zu modellieren. Zum anderen konnten einige Verbesserungsvorschläge für das Renew Dateiformat erarbeitet werden. Die entwickelte Grammatik konnte als eigenes Elixir-Softwarepaket veröffentlicht werden.

In Kapitel 7 „Parametrische Grafiksymbole“ wurde ein parametrisches Vektorgrafikformat entwickelt. Dies war nötig, um alle von Renew definierten Grafiksymbole, die in den eingelesenen Dokumenten referenziert sein können, in ein einheitliches Format zu übersetzen, sodass sie auch außerhalb von Renew im Web-Editor verarbeitet werden können. Dieses parametrisierte Vektorformat konnte ebenfalls als eigenständiges Softwarepaket veröffentlicht werden.

In Kapitel 8 „Relationales Datenschema“ wurde ein relationales Datenmodell entwickelt. Für dieses war wichtig, dass es vollständig kompatibel mit dem von Renew verwendeten objektorientierten Modell ist, damit sich alle eingelesenen Renew Dokumente verlustfrei übersetzen lassen. Dabei konnten die in Renew definierte Hierarchie aus über 200 Java-Klassen in eine geringe Menge an Entitäten im relationalen Modell reduziert werden.

In Kapitel 9 „Automatisches Layout von Kanten“ wurde ein Modell zur automatischen Positionierung von Kantenendpunkten entwickelt. Dieses sollte im Verhalten mit dem von Renew übereinstimmen. Allerdings wurde eine gegenüber Renew vereinfachte und einheitlichere Umsetzung mittels Raycasting und SDF gewählt. Verschiedene Positionsdefinitionen konnten in ein einheitliches Datenformat überführt werden. Dieses Datenformat konnte ebenfalls als eigenes Softwarepaket veröffentlicht werden.

In Kapitel 10 „Operationen zur Bearbeitung von Dokumenten“ wurden die vom Editor unterstützten Bearbeitungsoperationen erarbeitet. Dabei wurde auf die Trennung von Lese- und Schreibzugriffen und die Verwendung des Kommando-Entwurfsmusters gesetzt. Es wurde erklärt, wie sich alle Operationen als Transaktionen im relationalen Modell formulieren lassen.

In Kapitel 11 „Versionierung von Dokumenten“ wurde eine Versionshistorie umgesetzt. Diese erlaubt, durch das Wiederherstellen gespeicherter Zwischenstände, das Rückgängigmachen von Änderungsoperationen im Editor. Es wurde sich für eine robuste und einfache Umsetzung entschieden, bei der die gesamte Kopie eines Dokuments als Zwischenstand gespeichert wird.

Auf Basis der in Kapitel 6 „Import von rnw-Dateien“ entwickelten Grammatik wurde in Kapitel 12 „Export von rnw-Dateien“ ein Export entwickelt. Dieser ermöglicht das Speichern von Dokumenten im Renew Dateiformat. So können die im Web-Editor erstellten Dokumente in Renew geladen werden. Dadurch wird auch die spätere Simulation der Netze durch Renew ermöglicht.

In Kapitel 13 „Web-Schnittstelle des Editor Servers“ wurde schließlich eine HTTP- und WebSocket-basierte Schnittstelle entwickelt, über welche die im Web-Editor bearbeiteten Dokumente abgerufen und verändert werden können. Auch hier wurde die Trennung zwischen Schreib- und Lesezugriff gesetzt. Der zentrale Aspekt der entwickelten Schnittstelle ist die einheitliche Übertragung von Änderungen als JSON-Patches vom zentralen Server an alle Clients.

29.4. Umsetzung des Editor-Clients

In Kapitel 14 „Kommunikation mit Editor-Server“ wurde die clientseitige Umsetzung der Schnittstelle behandelt. Der Client wurde als JS-Anwendung im Webbrowser umgesetzt. Da sich aus der Gestaltung der Schnittstelle ein sehr linearer Datenfluss ergibt, hat sich die Umsetzung als sehr einfach herausgestellt.

In Kapitel 15 „Grafische Darstellung“ wurde die grafische Darstellung der vom Server bereitgestellten Dokumenteninhalte im Client erarbeitet. Hierfür wurden zwei verschiedene Ansätze verglichen. In einem Experiment wurde versucht, Dokumente mittels WebGL zu rendern. Das hat für eine vereinfachte Darstellung gut funktioniert, wurde aber dann nicht weiter verfolgt. Stattdessen wurde SVG verwendet, um Dokumente in ihrem vollen Detailgrad grafisch darzustellen.

In Kapitel 16 „Reaktives Datenmodell“ wurde vorgestellt, wie im Client dynamisch auf vom Server bereitgestellte Änderungen eines Dokuments reagieren kann, um die als SVG erzeugte grafische Darstellung stets aktuell zu halten. Dafür wurde das Paradigma der reaktiven Programmierung eingesetzt. Dieses lässt sich mit Hilfe verschiedener Programmbibliotheken in JavaScript umsetzen. Die in dieser Arbeit verwendeten Bibliotheken wurden kurz genannt, aber nicht vertiefend erklärt.

In Kapitel 17 „Kamera-Navigation“ wurde ein Modell zur Steuerung der Kameraansicht in der grafischen Darstellung eines Dokuments entwickelt. Mittels der entwickelten Steuerung lässt sich ein Dokument heranzoomen, drehen und auf dem Bildschirm verschieben. Zunächst wurde untersucht, welche Schwächen die Kameranavigation in Renew derzeit aufweist. Bei der Entwicklung der Kamera für den Web-Editor wurde sehr viel Detailarbeit darauf verwendet, dass sich die Kamera auf unterschiedlichen Geräten über verschiedene Eingabemodi störungsfrei navigieren lässt.

In Kapitel 18 „Lokale und verteilte Bearbeitung“ wurde untersucht, inwiefern sich die gewählte Client-Server-Architektur auf die wahrnehmbare Latenz von Bearbeitungsoperationen auswirkt. Die Bearbeitungsoperationen wurden so entworfen, dass durch einen abgestimmten Wechsel zwischen lokalen Schreiboperationen und vom Server verarbeiteten Operationen ein angenehmer Arbeitsablauf ermöglicht wird.

In Kapitel 19 „Kollaboratives Arbeiten“ wurden einige Werkzeuge für die Benutzeroberfläche entwickelt, die das kollaborative Arbeiten unterstützen sollen. Eine Liste der aktiven Benutzerinnen, die zurzeit in einem Dokument arbeiten, wird dargestellt . Farbige Mauszeiger und farblich hervorgehobene Elemente erlauben das Nachvollziehen der von anderen Benutzern durchgeführten Änderungen.

29.5. Umsetzung des Simulators

In Kapitel 20 „Anbindung an Java Renew“ wurde erarbeitet, wie sich der entwickelte Web-Editor an den in Renew enthaltenen Simulator anschließen lässt. Hierfür wurde die von Renew zur Verfügung gestellte Textbefehlszeile als Schnittstelle verwendet.

In Kapitel 21 „Speicherung des Simulationszustandes“ wurde ein relationales Datenmodell entworfen, damit der Weg“=Editor eine eigene Kopie eines Simulationszustandes abspeichern und an Clients bereitstellen kann. Außerdem wurde ein Prozess entwickelt, um die von Renew bereitgestellten Simulationsereignisse in einen aktuellen Gesamtzustand zu akkumulieren. Der erreichbare Funktionsumfang war hierbei vor allem durch die von Renew bereitgestellte Schnittstelle beschränkt.

In Kapitel 22 „Web-Schnittstelle des Simulators“ wurde auch für die Bereitstellung des Simulationszustands eine HTTP-Schnittstelle entwickelt. Dabei wurde die gleiche Architektur wie in Kapitel 13 „Web-Schnittstelle des Editor Servers“ gewählt.

