

BACHELORTHESIS
Adrian Helberg

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Adrian Helberg

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Michael Neitzke

Eingereicht am: 16. März 2021

Adrian Helberg

Thema der Arbeit

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

Stichworte

Verzweigungsstruktur, 2D Generierung, L-System, Template, Prozedurale Modellierung, Inverse Prozedurale Modellierung, Baumstruktur, Formale Grammatik, Ähnlichkeit

Kurzzusammenfassung

Prozedurale Modellierung beschreibt effiziente Methoden zur Erzeugung einer Vielzahl an Modellen nach bestimmten Regeln. Die Erstellung eines Systems zur Umsetzung solcher Methoden ist durch einen Mangel an Kontrolle und einer geringen Vorhersagbarkeit der Ergebnisse erschwert. Diese Bachelorarbeit präsentiert ein System zur Synthese von Verzweigungsstrukturen, die einer benutzerdefinierten Struktur ähnlich sind. Dabei wird gezeigt, wie sich aktuelle Ansätze der Forschung in einem Programm umsetzen lassen. Templates werden eingelesen und vom Benutzer zu einer Basisstruktur organisiert. Anschließend wird ein L-System über die Topologie dieser Struktur inferiert, komprimiert und dann generalisiert. Nach der Interpretation des L-Systems können vom Benutzer gesetzte Transformationsparameter aus einer Häufigkeitsverteilung angewendet werden. Zum Schluss wird die resultierende Verzweigungsstruktur visualisiert.

Adrian Helberg

Title of Thesis

Template-based synthesis of branching structures using L-Systems

Keywords

Branching Structure, 2D Generation, L-System, Template, Procedural Modeling, Inverse Procedural Modeling, Tree Structure, Formal Grammar, Similarity

Abstract

Procedural modeling describes efficient methods for creating various models according to certain rules. The creation of a system to implement such methods is complicated due to a lack of control and a low predictability of the results. This bachelor thesis presents a system for synthesizing branching structures that are similar to custom structures created by a user. It is shown how current research approaches can be implemented. A basic structure is build from templates by the user. Subsequently an L-System is inferred from the topology of this structure, compressed and then generalized. User-defined transformation parameters from a frequency distribution can be applied to the interpretation of this L-System. Finally, the resulting branching structure is visualized.

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Listings	ix
1 Einleitung	1
1.1 Problemstellung	2
1.2 Ziele	3
1.3 Methodik	3
1.4 Aufbau	5
2 Grundlagen	6
2.1 Grundbegriffe	6
2.2 Grundlegende Arbeiten	7
2.3 Verwandte Arbeiten	11
3 Konzepte	13
3.1 Probleme & Lösungsansätze	13
3.2 Workflows & Algorithmen	16
3.3 Softwaretechnik	22
3.4 Softwarearchitektur	23
4 Implementierung	32
4.1 Projektstruktur	32
4.2 Technologien	34
4.3 Konzeptumsetzung	35
5 Evaluierung	42
6 Fazit und Ausblick	51
Literaturverzeichnis	53

A Anhang	56
Selbstständigkeitserklärung	76

Abbildungsverzeichnis

1.1	Systemarchitektur	5
3.1	Zyklische Umsetzung des Softwareprojektes	23
3.2	System und Systemumgebung	26
3.3	Interaktion zwischen System und Systemumgebung	26
3.4	Subsysteme mit fachlichen Abhängigkeiten	27
3.5	Subsystem GUI	28
3.6	Laufzeitsicht	29
3.7	Infrastruktur Windows-PC	30
4.1	Softwareprojekt Dateistruktur	33
4.2	Softwarepakete mit zugehörigen Funktionen	33
4.3	Startskript Generator.bat	35
4.4	Programm nach Ausführung des Start-Skripts	35
4.5	Templates-Datei zum Einlesen der Template-Strukturen	36
4.6	Erster Anker ist vorselektiert (links) & ein gesetzter Parameter (rechts)	37
4.7	Auswahl des ersten Templates	37
4.8	Der Verzweigungsstruktur hinzugefügte Template-Instanz	38
4.9	Vom Benutzer fertiggestellte Verzweigungsstruktur	39
4.10	Verzweigungsstruktur & dazugehörige Baumstruktur	40
5.1	templates.txt	43
5.2	Grafische Benutzeroberfläche nach der Erstellung einer Basisstruktur	44
5.3	Baumstruktur der erstellten Verzweigungsstruktur	45
5.4	Inferiertes L-System	46
5.5	Komprimierter Baum	47
5.6	Synthetisierte Verzweigungsstrukturen	49
5.7	Untersuchung der synthetisierten Verzweigungsstrukturen	50

A.1	Baumstruktur	56
A.2	Pipeline	57
A.3	Subsysteme der Generierungs-Pipeline	58
A.4	Generierte Verzweigungsstrukturen	58
A.5	Qualitätsbaum	75

Listings

3.1	Erstellen einer Verzweigungsstruktur	16
3.2	Inferieren eines L-Systems aus einer Baumstruktur	17
3.3	Erstellen eines kompakten L-Systems mit Gewichtung w_l	19
3.4	Kostenfunktion C_i mit Gewichtung w_l	19
3.5	Längenfunktion L für Grammatiken	20
3.6	Grammar Edit Distance	20
3.7	Kostenfunktion C_g mit Gewichtung w_0	20
3.8	Generalisieren eines L-Systems mit Gewichtung w_0	21
A.1	Klasse <code>TreeNode</code> zur Erstellung einer Baumstruktur	59
A.2	Klasse <code>TreeNodeIterator</code> als Iterator für die Baumstruktur	59
A.3	Pipeline Klasse zur Organisation von Prozessen	60
A.4	Pipe Interface als Vorlage zur Erstellung eines Teilprozesses einer Pipeline	60
A.5	Erstellen des Pipeline Kontextes	61
A.6	Inferer Klasse zur Inferierung eines L-Systems aus einer Baumstruktur	61
A.7	InfererPipe Klasse als Teilprozess der Pipeline	62
A.8	Inferierungsalgorithmus der Inferer Klasse	62
A.9	Klasse <code>Compressor</code> zur Komprimierung eines L-Systems	64
A.10	Komprimierungsalgorithmus der <code>Compressor</code> Klasse	64
A.11	Algorithmus zum Finden eines maximalen Unterbaums	66
A.12	Algorithmus zum Finden aller Vorkommen eines Unterbaums	67
A.13	Funktion zur Berechnung der Kosten eines L-Systems	67
A.14	Funktion zur Ermittlung der Anzahl Anwendungen einer Produktionsregel	68
A.15	Generalizer Klasse zur Generalisierung eines L-Systems	68
A.16	Generalisierungsalgorithmus der Generalizer Klasse	69
A.17	Kostenfunktion zum Vergleich zweier L-Systeme	70
A.18	Längenfunktion über ein L-System	70
A.19	Grammar Edit Distance	70

A.20 String Edit Distance	71
A.21 Modules Klasse mit Funktion zur Ermittlung der String Edit Distance . .	72
A.22 Estimator Klasse zur Erstellung einer Verteilung über Transformationspa- rameter	72
A.23 Verteilungsalgorithmus der Estimator Klasse	73
A.24 Abrufen eines zufälligen Parameters aus der Verteilung für ein Template .	73
A.25 Berechnung des Durchschnitts eines Parameters für ein Template	74