In Kapitel 23 „Darstellung des Simulationszustandes“ wurde die grafische Darstellung von Simulationen im Client entwickelt. Hierbei wurde auf der zuvor entwickelten grafischen Darstellung von Dokumenten aufgebaut.

29.6. Diskussion und Evaluation

In Kapitel 24 „Zusammenfassung der Architektur“ wurden alle zuvor entwickelten Einzelkomponenten in einen gemeinsamen Kontext gesetzt und als gesamte Architektur des Systems vorgestellt. Dabei wurde besonders die gewählte Client-Server-Architektur noch einmal hervorgehoben.

In Kapitel 25 „Diskussion der erfüllten Anforderungen“ wurde untersucht, inwiefern die zu Beginn aufgestellten Anforderungen von dem in dieser Arbeit entwickelten Web-Editor erfüllt werden. Es wurde festgestellt, dass alle Anforderungen erfolgreich umgesetzt werden konnten. Im Rahmen dieser Arbeit konnte eine funktionsfähige Web-Anwendung für das gemeinsame Bearbeiten und Simulieren von Referenznetzen entwickelt werden.

In Kapitel 26 „Diskussion der gewählten Strategie“ wurde rückblickend untersucht, wie sich die zu Beginn gewählte Strategie auf die erfolgreiche Umsetzung dieser Arbeit ausgewirkt hat. Das gewählte Vorgehen und die eingesetzten Werkzeuge und Programmiersprachen haben die Umsetzung dieser Arbeit stark unterstützt.

In Kapitel 27 „Zukünftige Erweiterungen und Verbesserungen“ wurden einige Vorschläge unterbreitet, wie sich der entwickelte Web-Editor zukünftig noch weiter erweitern und verbessern ließe. Im begrenzten Zeitrahmen dieser Arbeit mussten einige Funktionen von vornherein ausgeschlossen werden. Entsprechend ist die Menge an nun möglichen Erweiterungen sehr groß — besonders wenn man sich an dem bestehenden Funktionsumfang von Renew orientiert. Es kann aber vermutet werden, dass sich die Architektur

des Web-Editors gut dafür eignet, um die bisher nicht umgesetzte Funktionalität erweitert zu werden.

In Kapitel 28 „Erkenntnisse für Java Renew“ wurden einige Vorschläge zusammengefasst, wie sich Renew zukünftig verbessern lässt. Im Laufe dieser Arbeit sind einige Hürden aufgefallen, die die Interaktion zwischen dem Web-Editor und Renew erschwert haben. Wenn einige Schnittstellen von Renew erweitert und präzisiert werden, lässt es sich noch besser als Simulator für den Web-Editor verwenden. Außerdem lassen sich auch einige für den Web-Editor entwickelte Techniken auf Renew anwenden. Beispiele hierfür sind das Kameramodell und die parametrischen Vektorgrafiken.

Abschließend konnte festgestellt werden, dass das Ziel dieser Arbeit erfolgreich umgesetzt werden konnte. Der entwickelte Web-Editor lässt sich verwenden, um Renew-Dokumente zu bearbeiten und zu simulieren. Es wurden einige technische Erkenntnisse gesammelt, die in den Entwicklungsprozess von Renew zurückfließen können.

30. Ausblick

In Kapitel 27 „Zukünftige Erweiterungen und Verbesserungen“ und Kapitel 28 „Erkenntnisse für Java Renew“ wurden schon einige Vorschläge für zukünftige technische Erweiterungen sowohl des Web-Editors als auch von Renew aufgeführt.

In diesem Kapitel wird nochmal zusammenfassend ein kurzer Ausblick gegeben, wie sich der Web-Editor in der Praxis einsetzen lässt und welches Potenzial es für dessen Weiterentwicklung gibt.

30.1. Anwendungsfälle

Für die Zukunft ist vor allem interessant, wie sich der Web-Editor nun in der Praxis einsetzen lässt. Der entwickelte Editor wurde im Laufe dieser Arbeit schon einzelnen erfahrenen Renew Benutzer:innen zur Verfügung gestellt. Allerdings wurden noch keine strukturierten Nutzerstudien durchgeführt, um zu untersuchen, wie sich der Editor für einzelne Anwendungsfälle verwenden lässt. Renew wird unter anderem in der Lehre eingesetzt, um die Verwendung von Petrinetzen zum Entwurf von verteilten Systemen zu unterrichten. Besonders für diesen Anwendungsfall verspricht der Web-Editor einige Vorteile gegenüber Renew. Die Benutzbarkeit im Webbrowser verringert die Hürde für Einsteiger, da keine Installation nötig ist. Das kollaborative Arbeiten erlaubt es Studentinnen, gemeinsam an einer Aufgabe zu arbeiten. Die mit Mobilgeräten kompatible Benutzeroberfläche erlaubt die Verwendung auch auf einem Tablet oder Smartphone während einer Lehrveranstaltung. Ob sich der Web-Editor tatsächlich in diesem Kontext wie erhofft einsetzen lässt und welche Hürden dafür noch zu überwinden sind, wird sich zeigen.

30.2. Weiterentwicklung

Auch wenn die Kompatibilität zu Renew ein zentraler Aspekt dieser Arbeit war, wurde der Web-Editor absichtlich als eigenständiges Projekt unabhängig von Renew entwickelt. Für die Zukunft des WebEditors ist zu klären, inwieweit die Arbeit an ihm neben dem Renew Projekt aktiv fortgesetzt werden kann. Selbst wenn der WebEditor nicht weiterentwickelt wird, hat diese Arbeit einige für Renew interessante Erkenntnisse geliefert und dient hoffentlich als Inspiration, auch Renew selbst voranzutreiben. Allerdings ist natürlich auch zu hoffen, dass der WebEditor selbst eine Gemeinschaft von Entwicklerinnen und Entwicklern findet, die seine Entwicklung weiter vorantreibt.

Um interessierte Menschen für die Weiterentwicklung des Editors zu begeistern und den Einstieg in den Entwicklungsprozess zu erleichtern, befindet sich im Anhang dieser Arbeit eine Zusammenstellung technischer Informationen.

A. Anhang

Einige organisatorische und technische Details zur Entwicklung des Web-Editors wurden im Hauptteil dieser Arbeit ausgelassen, um den roten Faden nicht zu unterbrechen. In diesem Kapitel werden diese Details nun noch nachgereicht.

A.1. Projektname und Logo

Ursprünglich wurde der in dieser Arbeit entwickelte Editor als *Prototyp eines* Renew *Webeditors* bezeichnet. Nachdem sich aber gegen Ende des Arbeitsprozesses herauskristallisiert hatte, dass es gelang, ein eigenständiges, abgerundetes Produkt zu entwickeln, wurde ein neuer, von Renew unabhängiger Name vergeben.

Der Web-Editor wurde *PetriStation* getauft. Der Name sollte einzigartig sein, um per Suchmaschine im Internet gut auffindbar zu sein. *PetriPad*, angelehnt an den kollaborativen Texteditor Etherpad, war als Name eines existierenden Renew Plugins schon vergeben. PetriStation ist ein Wortspiel als Mischung aus Petrinetz und dem britisch-englischen Begriff *petrol station* für Tankstelle.*PetriStation* ist also eine Anlaufstelle, die Energie für die Simulation von Petrinetzen liefert.

Passend zum Namen wurde auch ein einfaches Logo entworfen. Dieses ist inspiriert durch das Renew Logo, beinhaltet aber einen angedeuteten Buchstaben *S* aus dem Wort *Station*.

Die Domäne wurde registriert, um eine Instanz des Editors im Internet bereitzustellen. Diese ist aber nicht öffentlich zugänglich, da, wie in Kapitel 27 „Zukünftige Erweiterungen und Verbesserungen“ erklärt, die nötige Benutzerverwaltung noch fehlt.

Alternativ zur PetriStation wurde auch ein etwas verspielterer Name *PetriGator* mit einem zugehörigen Logo in Betracht gezogen.