1 Einleitung

Effizientes Objekt-Design und -modellierung sind entscheidende Kernfunktionalitäten in verschiedenen Bereichen der digitalen Welt. Da die Erstellung geometrischer Objekte für Laien unintuitiv ist und ein großes Maß an Erfahrung und Expertise erfordert, ist dieses stetig komplexer werdende Feld für Neueinsteiger nur sehr schwer zu erschließen. Die Forschung liefert hierzu einige Arbeiten zur prozeduralen Modellierung, um digitale Inhalte schneller und automatisiert zu erstellen. Gerade wenn es um die Darstellung natürlicher Umgebung geht, ist die Erstellung von ähnlichen Objekten, wie zum Beispiel verschiedene Bäume derselben Gattung eines Waldes, ein schwieriges Problem. Kleine Änderungen in prozeduralen Systemen können zu großen Veränderungen der Ergebnisse führen. Darum beschäftigt sich die inverse prozedurale Modellierung unter anderem mit dem Inferieren von Regeln und Mustern aus gegebenen Objekten, um diese nach bestimmten Regeln zu modellieren. Ein wichtiges Werkzeug hierbei ist die Verwendung formaler Grammatiken als fundamentale Datenstruktur der Informatik, um Strukturen zu beschreiben. Eine spezielle Untergruppe sind die L-Systeme, die häufig bei der Beschreibung von Verzweigungsstrukturen und Selbstähnlichkeit zum Einsatz kommen.

Diese Arbeit beschäftigt sich mit der Erstellung eines prozeduralen Systems zur Synthese von Ähnlichkeitsabbildern. Hierzu soll über eine Benutzerschnittstelle eine Struktur erzeugt werden, aus der ein parametrisiertes L-System inferiert werden kann, welches dann zur Generierung von ähnlichen Strukturen genutzt werden kann.

1.1 Problemstellung

Während Methodiken zur Kodifizierung bestimmter Strukturen in den Bereich der prozeduralen Modellierung fällt, findet das Ableiten von Regeln Anwendung in der inversen prozeduralen Modellierung. Mit dem digitalen und naturwissenschaftlichen Fortschritt steigt die Anwendung immer komplexerer Strukturen, die ein Herausarbeiten der schwer zu kontrollierenden, prozeduralen Regeln immer schwieriger machen.

Ein Beispiel hierzu aus der Gaming-Industrie ist die Verwendung unorganisierter Modelle. Diese Modelle weisen keine hierarchische Struktur auf und werden erst durch Datenstrukturen, wie bspw. Octrees organisiert. Unorganisierte Modelle sind nur bedingt wiederverwendbar, da nur die vorliegende Modellierung genutzt werden kann. Es besteht eine hohe Speicherkomplexität bei geringer Laufzeitkomplexität. Für kleinste Veränderungen am Modell ist eine erneute Modellierung erforderlich, die wiederum Speicher benötigt, um sie persistent speichern zu können. Heutzutage werden die Objekte nach bestimmten Kriterien organisiert, um eine automatisierte Modellierung durch Algorithmen zu ermöglichen, und so aus einer Grundstruktur weitere Modelle zu erzeugen. Speicher- und Laufzeitkomplexität nähern sich an. Aus diesem Grund werden allgemeingültige, vielseitig anwendbare Algorithmen gesucht, die bestimmte natürliche Eigenschaften von Strukturen herausarbeiten (Reverse Engineering), um diese für die (inverse) prozedurale Modellierung zur Verfügung zu stellen.

Diese Arbeit soll zeigen, wie sich durch die Erstellung eines Systems zur Generierung von ähnlichen Strukturen aus einer Basistruktur, aktuelle Ansätze aus der Forschung in einem Programm umsetzen lassen.

1.2 Ziele

Die Erstellung eines Systems zur Synthese von ähnlichen Strukturen aus einer Basisstruktur soll zentrale Aufgabe dieser Arbeit sein. Zunächst wird ein System benötigt, das eine Verzweigungsstruktur bestimmt, aus der die ähnlichen Strukturen erzeugt werden können. Um bestimmte Methodiken und Algorithmen der aktuellen Forschung auf die Eingabestruktur anwenden zu können, sollte diese in einer Datenstruktur organisiert werden. Anschließend kann ein L-System aus der Struktur inferiert werden. Die inferierte Grammatik kann durch diverse Algorithmen verändert werden, bis das erzeugte L-System in der Lage ist Ähnlichkeitsstrukturen zu bilden.

1.3 Methodik

Der Benutzer des Systems legt atomare Strukturen in Form von Zeichenketten an, die vom Programm eingelesen und als Templates zur Verfügung gestellt werden. Die Templates sind beliebig und können eine einfache Linie oder eine komplexe Verzweigung darstellen. Werden diese Strukturen auf der graphischen Oberfläche platziert und mit diversen Transformationen verändert, spricht man von Instanzen oder Template-Instanzen.

Strukturieren

Der Benutzer verwendet die grafische Schnittstelle, um aus einzelnen Templates eine zusammenhängende Basisstruktur zu erstellen. Neben der Position der Instanzen werden Transformationsparameter, wie Rotation oder Skalierung, angepasst.

Visualisieren

Der aktuelle Stand der Strukturierung ist jederzeit sichtbar. Liniensegmente und Bindungselemente werden in einem graphischen Element visualisiert und für eine Interaktion zur Verfügung gestellt.

Datenaufbereitung

Das Ergebnis der Strukturierung wird in einer Baumstruktur organisiert, in der jeder Knoten einer bestimmten Template-Instanz entspricht und die eingehende Kante die geometrischen Transformationen relativ zum Elternknoten beschreibt.

Inferieren

Aus der Baumstruktur kann eine formale Grammatik in Form eines L-Systems abgeleitet werden. Diese Grammatik beschreibt lediglich die erstellte Basisstruktur und beinhaltet keine Transformationsparameter, da hier nur auf die Topologie des Baumes und nicht auf geometrische Unterschiede der Instanzen eingegangen wird.

Komprimieren

Um sich wiederholende Produktionsregeln zu vermeiden und so sowohl das Alphabet, als auch die Produktionsregelmenge zu komprimieren, wird die Baumstruktur nach identischen, maximalen Unterbäumen durchsucht und durch zusammengefasste Instanzen vereinfacht. Hierbei gilt die Baumstruktur selbst nicht als Unterbaum.

Generalisieren

Ähnliche Produktionsregeln des L-Systems werden mithilfe einer Kostenfunktion zusammengefasst, um diese mit nicht-deterministischen Regeln zu generalisieren. Die Kostenfunktion entscheidet hierbei über den Grad der Ähnlichkeit zweier Produktionsregeln.

Randomisieren

Jedes Symbol der Grammatik nimmt eine Liste an Parametern entgegen, die nach bestimmten Kriterien pseudo-randomisiert werden, um Variationen von Template-Instanzen zu erstellen. Das Ausführen des L-Systems kann nun Ähnlichkeitsstrukturen für die Basisstruktur erzeugen.

1.4 Aufbau

Die Methodik zum Umsetzen des beschriebenen Systems wird wie folgt umgesetzt.

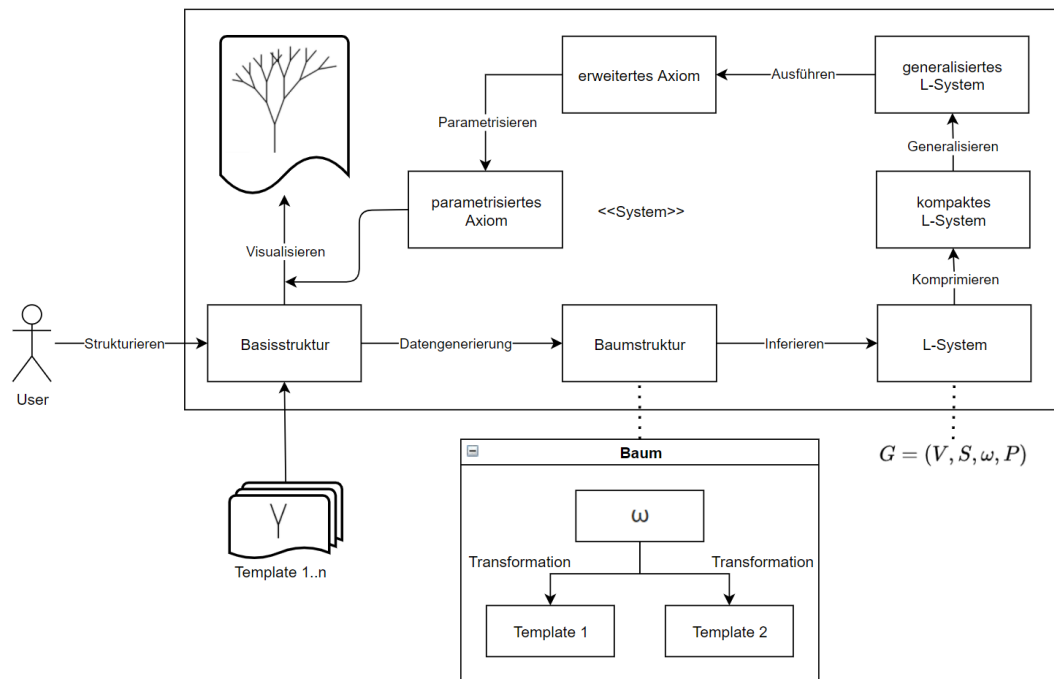


Abbildung 1.1: Architektur des Systems mit einigen Datenstrukturen

Die Darstellung zeigt eine grobe Übersicht der Anwendungsfälle innerhalb des Systems. Der Benutzer strukturiert vom System importierte Templates zu einer Basisstruktur mittels grafischer Bedienelemente. Die Template-Instanzen werden hierbei in einer Baumtopologie organisiert. Anschließend wird ein L-System aus der Baumstruktur generiert und komprimiert. Das kompakte L-System wird generalisiert und ausgeführt. Zuletzt wird das abgeleitete Axiom parametrisiert und sichtbar gemacht.

2 Grundlagen

Die Modellierung mithilfe von Grafiksoftware ist eine vergleichbar händische, langwierige Erstellung von Objekten. Hierbei hat der Designer (Modellierer) die volle Kontrolle über die Strukturen des Objektes.

Bei der prozeduralen Modellierung werden spezifische Strukturen eines zu erstellenden physikalischen Objektes generalisiert und meist über eine Grammatik und globale Parameter abgebildet. Während bei der klassischen Modellierung die menschliche Intuition und bei der prozeduralen Modellierung eine parametrisierte Grammatik genutzt werden kann, arbeitet die inverse prozedurale Modellierung mit bestehenden Modellen. Sie extrahiert („lernt“) die Strukturen des Objekts, die automatisch in eine formale Grammatik überführt werden können. Die Generierung von prozeduralen Modellen ist ein wichtiges, offenes Problem [4].

2.1 Grundbegriffe

Modellierung

Um einen physikalischen Körper in ein digitales Objekt zu überführen, wird mithilfe von Abstraktion (Modellierung) ein mathematisches Modell erstellt, das diesen Körper formal beschreibt. 3D Grafiksoftware, wie bspw. Blender [25], wird genutzt um geometrische Körper zu modellieren, texturieren und zu animieren.

Prozedurale Modellierung

„It encompasses a wide variety of generative techniques that can (semi-)automatically produce a specific type of content based on a set of input parameters“ [21]

Prozedurale Modellierung beschreibt generative Techniken, die (semi-)automatisch spezifische, digitale Inhalte anhand von deskriptiven Parametern erzeugen. SMELIK ET AL. beschreibt einen Prozess, welcher durch das Nutzen globaler Parameter und deskriptiver Regeln Modelle erzeugt [21].

Inverse prozedurale Modellierung

ALIAGA ET AL. spricht bei der inversen prozeduralen Modellierung von dem Finden einer prozeduralen Repräsentation von Strukturen bestehender Modelle [3]. Die Methodik aus Strukturen bestimmte Regeln und Parameter abzuleiten ist der Hauptgegenstand dieses Feldes der Computergrafik und aktueller Gegenstand der Forschung.

2.2 Grundlegende Arbeiten

SMELIK ET AL. untersucht prozedurale Methoden, um diverse Strukturen, wie Vegetation, Straßen u.v.m zu erzeugen. Es wird ein Überblick aktueller, vielversprechender Studien gegeben, deren Anwendung sowohl in technischen Bereichen, als auch in nicht-technischen, kreativen Bereichen, diskutiert wird. Diese Übersicht gilt als grundlegender Einstiegspunkt in die Bereiche der prozeduralen Modellierung [21]. Einen aktuellen Stand der Forschung zur inversen prozeduralen Modellierung (IPM) liefert ALIAGA ET AL.. Es wird gezeigt, dass IPM-Ansätze in entsprechende Kernprobleme der Informatik aufgeteilt und meist getrennt voneinander durch verschiedene Methodiken und Algorithmen bearbeitet werden [3]. Weiter wird ein Einblick in die Kategorisierung und Bewertung einiger Ergebnisse von IPM-Systemem gegeben. DE LA HIGUERA weist darauf hin, dass ein wesentlicher Unterschied zwischen der Induktion einer Grammatik und der Grammatikinferierung besteht [10]. Die Induktion beschreibt das Finden einer Grammatik, welche ein Datum am genauesten beschreibt. Die Grammatikinferierung ist das Finden einer Grammatik, die eine bestimmte Zeichenfolge abdeckt.

DE LA HIGUERA zählt die inverse Generierung von L-Systemen zum Problem der Grammatikinferenz, welches er als gut erforschtes Gebiet beschreibt. Verzweigungsstrukturen im Kontext der inversen prozeduralen Modellierung tauchen in wissenschaftlichen Studien wenig auf. GUO ET AL. adressiert einige IPM-Ansätze in seiner Arbeit [9]:

Die Anwendung von modellierten Bäumen und Landschaften in den Bereichen wie Simulation, VR, Botanik und Architektur wird in der Arbeit von DEUSSEN/LINTERMANN gezeigt [8]. Hier liegt der Fokus auf der Erstellung und Kombination künstlicher Modelle, um natürlich wirkende Umgebungen zu schaffen.

Mit der Erkennung von Wiederholungsmustern in Hausfassaden beschäftigt sich ALHALAWANI ET AL.. Durch das Finden diverser Verformungsparameter mithilfe einer faktorisierten Fassadendarstellung werden neue Bildbearbeitungsmöglichkeiten vorgestellt [2].

Ein virtueller Wiederaufbau der archäologischen Stätte von Pompeji mithilfe der Erstellung von Gebäudemodellen wird von MÜLLER ET AL. vorgestellt.

Das Modellieren ganzer Städte über das CityEngine-System wird in der Arbeit von PARISH/MÜLLER gezeigt. Zunächst werden geografische Bilder eingelesen, aus welchen dann die Anordnung von Straßen, Gärten und Gebäuden ausgelesen werden können. Anschließend können virtuelle Städte nach dem Vorbild der Eingabe erstellt werden.