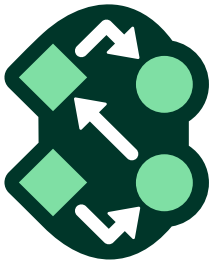


Abbildung A.1.: Durch Renew inspiriertes Logo für den *PetriStation* getauften Web-Editor

Abbildung A.2.: Alternatives Logo für die Veröffentlichung des Web-Editor unter dem alternativen Namen *PetriGator*

A.2. Videodokumentation

Der gesamte Entwicklungsprozess am Web-Editor wurde ergänzend zum schriftlichen Teil dieser Arbeit auch in Form von über 50 Kurzvideos dokumentiert. Jedes Video hat eine Dauer von wenigen Minuten und führt einzelne Funktionen des Web-Editors kleinschrittig vor.

Ursprünglich wurden die Videos nur im Confluence-Wiki vom Renew bereitgestellt und waren an die Gemeinschaft von Renew-Entwicklerinnen und Entwicklern adressiert.

Die Videos haben als eine besondere Art der *Definition of Done* fungiert. Eine Funktion konnte in dem Moment als ausreichend ausführlich implementiert angesehen werden, wenn ein sinnvolles Video zur Demonstration der Funktion aufgezeichnet werden konnte. Das Aufzeichnen der Videos hat verlangt, dass der Web-Editor vielfach wiederholt in seinem vollen Funktionsumfang verwendet wird. Hin und wieder sind bei der Videoaufzeichnung unerwartete Situationen aufgetreten. Der Editor musste dann entsprechend angepasst und in der Benutzbarkeit verbessert werden.

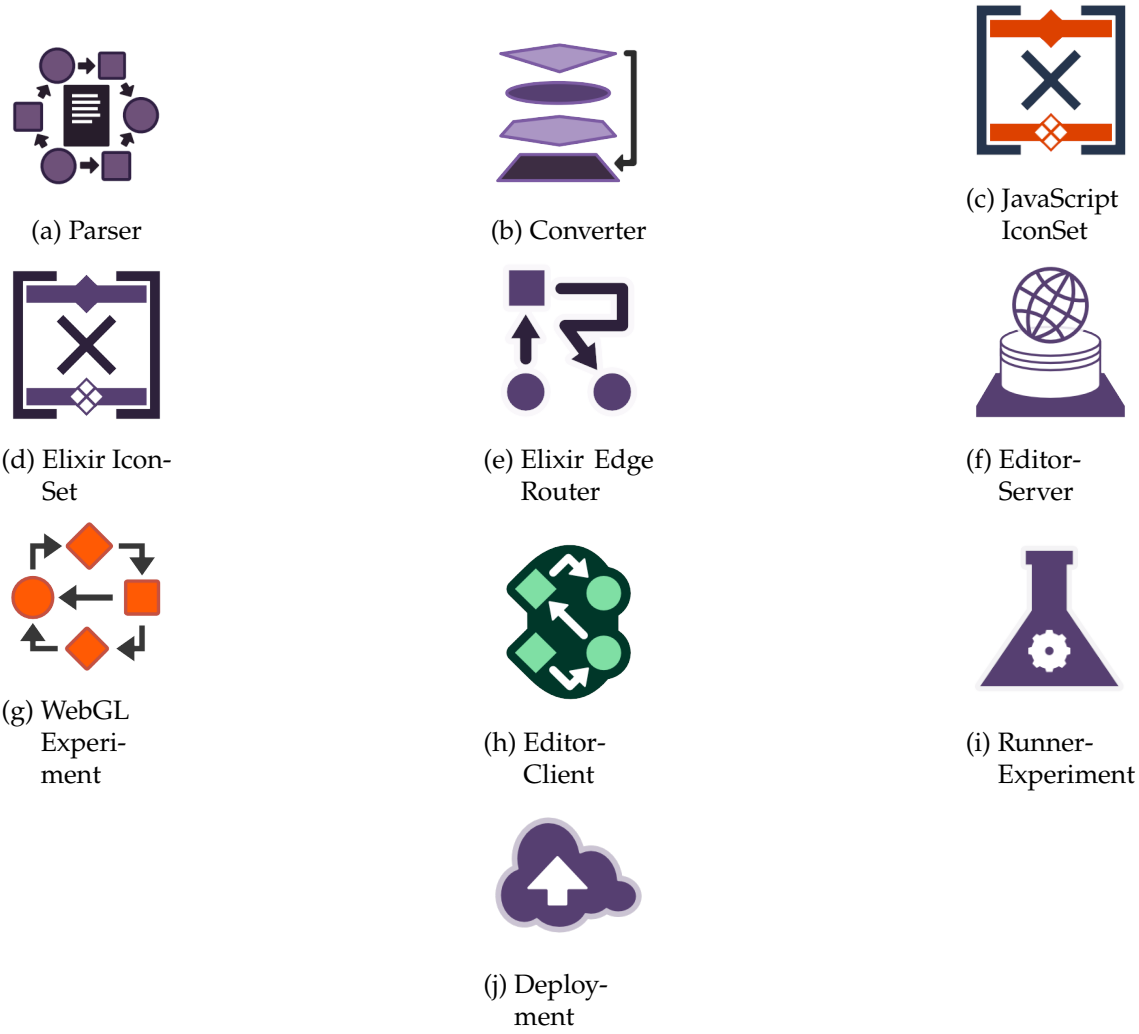
Alle Videos sind nun auch auf YouTube öffentlich verfügbar. In ihnen sind auch viele Details des Editors dargestellt und erklärt, auf die in Textform nicht eingegangen werden konnte [Kor25a].

A.3. Quelltext Repositorien

Neben der Einteilung in Server- und Client-Komponente lässt sich das gesamte Projekt in mehrere kleinere Module zerteilen. In Kapitel 6 „Import von rnw-Dateien“ und Kapitel 7 „Parametrische Grafiksymbole“ wurde erwähnt, dass die Grammatik zum Einlesen von Renew Dokumenten und das Datenformat für parametrische Vektorgrafiken als eigenständige Pakete im Elixir Hex Register veröffentlicht wurden.

Insgesamt wurde das gesamte Projekt in 10 Git-Repositorien aufgeteilt. Einige enthalten vollwertige Module und einige nur alleinstehende Experimente, die im Laufe der Arbeit durchgeführt wurden.

Alle Repositorien sind im GitLab des Informatikums der Uni Hamburg gespeichert. Für jedes Repositorium wurde ein zugehöriges Grafiksymbol entworfen, um eine optisch ansprechende Listendarstellung zu ermöglichen.



(a) Parser

(b) Converter

(c) JavaScript IconSet

(d) Elixir Icon-Set

(e) Elixir Edge Router

(f) Editor-Server

(g) WebGL Experi-ment

(h) Editor-Client

(i) Runner-Experiment

(j) Deploy-ment

Abbildung A.3.: Logos für die einzelnen Teilprojekte des Web-Editors

A.3.1. Renew Parser

Dieses Repositorium enthält die Module von Lesen und Schreiben von Renew rnw-Dokumenten. Es enthält auch eine Sammlung Unittests, die die Kompatibilität mit über 1000 existierenden Dateien überprüfen. Es ist auch als eigenständiges Hex-Paket verfügbar. Abb. A.3(a) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/parser`

Hex Veröffentlichung:
`https://hex.pm/packages/renewex`

A.3.2. Renew-Converter

Dieses Repositorium enthält das Modul zur Umwandlung zwischen von eingelesenen Renew Dateien in das relationale Datenmodell des Web-Editors. Es enthält auch einige zugehörige Unittests und steht auch als Hex-Paket zur Verfügung. Abb. A.3(b) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/converter`

Hex Veröffentlichung:
`https://hex.pm/packages/renewex_converter`

A.3.3. Parametrische Vektorgrafiken in JavaScript

Dieses Repositorium enthält die JS-Umsetzung des parametrisierten Vektorgrafikformats, welches für den Web-Editor entwickelt wurde. Es enthält auch eine Menge in diesem Format umgesetzte Grafiksymbole. Es wurde nicht als NPM-Paket veröffentlicht. Dies wäre aber für die Zukunft eine Möglichkeit. Abb. A.3(c) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/icon-set-js`

A.3.4. Parametrische Vektorgrafiken in Elixir

Dieses Repositorium enthält die Elixir-Umsetzung des parametrisierten Vektorgrafikformats, welches für den Web-Editor entwickelt wurde. Es enthält auch Unittests, aber keine vordefinierte Menge an Grafiksymbolen. Es wurde als Hex-Paket veröffentlicht. Abb. A.3(d) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/icon-set`

Hex Veröffentlichung:
`https://hex.pm/packages/renewex_iconset`

A.3.5. Positionierung von Kantenendpunkten

Dieses Repositorium enthält die Elixir-Module zur Berechnung von Kantenpositionen im Web-Editor. Es enthält zugehörige Unittests und wurde als Hex-Paket veröffentlicht. Abb. A.3(e) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/edge-router`

Hex Veröffentlichung:
`https://hex.pm/packages/renewex_routing`

A.3.6. Editor-Server

Dieses Repositorium enthält die gesamte Umsetzung des Editor-Servers und der Simulatoranbindung in Elixir. Es hängt von den anderen auf Hex veröffentlichten Elixir-Paketen ab. Für dieses GitLab-Projekt ist auch die Pipeline zum Bauen des zugehörigen Docker-Containers konfiguriert. Abb. A.3(f) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/editor-server`

A.3.7. Experimenteller WebGL Client

Dieses Repositorium enthält die JavaScript-Umsetzung der Dokumentendarstellung mittels WebGL und Three. Abb. A.3(g) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/webgl-client-experiment`

A.3.8. Editor-Client

Dieses Repositorium enthält die JavaScript-Umsetzung des Editor-Clients, der im Hauptteil dieser Arbeit diskutiert wurde. Dazu gehören unter anderem die Module zur Kamerasteuerung und zur Darstellung der Dokumente und Simulationen als SVG. Abb. A.3(h) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/editor-client`

A.3.9. Experiment zur Steuerung von Renew mittels Elixir Port

Dieses Repositorium enthält einige experimentelle Elixir-Skripte, die für die Ausarbeitung der Simulatoranbindung verwendet wurden. Sie befassen sich hauptsächlich mit der

korrekten Verwendung des Elixir `Port` Moduls im Zusammenspiel mit Renew. Abb. A.3(i) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/runner-experiment`

A.3.10. Scripts für Bereitstellung

Dieses Repositorium enthält eine Sammlung von Ansible-Playbook-Skripten für die Bereitstellung des Web-Editors in verschiedenen Konfigurationen. Abb. A.3(j) zeigt das zugehörige Grafiksymbol.

GitLab Repositorium:
`https://git.informatik.uni-hamburg.de/tgi/petri-station/deployment`

A.4. Bereitstellung am Informatikum

Für die interne Bereitstellung des Editors im Netzwerk des Informatikums der Uni Hamburg wurden entsprechende Container-Definitionen und ein Ansible-Skript angelegt. Am Informatik-Rechenzentrum der Uni Hamburg wurde eine virtuelle Maschine beantragt, die sich über das Ansible-Skript mit einer aktuellen Version des in dieser Arbeit entwickelten Web-Editors bespielen lässt.

Im GitLab wurden Pipelines konfiguriert, die jeweils aus dem Inhalt des Editor-Server-Repositoriums und des Editor-Client-Repositoriums ein Containerabbild bauen.

Das Ansible-Script lädt diese Containerabbilder herunter und startet mittels Docker-Compose eine Webserver-Konfiguration. Dadurch wird der Editor dann unter `https://petristation.informatik.uni-hamburg.de` bereitgestellt.

Zum aktuellen Zeitpunkt wurde noch kein signiertes Secure Sockets Layer (SSL)-Zertifikat für die Hypertext Transfer Protocol Secure (HTTPS)-Verbindung ausgestellt.

Die Zugangsdaten und weitere Details zur Durchführung des Bereitstellungsprozesses wurden im Renew Confluence dokumentiert.

Verzeichnisse

Tabellenverzeichnis

Quelltextverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AIP Agent-Interaction-Protocol

API application programming interface

AR Augmented Reality

AWT Abstract Window Toolkit

BEAM Bogdan's Erlang Abstract Machine

BPMN Process Model and Notation

CMMN Case Management Model and Notation

CPN Colored Petri Nets

CQRS Command-Query-Responsibility-Segregation

CRDT Conflict-free replicated data type

CSS Cascading Style Sheets

DMN Decision Model and Notation

DOM Document Object Model

DSL Domainspecific Language (domänenspezifische Sprache)

ECMAScript JavaScript

ECNO Event Coordination Notation

EMF Eclipse Modelling Framework

ER Entity-Relationship

GUI Graphical User Interface (grafische Benutzeroberfläche)

HMD Head-Mounted Display

HTML Hyptertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated Development Environment

JPMS Java Platform Module System

JS JavaScript

JSON JavaScript Objekt Notation

JVM Java Virtual Machine

MuPSi Multitouch Petrinet Simulator

NPM Node Package Manager

ORM Object Relational Mapper

OT Operational Transforms

P2P peer-to-peer

PNML Petri Net Markup Language

POSIX Portable Operating System Interface

REST Representational State Transfer

S-Expr Symbolic Expression

SDF Signed Distance Function

SNS Shadow Net System

SQL Structured Query Language (Strukturierte Abfrage-Sprache)

SSE Server Sent Events

SSL Secure Sockets Layer

SVG Scalable Vector Graphics

TCP Transmission Control Protocol

URL Uniform Resource Locator

UUID Universally Unique Identifier

VR Virtual Reality

WebRTC Web Real Time Connection

WWW World Wide Web

XML Extensible Markup Language

XVFB X Window Virtual Framebuffer

Literaturverzeichnis

[Ans18] Anschuetz, Henryk: *GitHub - Uzuul23/HPetriSim: HPetriSim has a graphical editor which provides basic editing and simulation of Petri Nets. — github.com*. https://github.com/Uzuul23/HPetriSim and https://github.com/Uzuul23/HOldPetriSim, 2018. – [Accessed 15-02-2025]

[Arm10] Armstrong, Joe: erlang. In: *Communications of the ACM* 53 (2010), Nr. 9, S. 68–75

[Atw06] Atwood, Jeff: *Object-Relational Mapping is the Vietnam of Computer Science — blog.codinghorror.com*. https://blog.codinghorror.com/object-relational-mapping-is-the-vietnam-of-computer-science/, 2006. – [Accessed 11-02-2025]

[Bac09] Backström, Lukas: *Erlang Zip Module Documentation*. https://www.erlang.org/doc/apps/stdlib/zip.html, 2009. – [Accessed 26-02-2025]

[Bar24] Barbosa, Douglas: *GitHub - douglasmbarbosa/tcc-petri-net-simulator: Repositório destinado ao desenvolvimento de uma aplicação web, em que será possível criar e simular Redes de Petri (RdP). A aplicação será utilizada como trabalho final de curso. — github.com*. https://github.com/douglasmbarbosa/tcc-petri-net-simulator and https://petrinetsimulator.com/, 2024. – [Accessed 15-02-2025]

[BCD+13] Betz, Tobias ; Cabac, Lawrence ; Duvigneau, Michael ; Wagner, Thomas ; Wester-Ebbinghaus, Matthias: Integrating Web Services in Petri Net-based Agent Applications. In: Moldt, Daniel (Hrsg.) ; Rölke, Heiko (Hrsg.): *Petri Nets and Software Engineering. International Workshop PNSE'13, Milano, Italia, June 2013. Proceedings* Bd. 989, CEUR-WS.org, Juni 2013 (CEUR Workshop Proceedings). – ISSN 1613–989, 97–116