Sowohl MERRELL ET AL., als auch ZHANG ET AL. stellen Systeme zur Erzeugung von Inneneinrichtung vor. Mithilfe eines Layout-Systems werden Design-Patterns gelernt, in Terme einer Dichtefunktion überführt, um dann virtuelle Möbel zu erstellen [14]. Eine Übersicht diverser Kriterien, die zur Erstellung von Innenraumszenen nötig sind, zeigt die Wichtigkeit über die Auswahl sinnvoller Objekte und deren Anordnung, um plausible Räume zu erstellen [26].

L-Systeme

LINDENMAYER führt eine mathematische Beschreibung zum Wachstum fadenförmiger Organismen ein. Sie zeigt, wie sich der Status von Zellen infolge ein oder mehrerer Einflüsse verhält [12]. Darüber hinaus führt er Systeme ein, die atomare Teile mithilfe von Produktionsregeln ersetzen (Ersetzungssysteme). Diese L-Systeme (Lindenmayer-Systeme) nutzt er zur formalen Beschreibung von Zellteilung. Später werden Symbole zur formalen Beschreibung von Verzweigungen, die von Filamenten abgehen, eingeführt [20]. Die bekanntesten L-Systeme sind zeichenketten-basiert und werden von *Noam Chomsky* eingeführt [7]. Sie ersetzen parallel Symbole eines Wortes, welche von einer Grammatik über eine Sprache akzeptiert werden. L-Systeme können unter anderem parametrisiert oder nicht-parametrisiert und kontextfrei oder kontextsensitiv sein.

L-Systeme sind Grammatiken mit folgender Form:

$$\mathcal{L} = \langle M, \omega, R \rangle, \text{ mit}$$

- M als Alphabet, das alle Symbole enthält, die in der Grammatik vorkommen,
- ω als Axiom oder „Startwort“ und
- R als Menge aller Produktionsregeln, die für \mathcal{L} gelten

Das Alphabet eines parametrisierten Systems enthält Module (Symbole mit Parametern) anstelle von Symbolen:

$$M = \{A(P), B(P), \dots\} \text{ mit}$$

- $P = p_1, p_2, \dots$ als Modulparameter

Zeichen des Alphabets, die Ziel einer Produktionsregel sind, heißen Variablen oder Non-terminale. Alle anderen Zeichen aus M sind Konstanten oder Terminale. Das Axiom ω ist eine nicht-leere Sequenz an Modulen aus M^+ mit

- M^+ als Menge aller möglichen Zeichenketten aus Modulen aus M

Produktionsregeln sind geordnete Paare aus Wörtern über dem Alphabet, die bestimmte Ersetzungsregeln umsetzen. Hierbei werden Symbole aus einem Wort, die einer linken Seite (*engl. left hand side / LHS*) einer Produktionsregel entsprechen, durch die rechte Seite (*engl. right hand side / RHS*) ersetzt. Sie sind folgendermaßen aufgebaut:

$A(P) \rightarrow x, x \in M^*$, mit

- M^* als Menge aller möglichen Zeichenketten von M inklusive der leeren Zeichenkette ε .

Ist die RHS jeder Produktionsregel ein einzelnes Symbol und gibt es zu jeder Variablen eine Regel, spricht man von einem kontextfreien, andernfalls von einem kontextsensitiven L-System.

L-System Interpretation

Lindenmayer-Systeme können Worte über ihr Alphabet interpretieren. Hierzu werden Symbole des Wortes, die Ziel einer Produktionsregel sind, in Iterationen durch die RHS der Produktionsregel ersetzt. Bei der Ausführung eines L-Systems wird kein beliebiges Wort interpretiert, sondern das in der Grammatik definierte Axiom.

Logo-Turtle-Algorithmus

Der Logo-Turtle-Algorithmus [19] setzt ein Vorgehen zur graphischen Beschreibung von L-Systemen, bei dem jeder Buchstabe in einem Wort einer bestimmten Zeichenoperation zugewiesen wird, um. So kann aus einem L-System ein grafisches Muster generiert werden, das mit einer Abfolge von Zeichenbefehlen an eine „Schildkröte“ gezeichnet wird. Das Triplett (x, y, θ) definiert den Status (State) der Schildkröte. Dieser setzt sich aus der aktuellen Position $\begin{pmatrix} x \\ y \end{pmatrix}$ und dem aktuellen Rotationswinkel θ , der die Blickrichtung bestimmt, zusammen.

Der Algorithmus kann als Komprimierung eines geometrischen Musters gesehen werden.

Symbole mit zugehörigen Steuerungsbefehlen und Statusveränderungen:

Symbol	Steuerung	Statusveränderung
$F(d)$	Gehe vom derzeitigen Punkt p_1 d Einheiten in die Blickrichtung zu dem Punkt p_2 . Zeichne ein Liniensegment zwischen p_1 und p_2 .	ja
$+(\alpha)$	Setze neuen Rotationswinkel $\theta = \theta + \alpha$.	ja
$-(\alpha)$	Setze neuen Rotationswinkel $\theta = \theta - \alpha$.	ja
[Lege den aktuellen State auf einen Stack.	nein
]	Hole den State vom Stack.	nein

2.3 Verwandte Arbeiten

Die Erstellung von ähnlichen 3D-Objekten aus komplexen, dreidimensionalen Eingabeobjekten wird in der Arbeit von BOKELOH ET AL. präsentiert [5]. Durch ein „Zerschneiden“ des Eingabeobjektes in mehrere Unterobjekte, ist es möglich ähnliche Modelle aus den Abschnitten zusammensetzen. Hierbei werden Regeln zur Veränderung des Eingabeobjektes gefunden, die eine bestimmte lokale Ähnlichkeit beibehalten. Alle benötigten Informationen werden direkt aus dem Modell und ohne Benutzerinteraktion abgeleitet. Das Lernen von Design-Patterns mithilfe von Bayes-Grammatiken ist Gegenstand der Arbeit von TALTON ET AL.. Ein System zur Generierung geometrischer Modelle und Websites wird hier eingeführt, welches eine organisierte Struktur von bezeichneten Teilmodellen entgegennimmt. Über einen Prozess der MCMC-Optimierung wird eine Bayes-optimale Grammatik erstellt, um neue Modelle zu generieren [24]. Das Markow-Chain-Monte-Carlo-Verfahren (MCMC-Verfahren) ist eine speicherlose, stichprobenartige Auswahl aus Wahrscheinlichkeitsverteilungen eines definierten Bereichs, der für das Erreichen eines bestimmten Ziels von Interesse ist.

Auch STAVA ET AL. etabliert einen MCMC-Ansatz zum Finden prozeduraler Repräsentationen für biologische Bäume. Hier wird zunächst über eine Laplace-Glättung ein Grundgerüst gefunden, das dann in einer Baumtopologie organisiert wird [23].

Das von MARTINOVIC/VAN GOOL eingeführte System nutzt ähnlich organisierte Eingabestrukturen in Form von Gebäudefassaden, um durch bayesische Grammatikinduktion eine kontextfreie Grammatik zu finden [13].

Das Erstellen kontextfreier Grammatiken mithilfe statistischer Methoden zur Verteilung von zweidimensionalen Clustern wird in einer Arbeit von STAVA ET AL. gezeigt [22].