[BCD+14] Betz, Tobias ; Cabac, Lawrence ; Duvigneau, Michael ; Wagner, Thomas ; Wester-Ebbinghaus, Matthias: Software Engineering with Petri Nets: a Web Service and Agent Perspective. In: *Transactions on Petri Nets and Other Models of Concurrency IX (ToPNoC)* (2014), Dezember, 41–61. http://dx.doi.org/10.1007/978-3-662-45730-6_3

[BCWE11] Betz, Tobias ; Cabac, Lawrence ; Wester-Ebbinghaus, Matthias: Gateway Architecture for Web-Based Agent Services. Version: 2011. http://dx.doi.org/10.1007/978-3-642-24603-6_17. In: Klügl, Franziska (Hrsg.) ; Ossowski, Sascha (Hrsg.): *Multiagent System Technologies* Bd. 6973. Springer-Verlag, 2011. – ISBN 978–3–642–24602–9, 165-172

[Bet11] Betz, Tobias: *Entwicklung eines Rahmenwerks für webbasierte Agentendienste – Modellierung auf Basis von Petrinetzen*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, September 2011

[BH12] Burkhart, Julian ; Haustermann, Michael: PetriPad – A Collaborative Petri Net Editor. In: [CDM12], 181–195

[BK22] Bergenthum, Robin ; Kovář, Jakub: *I Love Petri Nets — fernuni-hagen.de*. https://www.fernuni-hagen.de/ilovepetrinets/, 2022. – [Accessed 15-02-2025]

[BL89] Berners-Lee, Tim: Information Management: A Proposal. (1989). – Verfügbar unter: https://www.w3.org/History/1989/proposal.html

[BLP+07] Bonet, Pere ; Lladó, Catalina M. ; Puijaner, Ramon ; Knottenbelt, William J. u. a.: PIPE v2. 5: A Petri net tool for performance modelling. In: *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, 2007, S. –

[BM20] Bengtsson, Peter ; Mozilla Developer Network (MDN): *preserveAspectRatio - SVG: Scalable Vector Graphics | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/preserveAspectRatio, 2020. – [Accessed 27-02-2025]

[BN13] Bryan, Paul C. ; Nottingham, Mark: *JavaScript Object Notation (JSON) Patch*. RFC 6902. http://dx.doi.org/10.17487/RFC6902. Version: April 2013 (Request for Comments)

[BPSMM98] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve: *Extensible Markup Language (XML) 1.0 (Second Edition)*. https://www.w3.org/TR/1998/REC-xml-19980210. Version: 1998. – W3C Recommendation, Abgerufen am 11. Februar 2025

[Bra05] Bramer, Max A.: *Logic programming with Prolog*. Bd. 9. Springer, 2005

[Bra17] Bray, Tim: *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. http://dx.doi.org/10.17487/RFC8259. Version: Dezember 2017 (Request for Comments)

[Bud04] Budinsky, Frank: *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004

[Cab10] Cabac, Lawrence: *Modeling Petri Net-Based Multi-Agent Applications*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Department Informatik, Dissertation, April 2010. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:gbv:18-46661. – URN urn:nbn:de:gbv:18–46661

[Car21] Carniato, Ryan: *SolidJS — solidjs.com*. https://www.solidjs.com/, 2021. – [Accessed 17-02-2025]

[CC13] Czaplicki, Evan ; Chong, Stephen: Asynchronous functional reactive programming for GUIs. In: *ACM SIGPLAN Notices* 48 (2013), Nr. 6, S. 411–422

[CDM12] Cabac, Lawrence (Hrsg.) ; Duvigneau, Michael (Hrsg.) ; Moldt, Daniel (Hrsg.): *Petri Nets and Software Engineering. International Workshop PNSE'12, Hamburg, Germany, June 2012. Proceedings*. Bd. *851*. CEUR-WS.org, Juni 2012 (CEUR Workshop Proceedings). – ISSN 1613–0074

[Che76] Chen, Peter Pin-Shan: The entity-relationship model—toward a unified view of data. In: *ACM transactions on database systems (TODS)* 1 (1976), Nr. 1, S. 9–36

[CK10] Charalambous, Alex ; Knottenbelt, William: Extension of pipe2 to support coloured generalised stochastic petri nets. In: *Final Year Project, Imperial College London* (2010)

[CMH+22] Clasen, Laif-Oke ; Moldt, Daniel ; Hansson, Marcel ; Willrodt, Sven ; Voß, Lukas: Enhancement of Renew to Version 4.0 using JPMS. In: Köhler-Bußmeier, Michael (Hrsg.) ; Moldt, Daniel (Hrsg.) ; Rölke, Heiko (Hrsg.): *Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022* Bd. 3170, CEUR-WS.org, 2022 (CEUR Workshop Proceedings), 165–176

[CN19] Chęć, Dariusz ; Nowak, Ziemowit: The performance analysis of web applications based on virtual DOM and reactive user interfaces. In: *Engineering Software Systems: Research and Praxis* Springer, 2019, S. 119–134

[Com20] Community, Mozilla D.: *<use> - SVG: Scalable Vector Graphics | MDN — developer.mozilla.org*. https://developer.mozilla.org/en-US/docs/Web/SVG/Element/use, 2020. – [Accessed 26-02-2025]

[con25a] contributors, Svelte: *Svelte Documentation: Overview*. https://svelte.dev/docs/svelte/overview. Version: 2025. – Abgerufen am 11. Februar 2025

[con25b] contributors, WHATWG: *The DOM (Document Object Model) specification*. https://dom.spec.whatwg.org/. Version: 2025. – Abgerufen am 11. Februar 2025

[D+13] Dirksen, Jos u. a.: *Learning Three.js: the JavaScript 3D library for WebGL*. Bd. 3. Packt Publishing Livery Place, UK, 2013

[DGL+11] Dengler, Patrick ; Grasso, Anthony ; Lilley, Chris ; McCormack, C ; Schepers, D ; Watt, J: *SVG Document Object Model (DOM) SVG 1.1 (Second Edition) — w3.org*. https://www.w3.org/TR/SVG11/svgdom.html, 2011. – [Accessed 26-02-2025]

[Dib12a] Dibbern, Dominic: *Portierung des Petrinetzsimulators von Renew auf das Betriebssystem Android*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, März 2012

[Dib12b] Dibbern, Dominic: Porting the Renew Petri Net Simulator to the Operating System Android. In: [CDM12], 267–268

[DKS09] Dingle, Nicholas J. ; Knottenbelt, William J. ; Suto, Tamas: PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. In: *ACM SIGMETRICS Performance Evaluation Review* 36 (2009), Nr. 4, S. 34–39. – https://pipe2.sourceforge.net/

[Doc25] Docs, MDN W.: *WebSocket vs HTTP*. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers. Version: 2025. – Abgerufen am 11. Februar 2025

[ECM15] ECMA International: *ECMAScript 2015 Language Specification (6th Edition)*. https://www.ecma-international.org/publications/standards/Ecma-262.htm. Version: 2015. – ECMA-262, Abgerufen am 11. Februar 2025

[Ehl23] Ehlers, Kjell: *Untersuchung des JPMS für Java Programmsysteme zur Dekomposition mittels Modulschnittstellen für einzelne Plugins der Open-Source Software Renew*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, September 2023

[Elm02] Elmasri: *Grundlagen Datenbank Sys*. Harlow, England : Pearson Studium, 2002 (Pearson Studium - IT)

[Eng14] Engine, Godot: *Godot Engine - Free and open source 2D and 3D game engine — godotengine.org*. https://godotengine.org/, 2014. – [Accessed 17-02-2025]

[Eur14] Europe, Serif: *Software für Grafikdesign und Illustration | Affinity Designer — affinity.serif.com*. https://affinity.serif.com/de/designer/, 2014. – [Accessed 17-02-2025]