Die meisten Arbeiten im Bereich der inversen prozeduralen Modellierung beschäftigen sich eher mit der Rekonstruktion von gegebenen Modellen, als mit der Generalisierung von Information. Aus diesem Grund führt GUO ET AL. ein Modell zum Lernen von L-Systemen von Verzweigungsstrukturen mithilfe maschinellen Lernens (Deep Learning) anhand beliebiger Grafiken ein [9]. Hierzu werden häufig genutzte Verzweigungen bezeichnet und in zufällige Verzweigungsstrukturen zusammengesetzt. Mit diesen Trainingsdaten wird ein neuronales Netz angelehrt, sodass atomaren Verzweigungen in beliebigen Strukturen erkannt werden können. In einer Eingabestruktur identifiziert das System die atomaren Strukturen und kennzeichnet diese mit einem Rechteck als Begrenzung. Anhand von Überlappungen der Rechtecke kann dann eine hierarchische Baumtopologie aufgebaut werden, aus der ein kompaktes L-System inferiert und generalisiert wird. Das generalisierte L-System wird genutzt, um der Eingabestruktur ähnliche Strukturen zu erzeugen.

Die Algorithmen zur Inferierung und Generalisierung von L-Systemen aus der Arbeit von GUO ET AL. sind Gegenstand dieser Arbeit und werden für die Fragestellung adaptiert und implementiert. Ein neuronales Netz zur Erkennung von Verzweigungsstrukturen liefert die vielversprechendsten Ergebnisse bei der Analyse solcher Strukturen. Da die Implementierung eines neuronalen Netzes für diese Arbeit aus Praktikabilitätsgründen nicht möglich ist, werden die für die Algorithmen benötigten Baumstrukturen während der benutzergesteuerten Erstellung einer Basisstruktur erstellt.

3 Konzepte

Mit der Erstellung eines Programms zur Synthetisierung von Ähnlichkeitsabbildungen einer vom Benutzer erstellten Verzweigungsstruktur soll die Praktikabilität aktueller Forschungsansätze untersucht werden.

Folgende Kernkonzepte werden erläutert und umgesetzt:

- Visualisierung und prozessorientiertes Erstellen von Basisstrukturen,
- Organisation in einer prozessoptimierten, baumähnlichen Topologie,
- L-System-Repräsentationen,
- Algorithmen zur Inferierung, Komprimierung, Generalisierung und
- Verarbeitung von Transformationsparametern

3.1 Probleme & Lösungsansätze

Visualisierung

Um eine geführte Erstellung der Basisstruktur zu ermöglichen, muss diese während der Erstellung sichtbar gemacht werden. Hierzu werden die Templates in Form von Zeichenketten angelegt und mittels Turtle-Grafik visualisiert. Eine Turtle-Grafik beschreibt die Interpretation einer Zeichenkette als Bild durch Ausführen eines Logo-Turtle-Algorithmus. Weiter wird auch zur Evaluation der Ergebnisse eine Visualisierung benötigt. Da die Verzweigungsstrukturen in L-System-Repräsentation vorliegen, wird hierzu eine Interpretationsfunktion benötigt, die diese Ersetzungssysteme in Bildform darstellen. Ein L-System wird durch Ausführung in eine erweiterte Zeichenkette überführt und als Turtle-Grafik beschrieben [19].

Basisstruktur

Der Benutzer verwendet grafische Bendienelemente, um eingeleseene Templates auszuwählen, Transformationsparameter anzupassen und anschließend die Instanzen der Basisstruktur hinzuzufügen. Im Folgenden wird diese Basisstruktur u.a. Grundstruktur oder Eingabestruktur genannt.

Baumstruktur

Um Grundstrukturen mittels verschiedener Algorithmen untersuchen zu können, werden die einzelnen Template-Instanzen in einer baumähnlichen Struktur organisiert. Transformationsparameter einer Instanz beschreiben die räumlichen Veränderungen gegenüber des zugrundeliegenden Templates und haben daher keine Aussagekraft in Bezug auf die Strukturtopologie der Basisstruktur. Der Fokus dieser Arbeit liegt auf topologischen Eigenschaften von Verzweigungsstrukturen (z.B. Rekursion). Darum bilden die einzelnen Template-Instanzen die Knoten der Baumtopologie, während die Kanten die räumlichen Transformationen darstellen. So wird eine datenstrukturelle Trennung zwischen Topologie und räumlicher Transformationen geschaffen. Diese baumähnliche Struktur ist inspiriert durch die Arbeit von GUO ET AL. [9].

Inferieren

Das Smallest Grammar Problem, also das Finden der kleinsten, kontextfreien Grammatik, welche eine bestimmte Zeichenkette generiert, ist ein offenes Problem der Informatik [6]. Primär wird in der Forschung nach Algorithmen gesucht, die ein akzeptables Ergebnis liefern. In dieser Arbeit wird ein Algorithmus präsentiert, der die Knoten der Baumstruktur in einzelne Symbole umwandelt, mit Produktionsregeln verknüpft und diese dem resultierenden L-System hinzufügt. Dieses L-System repräsentiert lediglich die Eingabestruktur.

Komprimieren

Zur Erzeugung eines kompakten, gewichteten L-Systems werden sich wiederholende Unterbäume gesucht und ersetzt. Um das zu erzeugende L-System mit einer kleinen oder großen Regelmenge auszustatten, wird eine Gewichtung angewendet. Eine Kostenfunktion stellt hierbei die Anzahl Symbole aller RHS der Produktionsregeln mit der Menge an Anwendungen der LHS gegenüber.

Generalisieren

Da das kompakte L-System eine Repräsentation der vom Benutzer erzeugten Verzweigungsstruktur darstellt, werden ähnliche Regeln miteinander verbunden (Merge) und mit einer Wahrscheinlichkeit versehen, um nicht-deterministische Regeln hinzuzufügen. Eine weitere Kostenfunktion bewertet den Merge zweier Produktionsregeln. Sie wendet eine Gewichtung über die Länge der Grammatik zur Änderungsdistanz von der alten (ohne Merge) zur neuen Grammatik an.

Transformationen

Die vom Benutzer vergebenen Werte der Transformationsparameter werden während der Erstellung der Eingabestruktur in einer Häufigkeitsverteilung organisiert und bei Ausführung des generalisierten L-Systems angewendet. So werden Transformationen nach ihrer statistischen Häufigkeit eingesetzt.

3.2 Workflows & Algorithmen

Verzweigungsstruktur erstellen

Um als Benutzer des Systems eine Verzweigungsstruktur zu erstellen, wird folgender Arbeitsablauf umgesetzt:

Algorithmus 31: Erstellen einer Verzweigungsstruktur

```
1   Erster Anker ist vorselektiert
2   Wiederhole, bis Struktur fertiggestellt ist:
3     Selektiere ein Template aus der Liste
4     Setze Parameter
5     Bestätige Auswahl und Parameter
6     Zeichne ausgewähltes Template mit Parametern
7     Wähle nächsten Anker aus
```

L-System inferieren

Aus der Verzweigungsstruktur kann nun ein L-System erzeugt werden. Hierzu wird ein neuer Algorithmus (siehe Algorithmus 32) präsentiert:

Zunächst werden L-System-Komponenten erzeugt (Zeile 2-6) für $\mathcal{L} = \langle M, \omega, R \rangle$:

- Das Alphabet wird mit den Symbolen F und S initialisiert, da F als Repräsentation einer grundlegenden Zeichenoperation und S als Axiom in jeder Anwendung des Algorithmus vorkommt.
- Die erste Produktionsregel α umfasst die Abbildung des Axioms auf ein neues Symbol, das nicht im Alphabet vorkommt. Das Alphabet wird stets um ein Symbol lexikografischer Ordnung ergänzt:
 - Bsp.: $\{A, B, C\}$ wird ein neues, unbekanntes Symbol hinzugefügt $\rightarrow \{A, B, C, D\}$
- Die Variable β hält zu untersuchende Knoten der Baumtopologie, welche nach Breitensuche iteriert werden. Der erste Durchlauf startet bei Wurzelknoten S.
- Als letzter Schritt der Initialisierung wird dem Alphabet ein neues Symbol hinzugefügt, das durch die Variable γ gehalten wird.