[FBC08] Forcier, Jeff ; Bissex, Paul ; Chun, Wesley J.: *Python web development with Django*. Addison-Wesley Professional, 2008

[Fie99] Fielding, Roy T.: *Hypertext Transfer Protocol – HTTP/1.1*. `https://tools.ietf.org/html/rfc2616`. Version: 1999. – RFC 2616, Abgerufen am 11. Februar 2025

[Fie00] Fielding, Roy T.: *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000

[Fig16] Figma, Inc.: *Figma: The Collaborative Interface Design Tool — figma.com*. `https://www.figma.com/`, 2016. – [Accessed 17-02-2025]

[FM08] Flanagan, David ; Matsumoto, Yukihiro: *The Ruby programming language*. Ö'Reilly Media, Inc.", 2008

[Fou94] Foundation, Blender: *blender.org - Home of the Blender project - Free and Open 3D Creation Software — blender.org*. `https://www.blender.org/`, 1994. – [Accessed 17-02-2025]

[Fow11] Fowler, Martin: » CQRS,«14 Juli 2011. In: *URL https://martinfowler. com/bliki/CQRS. html* (2011)

[Gac15] Gackenheimer, Cory: *Introduction to React*. Apress, 2015

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. Boston, MA : Addison-Wesley, 1994. – Das Buch beschreibt das Beobachter-Muster als eines der klassischen Entwurfsmuster.

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995

[Gla89] Glassner, Andrew S. (Hrsg.): *An introduction to ray tracing*. Oxford, England : Morgan Kaufmann, 1989 (The Morgan Kaufmann Series in Computer Graphics)

[GPB+22] Gaffney, Kevin P. ; Prammer, Martin ; Brasfield, Larry ; Hipp, D R. ; Kennedy, Dan ; Patel, Jignesh M.: Sqlite: past, present, and future. In: *Proceedings of the VLDB Endowment* 15 (2022), Nr. 12

[Heb13] Hebert, Fred: *Learn you some Erlang for great good!: a beginner's guide*. No Starch Press, 2013

[HK09] Hanus, Michael ; Kluß, Christof: Declarative Programming of User Interfaces. In: Gill, Andy (Hrsg.) ; Swift, Terrance (Hrsg.): *Practical Aspects of Declarative Languages*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978–3–540–92995–6, S. 16–30

[HKPT10] Hillah, Lom-Messan ; Kordon, Fabrice ; Petrucci, Laure ; Treves, Nicolas: PNML Framework: an extendable reference implementation of the Petri Net Markup Language. In: *Applications and Theory of Petri Nets: 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings 31* Springer, 2010, S. 318–327

[HT09] Hansson, David H. ; Team, Rails C.: Ruby on rails. In: *Development* 4 (2009), S. 0

[IHB12] Irgang, Thomas ; Harrer, Andreas ; Bergenthum, Robin: MuPSi-a multi-touch Petri net simulator for transition steps. In: *PNSE* Citeseer, 2012, S. 171–181

[Int11] Internet Engineering Task Force (IETF): *The WebSocket Protocol*. `https://tools.ietf.org/html/rfc6455`. Version: 2011. – RFC 6455, Abgerufen am 11. Februar 2025

[Jag12] Jagusch, Adrian: *GitHub - stromhalm/apo: An Application for Online Petri Net Design and Analysis — github.com*. `https://github.com/stromhalm/apo` and `https://apo.adrian-jagusch.de`, 2012. – [Accessed 15-02-2025]

[JI17] Jemerov, Dmitry ; Isakova, Svetlana: *Kotlin in action*. Simon and Schuster, 2017

[Jur24] Juric, Saša: *Elixir in action*. Simon and Schuster, 2024

[Kar16a] Karvonen, Vesa: *Introduction to Calmm.js*. `https://github.com/calmm-js/documentation/blob/master/introduction-to-calmm.md`. Version: 2016. – Abgerufen am 11. Februar 2025

[Kar16b] Karvonen, Vesa: *Partial Lenses — calmm-js.github.io*. `https://calmm-js.github.io/partial.lenses`, 2016. – [Accessed 17-02-2025]

[Khr18] Khronos Group: *WebGL 2.0 Specification*. `https://www.khronos.org/registry/webgl/specs/latest/2.0/`. Version: 2018. – Abgerufen am 11. Februar 2025

[Kil19] Kilian, Tim: *Entwicklung einer modularen JavaScript-Bibliothek zur Modellerstellung*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, 2019

[Kim17] Kim, Igor: *PNets — petri.hp102.ru*. `https://petri.hp102.ru/pnet.html` and `https://github.com/igorakim/pnet-simulator`, 2017. – [Accessed 15-02-2025]

[Kin11] Kindler, Ekkart: Integrating behaviour in software models: an event coordination notation–concepts and prototype. In: *Proceedings of the Third Workshop on Behavioural Modelling*, 2011, S. 41–48

[Kin12a] Kindler, Ekkart: An ECNO semantics for Petri nets. In: *Petri Net Newsletter* (2012), Nr. 81, S. 3–16

[Kin12b] Kindler, Ekkart: The event coordination notation: execution engine and programming framework. In: *Proceedings of the Fourth Workshop on Behaviour Modelling-Foundations and Applications*, 2012, S. 1–8

[Kin12c] Kindler, Ekkart: Modelling local and global behaviour: Petri nets and event coordination. In: *Transactions on Petri Nets and Other Models of Concurrency VI* (2012), S. 71–93

[Kor25a] Korte, Laszlo: *PetriStation Youtube Channel*. `https://www.youtube.com/@petristation`, 2025. – [Accessed 26-02-2025]

[Kor25b] Korte, Laszlo: *RenewEx Converter Hex Package*. `https://hex.pm/packages/renewex_converter`. Version: 2025. – Last accessed: February 26, 2025

[Kor25c] Korte, Laszlo: *RenewEx Hex Package*. `https://hex.pm/packages/renewex`. Version: 2025. – Last accessed: February 26, 2025

[Kor25d] Korte, Laszlo: *RenewEx IconSet Hex Package*. `https://hex.pm/packages/renewex_iconset`. Version: 2025. – Last accessed: February 26, 2025

[KW99] Kummer, Olaf ; Wienberg, Frank: *Renew – The Reference Net Workshop*. Available at: http://www.renew.de/. `http://www.renew.de/`. Version: März 1999. – Release 1.0

[KWD+23] Kummer, Olaf ; Wienberg, Frank ; Duvigneau, Michael ; Cabac, Lawrence ; Haustermann, Michael ; Mosteller, David: *Renew – The Reference Net Workshop*. `http://www.renew.de/`. Version: Februar 2023. – Release 4.1

[Lam86] Lamport, Leslie: Reactive Programming: The Next Generation of Programming Languages. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1986, 88-96. – Abgerufen am 11. Februar 2025

[Lan16] Lange, Adrian: *GitHub - iig-uni-freiburg/WOLFGANG: Petri Net Editor — github.com*. `https://github.com/iig-uni-freiburg/WOLFGANG` and `https://iig-uni-freiburg.github.io/`, 2016. – [Accessed 15-02-2025]

[LB09] Loop, Charles T. ; Blinn, James F.: *Resolution-independent curve rendering using programmable graphics hardware*. Juli 21 2009. – US Patent 7,564,459

[Leo14] Leopardi, Andrea: *Introduction to Mix & Elixir v1.19.0-dev — hexdocs.pm*. https://hexdocs.pm/elixir/main/introduction-to-mix.html, 2014. – [Accessed 11-02-2025]

[McC78] McCarthy, John: History of LISP. In: *History of programming languages*. 1978, S. 173–185

[MDN24] MDN Web Docs: *CSS Cascade – How CSS Selects Styles*. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_cascade/Cascade. Version: 2024. – Accessed: 2025-02-26