Die Schleife des Algorithmus beschäftigt sich mit der Iterierung des Baumes und dem Erstellen neuer Symbole und Produktionsregeln für das resultierende L-System (Zeile 8-17):

- Die in den Knoten des Baumes gehaltenen Template-Instanzen entsprechen einer Zeichenkette, die durch eine Turtle-Grafik interpretiert, einem vom Benutzer transformierten Template entspricht. Diese Zeichenkette wird in δ gespeichert.
- In Zeile 9-12 wird die genannte Zeichenkette auf Verzweigungsvariablen ($A - Z$; F ausgeschlossen) untersucht. Ein neues Symbol ersetzt das dem Alphabet hinzugefügte Symbol. Anschließend wird eine Produktionsregel, die auf die veränderte Zeichenkette abbildet, der Produktionsregelmenge hinzugefügt.
- Zeile 13 prüft, ob es ein Symbol im Alphabet gibt, das nicht als Ziel einer Produktionsregel in der Produktionsregelmenge definiert ist. In diesem Fall wird das Symbol als Ziel der nächsten Produktion gesetzt. Andernfalls schließt der Algorithmus ab.
- Das Schleifen-Attribut ist das Setzen des nächsten Knotens am Ende der Schleife (Zeile 17).

Algorithmus 32: Inferieren eines L-Systems aus einer Baumstruktur

```

1   Initialisierung :
2        $M = \{F, S\}$ 
3        $\omega = S$ 
4        $R \leftarrow \{\alpha : S \rightarrow A\}$ 
5        $\beta =$  nächster Knoten
6        $M \leftarrow \gamma \in \{A, B, \dots, Z\}$ , mit  $\gamma \notin M$ 
7   Schleife :
8        $\delta =$  Wort von  $\beta$ 
9        $\forall \{A, B, \dots, Z\} \setminus F \in \delta :$ 
10      Ersetze mit  $\zeta \in \{A, B, \dots, Z\}$ , mit  $\zeta \notin M$ 
11       $M \leftarrow \zeta$ 
12       $R \leftarrow \{\gamma \rightarrow \delta\}$ 
13      Wenn es ein Symbol  $\eta$  in  $M \setminus \{F, S\}$  gibt mit  $\{\eta \rightarrow bel.\} \notin R :$ 
14           $\gamma = \eta$ 
15      Sonst :
16          Breche Schleife ab
17       $\beta =$  nächster Knoten

```

L-System komprimieren

GUO ET AL. führt einen Algorithmus zum Inferieren einer Grammatik aus einer Baumstruktur ein [9]. Zum Einen wird ein L-System aufgebaut, zum Andern die Baumtopologie durch Finden maximaler Subbäume reduziert. Diese Reduktion wird im folgenden Algorithmus (siehe Algorithmus 33) adaptiert:

Initialisierung (Zeile 2-5):

- Das L-System, welches der Eingabe des Algorithmus entspricht, wird ausgeführt und die resultierende Zeichenkette wird in \mathcal{L}^+ gespeichert.
- Ein Parameter w_l wird eingeführt, der eine Kostenfunktion (Zeile 11) nach Anzahl an Symbolen der RHS von Produktionsregeln und die Anzahl deren Anwendung gewichten soll.
- Anschließend wird ein maximaler Subbaum durch geschachtelte Iteration des Baumes T gesucht und als T' gesetzt.
- Es werden ausschließlich maximale Unterbäume behandelt, die mindestens zweimal im Baum vorkommen.

Die Schleife (Zeile 7-13) stellt die Reduzierung dar:

- Zunächst werden alle Vorkommen des maximalen Unterbaumes durch ein neues Symbol ersetzt.
- Aus diesem Subbaum wird ein L-System inferiert, das wiederum in eine erweiterte Zeichenkette ausgeführt wird.
- Die Zeichenkette wird als LHS einer neuen Produktionsregel gesetzt, die auf das neue Symbol abzielt.
- Das alte L-System kann nun mit dem veränderten L-System mittels Kostenfunktion verglichen werden (Zeile 9): Liegen die Kosten des veränderten L-Systems unter den Kosten des Alten, wird die Reduktion beendet. Andernfalls gilt der Subbaum nun als Eingabebaum und das veränderte Ersetzungssystem als Eingabe-L-System.

Algorithmus 33: Erstellen eines kompakten L-Systems mit Gewichtung w_l

```

1 Initialisierung :
2    $\mathcal{L}^+ \leftarrow L_s$ 
3    $\mathcal{L} = \emptyset$ 
4   Setze Gewichtungparameter  $w_l \in [0,1]$ 
5   Finde maximalen Unterbaum  $T'$  aus  $T$  mit Wiederholungen  $n > 1$ 
6 Schleife (Reduzierung):
7   Ersetze alle Vorkommen von  $T'$  mit demselben Symbol  $\gamma \in \{A, B, \dots, Z\}$ 
8    $R \leftarrow \{\gamma \rightarrow L_s\}$  mit  $L_s$  aus  $T'$ ,  $R$  aus  $\mathcal{L}$ 
9   Wenn  $C_i(\mathcal{L}) \geq C_i(\mathcal{L}^+)$ :
10    Breche Schleife ab
11    $T \leftarrow T'$ 
12    $\mathcal{L}^+ \leftarrow \mathcal{L}$ 
13   Finde maximalen Unterbaum  $T'$  aus  $T$  mit Wiederholungen  $n > 1$ 

```

Algorithmus 34: Kostenfunktion C_i mit Gewichtung w_l

$$1 \quad C_i(\mathcal{L}) = \sum_{A(P) \rightarrow M^* \in \mathcal{L}} w_l * |M^*| + (1 - w_l) * N(A(P) \rightarrow M^*)$$

$N(\cdot)$ dient als Zählfunktion für die Anzahl Wiederholungen einer *LHS* in einem ausgeführten L-System.

L-System generalisieren

Da das kompakte L-System eine Repräsentation der vom Benutzer erzeugten Verzweigungsstruktur darstellt, werden nun ähnliche Regeln miteinander verbunden und mit einer Wahrscheinlichkeit versehen, um nicht-deterministische Regeln hinzuzufügen. Sowohl Längenfunktionen, Kostenfunktionen und Distanzalgorithmen, als auch der Grundalgorithmus sind aus einer Arbeit von GUO ET AL. entnommen und bauen sich wie folgt auf [9]:

- Die Längenfunktion L , die auf eine Grammatik angewendet wird, summiert die Anzahl Symbole des Alphabets mit der Anzahl an RHS der Produktionsregeln. Sie berechnet somit die Gesamtheit aller Symbole, die das L-System abbilden soll. (Länge der Grammatik).
- Der Abstand zweier Zeichenketten kann mit der *String Edit Distance* ermittelt werden. Diese wird über die Funktion D_s abgebildet. Hierbei wird die Anzahl an Operationen summiert, die für die Überführung einer Zeichenkette in eine Andere nötig sind (Zeichenkettenaustausch, -einschub und -löschung)

- Mit D_s kann nun auch der Abstand zweier Grammatiken zueinander bestimmt werden. Diese Funktion wird mit D_g abgebildet.
- Die Kostenfunktion C_g nutzt die Länge und Distanz von Grammatiken, um Kosten einer Überführung von einer Grammatik in eine andere messen zu können. Sie berechnet somit die Kosten, um L^* in L^+ zu überführen. Der Parameter w_0 gewichtet hierbei die Differenz der Länge der Grammatiken und die Anzahl Operationen, die zur Überführung nötig sind. Die Überführungskosten werden in der Variable C_g^{old} zur Verfügung gestellt.

Algorithmus 35: Längenfunktion L für Grammatiken

$$1 \quad L(\mathcal{L}) = |M| + \sum_{A(P) \rightarrow M^* \in \mathcal{L}} |M^*|$$

Algorithmus 36: Grammar Edit Distance

$$1 \quad D_g(\mathcal{L}^+, \mathcal{L}^*) = \sum_{(A(P) \rightarrow M_A^*, B(P) \rightarrow M_B^*) \in M(\mathcal{L}^+ \rightarrow \mathcal{L}^*)} D_s(M_A^*, M_B^*)$$

Algorithmus 37: Kostenfunktion C_g mit Gewichtung w_0

$$1 \quad C_g(\mathcal{L}^*, \mathcal{L}^+) = w_0 * (L(\mathcal{L}^*) - L(\mathcal{L}^+)) + (1 - w_0) + D_g(\mathcal{L}^+, \mathcal{L}^*)$$

Algorithmus 38 zum Generalisieren eines L-Systems:

Initialisierung (Zeile 2-4):

- Das zu untersuchende Tuple besteht aus zwei Produktionsregeln (Regelpaar) und wird in der Variable p^* gehalten.
- L-Systeme, die sich infolge eines Merges geändert haben, werden in L^+ und Eingabe-L-Systeme in L^* gespeichert.

Generalisierung (Zeile 6-13):

- \mathcal{P} ist die Menge aller möglichen Regelpaare aus L^* .
- Das Regelpaar, das beim Merge die geringsten Kosten für die Überführung in die neue Grammatik aufweist, wird in p^* gehalten.
- Sind diese Kosten positiv, wird die Generalisierung abgebrochen.
- Andernfalls werden die Variablen c^* als Delta-Kosten, C_g^{old} und L^* entsprechend gesetzt.
- Sollte die Differenz der Überführungskosten positiv sein, wird die Generalisierung abgebrochen.

Algorithmus 38: Generalisieren eines L-Systems mit Gewichtung w_0

```

1  Initialisierung :
2      Regelpaar  $p^* = \emptyset$ 
3       $\mathcal{L}^* = \mathcal{L}^+$ 
4       $C_g^{old} = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*)$ 
5  Schleife :
6      Finde Regelpaar  $p^*$  mit minimalen Kosten  $C_g(\mathcal{L}^* + \{p_i\}, \mathcal{L}^*), \forall p_i \in \mathcal{P}$ 
7      Wenn  $C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*) \geq 0$ :
8          Breche Schleife ab
9       $c^* = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*) - C_g^{old}$ 
10      $C_g^{old} = C_g(\mathcal{L}^* + \{p^*\}, \mathcal{L}^*)$ 
11      $\mathcal{L}^* = \mathcal{L}^* + \{p^*\}$ 
12     Wenn  $c^* > 0$ :
13         Breche Schleife ab

```

3.3 Softwaretechnik

Extreme Programming

Eine Fallstudie der Universität Karlsruhe [16] untersucht den Einsatz der Softwaretechnik Extreme Programming (XP) im Kontext der Erstellung von Abschlussarbeiten im Universitätsumfeld. Hierzu werden folgende Schlüsselpraktiken untersucht:

- XP als Softwaretechnik zur schrittweisen Annäherung an die Anforderungen eines Systems
- Änderung der Anforderungen an das System
- Funktionalitäten (Features) werden als Tätigkeiten des Benutzers (User Stories) definiert
- Zuerst werden Komponententests (Modultests) geschrieben und anschließend die Features (Test-driven Design)
- Keine separaten Testing-Phasen
- Keine formalen Reviews oder Inspektionen
- Regelmäßige Integration von Änderungen
- Gemeinsame Implementierung (Pair Programming) in Zweiergruppen

Aus der Fallstudie geht hervor, dass Extreme Programming einige Vorteile bei der Bearbeitung eines Softwareprojektes einer Bachelorarbeit bietet. Zum Einen können sich Anforderungen an das zu erstellende System durch parallele Literaturrecherche ändern, zum Anderen können Arbeitspakete durch Releases abgebildet werden. Diese Softwaretechnik wird in der Umsetzung des Softwareprojekts zu dieser Arbeit angewendet.

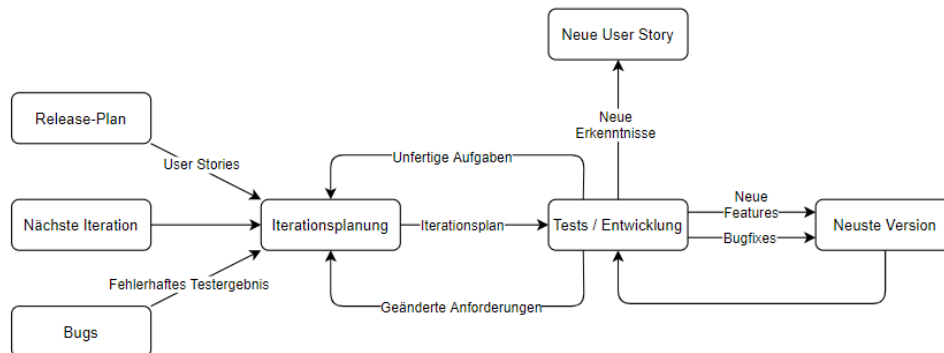


Abbildung 3.1: Zyklische Umsetzung des Softwareprojektes

3.4 Softwarearchitektur

Die Gliederung der Inhalte für die Softwarearchitektur erfolgt nach der arc42-Vorlage [11].

Qualitätsziele

Um die wesentlichen Features des Systems in einem Programm umzusetzen, werden Qualitätsziele in einem Qualitätsbaum (siehe A.5) definiert und Szenarien zugeordnet (Numerierung). Die nichtfunktionalen Anforderungen sind nach der DIN-Norm DIN 66272 strukturiert.

Der Benutzer ist in der Lage eine Verzweigungsstruktur zu erstellen, aus der dann eine oder mehrere ähnliche Strukturen erzeugt werden können **(1)**. Vordefinierte Templates sollen verwendet werden, um die Basisstruktur zu erzeugen **(2)**. Dabei soll der Benutzer visuelles Feedback während des Erstellungsprozesses bekommen **(3)**. Für den Benutzer besteht die Möglichkeit Parameter zu setzen, die die resultierenden Ähnlichkeitsstrukturen beeinflussen, um so zu einem akzeptablen Ergebnis zu gelangen **(4)**. Es soll nicht möglich sein eine ungültige Verzweigungsstruktur zu erstellen **(5)**. Der Erstellungsvorgang soll jeder Zeit neu begonnen werden können, ohne das Programm neu starten zu müssen **(6)**. Weiter sollte das Programm robust gegenüber Benutzereingaben sein **(7)**. Eine intuitive Nutzung des Programms soll gegeben sein, sodass der Benutzer wenig Zeit für die Nutzung der Bedienelemente aufbringen muss **(8)**. Das Erzeugen der Struktur aus Kapitel 5 (Evaluierung) soll ohne händische Nachbildung möglich sein **(9)**.