[MDW+24] Manach, Alexis ; Durman, David ; Williams, James ; Liker, Marcel ; Hozak, Marek ; Kaněra, Martin ; Brückner, Roman ; Talas, Vladimir: *JavaScript diagramming library for interactive UIs – JointJS — jointjs.com*. https://www.jointjs.com/, 2024. – [Accessed 17-02-2025]

[ME11] Muskała, Michał ; Elixir Community: *Ecto.Multi Ecto v3.12.5 — hexdocs.pm*. https://hexdocs.pm/ecto/Ecto.Multi.html, 2011. – [Accessed 26-02-2025]

[ME23a] McCord, Chris ; Elixir Community: *Channels Phoenix v1.7.20 — hexdocs.pm*. https://hexdocs.pm/phoenix/channels.html, 2023. – [Accessed 26-02-2025]

[ME23b] McCord, Chris ; Elixir Community: *Presence Phoenix v1.7.20 — hexdocs.pm*. https://hexdocs.pm/phoenix/presence.html, 2023. – [Accessed 26-02-2025]

[Mey23] Meyer, Leon: *Untersuchung des Anbindens mehrerer Oberflächen an RENEW*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, März 2023

[MFP91] Meijer, Erik ; Fokkinga, Maarten ; Paterson, Ross: Functional programming with bananas, lenses, envelopes and barbed wire. In: *Conference on functional programming languages and computer architecture* Springer, 1991, S. 124–144

[MHMS17] Möller, Pascale ; Haustermann, Michael ; Mosteller, David ; Schmitz, Dennis: Simulating Multiple Formalisms Concurrently Based on Reference Nets. In: Moldt, Daniel (Hrsg.) ; Cabac, Lawrence (Hrsg.) ; Rölke, Heiko (Hrsg.): *Petri Nets and Software Engineering. International Workshop, PNSE'17, Zaragoza, Spain, June 25-26, 2017. Proceedings* Bd. 1846, CEUR-WS.org, 2017 (CEUR Workshop Proceedings). – ISSN 1613–0073, 137–156

[Mic21] Microsoft: *Silverlight 5 - Product Lifecycle*. https://learn.microsoft.com/en-us/lifecycle/products/silverlight-5. Version: 2021. – Last accessed: February 11, 2025

[MJ17] Meadows-Jönsson, Eric: *Mix usage — hex.pm*. https://hex.pm/docs/usage, 2017. – [Accessed 11-02-2025]

[MOC13] Mane, Dashrath ; Ojha, Namrata ; Chitnis, Ketaki: The spring framework: An open source java platform for developing robust java applications. In: *International Journal of Innovative Technology and Exploring Engineering* 3 (2013), Nr. 2

[Möl16] Möller, Pascale: *Untersuchung der Umsetzung von Automaten in Renew*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, 2016

[Möl18] Möller, Pascale: *Modellierung und Simulation von Statecharts basierend auf Referenznetzen*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Masterarbeit, 2018

[Moz17] Mozilla: *Firefox Roadmap for Flash End-of-Life*. https://blog.mozilla.org/futurereleases/2017/07/25/firefox-roadmap-flash-end-life/. Version: 2017. – Last accessed: February 11, 2025

[Moz25] Mozilla Developer Network (MDN): *Canvas API*. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. Version: 2025. – Abgerufen am 11. Februar 2025

[Müh20] Mühlke, Sibylle: *Adobe Photoshop*. Rheinwerk Verlag, 2020

[Mül16] Müller, Eva: *Bereitstellung eines JavaScript-basierten Frameworks für die Entwicklung von agentenorientierten Webanwendungen*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, September 2016

[MW15] Matsuda, Kazutaka ; Wang, Meng: Applicative bidirectional programming with lenses. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, S. 62–74

[OA10] Owens, Mike ; Allen, Grant: *SQLite*. Apress LP New York, 2010

[Ope16] OpenJDK: *JEP 289: Deprecate the Applet API for Removal*. https://openjdk.org/jeps/289. Version: 2016. – Last accessed: February 11, 2025

[OSK16] Oswal, Savan ; Singh, Anjali ; Kumari, Kirthi: Deflate compression algorithm. In: *International Journal of Engineering Research and General Science* 4 (2016), Nr. 1, S. 430–436

[Pat24] Patel, Amit: *Signed Distance Field Fonts - basics — redblobgames.com*. https://www.redblobgames.com/x/2403-distance-field-fonts/, 2024. – [Accessed 26-02-2025]

[Pau10] Paulson, Lawrence C.: *ML for the Working Programmer*. 3rd. Cambridge, UK : Cambridge University Press, 2010. – Dritte Auflage mit erweiterten Beispielen und Anwendungen in der realen Welt.

[PB25] Pressler, Ron ; Bateman, Alan: *JEP 453: Structured Concurrency (Preview)*. Website. https://openjdk.org/jeps/453. Version: 2025

[Per09] Perry, J S.: *Log4J*. Ö'Reilly Media, Inc.", 2009

[Pet77a] Petri, C. A.: Modelling as a Communication Discipline. In: Beilner, Heinz (Hrsg.) ; Gelenbe, Erol (Hrsg.): *Measuring, Modelling and Evaluating Computer Systems, Proceedings of the Third International Symposium, Bonn - Bad Godesberg, Germany, October 3-5, 1977*, North-Holland, 1977. – ISBN 0–444–85058–9, S. 435–449

[Pet77b] Petri, Carl A.: Communication Disciplines. In: Shaw, B. (Hrsg.): *Computing System Design: Proc. of the Joint IBM University of Newcastle upon Tyne Seminar, Sep., 1976*, University of Newcastle upon Tyne, 1977, S. 171–183

[Pet77c] Petri, Carl A.: General Net Theory. In: Shaw, B. (Hrsg.): *Computing System Design: Proc. of the Joint IBM University of Newcastle upon Tyne Seminar, Sep., 1976*, University of Newcastle upon Tyne, 1977, S. 131–169

[Pil24] Pillet, Florent: *intro-to-reactive-programming/intro-to-reactive-programming.md at master · fpillet/intro-to-reactive-programming — github.com*. https://github.com/fpillet/intro-to-reactive-programming/blob/master/intro-to-reactive-programming.md, 2024. – [Accessed 26-02-2025]

[Pos81] Postel, Jon: *Transmission Control Protocol*. https://tools.ietf.org/html/rfc793. Version: 1981. – RFC 793, Abgerufen am 11. Februar 2025

[Qui22] Quilez, Inigo: *Inigo Quilez — iquilezles.org*. https://iquilezles.org/articles/distfunctions/, 2022. – [Accessed 26-02-2025]

[RFS+13a] Rehwaldt, Nico ; Fromme, Philipp ; Serra, Valentin ; Robbins, Calvin ; Danielak, Jarek ; Ahad, Abdul ; Konopski, Michał: *GitHub - bpmn-io/bpmn-js: A BPMN 2.0 rendering toolkit and web modeler. — github.com*. https://github.com/bpmn-io/bpmn-js, 2013. – [Accessed 17-02-2025]

[RFS+13b] Rehwaldt, Nico ; Fromme, Philipp ; Serra, Valentin ; Robbins, Calvin ; Danielak, Jarek ; Ahad, Abdul ; Konopski, Michał: *GitHub - bpmn-io/cmmn-js: A CMMN 1.1 rendering toolkit and web modeler. — github.com*.

https://github.com/bpmn-io/cmmn-js, 2013. – [Accessed 17-02-2025]

[RFS+13c] Rehwaldt, Nico ; Fromme, Philipp ; Serra, Valentin ; Robbins, Calvin ; Danielak, Jarek ; Ahad, Abdul ; Konopski, Michał: *GitHub - bpmn-io/dmn-js: View and edit DMN diagrams in the browser. — github.com*. https://github.com/bpmn-io/dmn-js, 2013. – [Accessed 17-02-2025]

[RFS+13d] Rehwaldt, Nico ; Fromme, Philipp ; Serra, Valentin ; Robbins, Calvin ; Danielak, Jarek ; Ahad, Abdul ; Konopski, Michał: *GitHub - bpmn-io/form-js: View and visually edit JSON-based forms. — github.com*. https://github.com/bpmn-io/form-js, 2013. – [Accessed 17-02-2025]