Zur Nachvollziehbarkeit sollten die verschiedenen L-System-Varianten in der Systemkonsole ausgegeben werden **(10)**. Der Generierungsprozess nach dem Erstellen der Basisstruktur sollte nicht länger als vier Sekunden in Anspruch nehmen **(11)**. Auch Fließkommazahlen sollte der Benutzer als Parameter festlegen können **(12)**. Ein Entwickler soll zur Manipulation von L-Systemen weitere Algorithmen hinzufügen können **(13)**. Das Programm sollte unter eine Java-Laufzeitumgebung auf mehreren Systemen nutzbar sein **(14)**.

Lösungsstrategie

Zur Erreichung der nichtfunktionalen Qualitätsziele werden folgende Architekturansätze umgesetzt. Die funktionalen Ziele werden durch die in Kapitel 3.1 vorgestellten Lösungsansätze abgedeckt.

Qualitätsziel	Architekturansatz
Funktionalität	<ul style="list-style-type: none">- Einlesen von Templates- Grafische Benutzerschnittstelle zur Anordnungen von Templates zu einer Verzweigungsstruktur- Generieren der Baumstruktur- Verarbeiten von L-Systemen
Zuverlässigkeit	<ul style="list-style-type: none">- Durch den User vorgegebene Parameter- Definieren der grafischen Bedienelemente, sodass kein unerwartetes Verhalten auftritt

Qualitätsziel	Architekturansatz
Benutzbarkeit	<ul style="list-style-type: none">- Zeichnen der Verzweigungsstruktur- Zeichnen vorläufiger Änderungen von Parametern an der Struktur- Zurücksetzen des Erstellungsprozesses durch den Benutzer- Intuitive Benennung von Schaltflächen- Schaltfläche zur Erzeugung des Beispiels aus Kapitel 5- Steuern des Generierungsprozesses durch Setzen von Parametern
Effizienz	<ul style="list-style-type: none">- Ausgabe der Teilschritte in der Systemkonsole
Änderbarkeit	<ul style="list-style-type: none">- Nutzen des Pipeline-Design-Patterns- Trennung der grafischen Oberfläche und der Logik
Übertragbarkeit	<ul style="list-style-type: none">- Erstellung einer ausführbaren Java-Archiv-Datei- Sinnvolle Aufteilung von Funktionalitäten auf Dateien und Software-Pakete- Effiziente Datenkapselung- Geschlossene Informationskontexte

Kontextabgrenzung

Die Systemgrenzen werden zum Einen durch die Interaktion mit dem Benutzer, zum Anderen durch die Interaktion mit dem Dateisystem des Host-Systems und dem Zugreifen und Lesen der Template-Dateien definiert. Hierbei wird die Erstellung der Basisstruktur als nicht-technische Interaktion und das Einlesen der Dateien als technische Interaktion gesehen.

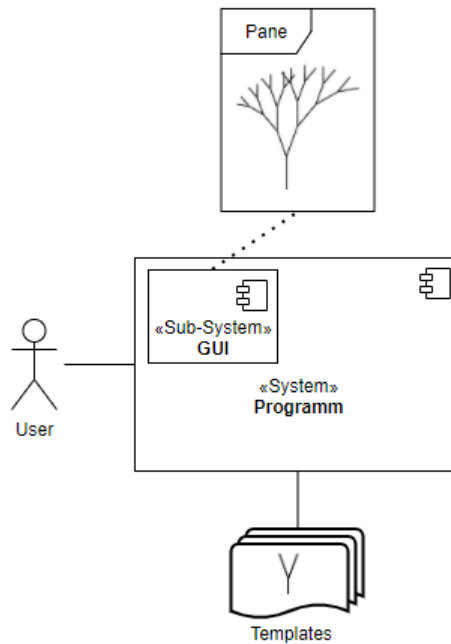


Abbildung 3.2: System und Systemumgebung

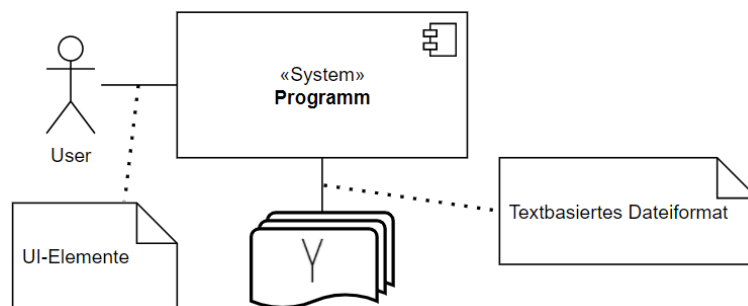


Abbildung 3.3: Interaktion zwischen System und Systemumgebung

Bausteinsicht

Der Benutzer interagiert über das User Interface mit dem Subsystem GUI, das sich mit der Visualisierung, dem Aufbau der Eingabestruktur und dem Anlegen einer internen Baumtopologie beschäftigt. Die Pipeline zum Erzeugen der Ausgabestrukturen beginnt mit dem Inferieren eines L-Systems aus der benutzerdefinierten Struktur und gibt das erzeugte Ersetzungssystem an die folgende Komponente weiter. Hierbei gilt der Pipeline-Kontext, welcher innerhalb der Pipeline an den jeweils nächsten Schritt weitergegeben und dort aktualisiert wird, als Eingabe der Pipeline. Hat die `Estimator`-Komponente eine Verteilung über Transformationsparameter angelegt, gilt der Kontext als Ausgabe der Pipeline.

Ebene 1

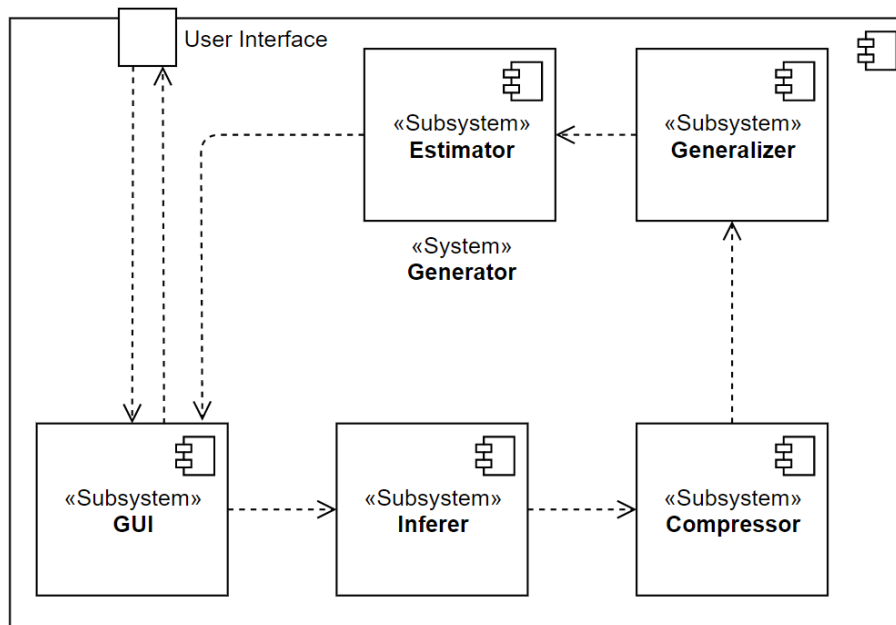


Abbildung 3.4: Subsysteme mit fachlichen Abhängigkeiten

Betrachtet man die GUI-Komponente genauer, setzt sich diese aus vier Subsystemen zusammen. Das Application-Modul beschreibt den Einstiegspunkt des Programms. Die grafische Benutzerschnittstelle wird aus einem View (Scene graph) mit zugehörigem Controller (Model) aufgebaut. Der Controller stellt die Logik für das View zur Verfügung. Neben der Erstellung der Basisstruktur, wird die Baumstruktur in einer separaten