[RFS+13e] Rehwaldt, Nico ; Fromme, Philipp ; Serra, Valentin ; Robbins, Calvin ; Danielak, Jarek ; Ahad, Abdul ; Konopski, Michał: *Web-based tooling for BPMN, DMN, CMMN, and Forms | bpmn.io — bpmn.io*. https://bpmn.io/, 2013. – [Accessed 17-02-2025]

[Ric07a] Richard, D: *About SQLite — sqlite.org*. https://www.sqlite.org/about.html, 2007. – [Accessed 11-02-2025]

[Ric07b] Richard, D: *Most Widely Deployed SQL Database Engine — sqlite.org*. https://www.sqlite.org/mostdeployed.html, 2007. – [Accessed 11-02-2025]

[Ric07c] Richard, D: *SQLite Is Serverless — sqlite.org*. https://www.sqlite.org/serverless.html, 2007. – [Accessed 11-02-2025]

[Ric07d] Richard, D: *SQLite Is Transactional — sqlite.org*. https://www.sqlite.org/transactional.html, 2007. – [Accessed 11-02-2025]

[Ric19] Richter, Marc: *Entwicklung einer modularen JavaScript-Bibliothek zur Modellerstellung*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Bachelorarbeit, 2019

[RR19] Reisig, Wolfgang (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Carl Adam Petri: Ideas, Personality, Impact*. Springer, 2019 https://doi.org/10.1007/978-3-319-96154-5. – ISBN 978–3–319–96153–8

[RWL+03] Ratzer, Anne V. ; Wells, Lisa ; Lassen, Henry M. ; Laursen, Mads ; Qvortrup, Jacob F. ; Stissing, Martin S. ; Westergaard, Michael ; Christensen, Søren ; Jensen, Kurt: CPN tools for editing, simulating, and analysing coloured Petri nets. In: *International conference on application and theory of petri nets* Springer, 2003, S. 450–462. – https://cpntools.org/

[SBMP08] Steinberg, Dave ; Budinsky, Frank ; Merks, Ed ; Paternostro, Marcelo: *EMF: eclipse modeling framework*. Pearson Education, 2008

[Sch03] Schumacher, Jörn: *Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, Oktober 2003

[SS18] SmartDraw Software, LLC: *Petri Nets - Place & Transition Systems — smartdraw.com*. https://www.smartdraw.com/other-software-diagrams/examples/petri-nets-place-andtransition-systems/, 2018. – [Accessed 15-02-2025]

[Sta24] Staltz, André: *The introduction to Reactive Programming you've been missing — gist.github.com*. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754, 2024. – [Accessed 26-02-2025]

[TGPM17] Torres, Alexandre ; Galante, Renata ; Pimenta, Marcelo S. ; Martins, Alexandre Jonatan B.: Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. In: *information and software technology* 82 (2017), S. 1–18

[Tho10] Thompson, Simon: *Haskell: The Craft of Functional Programming*. 3rd. Boston, MA : Addison-Wesley, 2010. – Behandelt funktionale Programmierung mit Haskell

[TMV19] Tate, Bruce ; McCord, Chris ; Valim, Jose: Programming phoenix 1.4: Productive |> reliable |> fast. (2019)

[TS95] Tanenbaum, Andrew S. ; Steen, Maarten van: *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ : Prentice Hall, 1995. – Chapter 1 covers Client-Server Architecture

[Unb24] Unbekannt: *Petri net editor — pes.vsb.cz*. https://pes.vsb.cz/petrineteditor/#/model, 2024. – [Accessed 15-02-2025]

[VF21] Verbeek, Eric ; Fahland, Dirk: CPN IDE: An extensible replacement for CPN tools that uses access/CPN. In: *3rd International Conference on Process Mining, ICPM 2021* CEUR-WS.org, 2021, S. 29–30

[Voß24] Voß, Lukas: *Prototyping Renew as a System of Microservices*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Masterarbeit, August 2024

[VR+07] Van Rossum, Guido u. a.: Python programming language. In: *USENIX annual technical conference* Bd. 41 Santa Clara, CA, 2007, S. 1–36

[VWW96] Virding, Robert ; Wikström, Claes ; Williams, Mike: *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996

[Wal13] Walke, Jordan: *React — react.dev*. `https://react.dev/`, 2013. – [Accessed 17-02-2025]

[WG99] Walsh, Aaron E. ; Graham, Scott D.: *Dynamic HTML: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, 1999. – Behandelt die Grundlagen von DHTML und seine Anwendung im Webdesign

[WH00] Wan, Zhanyong ; Hudak, Paul: Functional reactive programming from first principles. In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, S. 242–252

[WHA25] WHATWG: *HTML Living Standard*. `https://html.spec.whatwg.org/`. Version: 2025. – Abgerufen am 11. Februar 2025

[Wig12] Wiggins, David P.: *Xvfb—virtual framebuffer X server for X Version 11*. `https://manpages.debian.org/bookworm/xvfb/Xvfb.1.en.html`, 2012. – [Accessed 26-02-2025]

[Win18] Wincierz, Martin: *Verbesserung der Erweiterbarkeit und Benutzbarkeit der grafischen Oberfläche des Petrinetz Simulators Renew*. Vogt-Kölln Str. 30, D-22527 Hamburg, Universität Hamburg, Fachbereich Informatik, Masterarbeit, 2018

[WMH18] Wincierz, Martin ; Moldt, Daniel ; Haustermann, Michael: Improvement of the Renew Editor User Interface. In: Lorenz, Robert (Hrsg.) ; Metzger, Johannes (Hrsg.): *Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg*, 2018 (Reports / Technische Berichte der Fakultät für Angewandte Informatik der Universität Augsburg 2018-02), 55–56

[WMJ19] Wilson, Darin ; Meadows-Jonsson, Eric: Programming Ecto: Build Database Apps in Elixir for Scalability and Performance. (2019)

[Wor96] World Wide Web Consortium (W3C): *Cascading Style Sheets (CSS) 1.0 Specification*. `https://www.w3.org/TR/1996/REC-css1-19961217`. Version: 1996. – W3C Recommendation, Abgerufen am 11. Februar 2025

[Wor11a] World Wide Web Consortium (W3C): *Scalable Vector Graphics (SVG) 1.1: CSS Styling*. `https://www.w3.org/TR/SVG11/styling.html`. Version: 2011. – W3C Recommendation, Abgerufen am 11. Februar 2025

[Wor11b] World Wide Web Consortium (W3C): *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. `https://www.w3.org/TR/SVG2/`. Version: 2011. – W3C Recommendation, Abgerufen am 11. Februar 2025

[Wor12] World Wide Web Consortium (W3C): *Server-Sent Events*. https://www.w3.org/TR/eventsource/. Version: 2012. – W3C Recommendation, Abgerufen am 11. Februar 2025

[Wor18] World Wide Web Consortium (W3C): *Web Real-Time Communication (WebRTC) 1.0: Real-Time Communication Between Browsers*. https://www.w3.org/TR/webrtc/. Version: 2018. – W3C Recommendation, Abgerufen am 11. Februar 2025

[You10] Young, Greg: CQRS documents. In: *cqrs. files. wordpress. com* (2010)

[You14] You, Evan: *Vue.js — vuejs.org*. https://vuejs.org/, 2014. – [Accessed 17-02-2025]

[ZCZ+14] Zou, Yunxiao ; Chen, Zhenyu ; Zheng, Yunhui ; Zhang, Xiangyu ; Gao, Zebao: Virtual DOM coverage for effective testing of dynamic web applications. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, S. 60–70

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe. Sofern im Zuge der Erstellung der vorliegenden Abschlussarbeit generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten Wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereichs einverstanden.

Hamburg, den _____ Unterschrift: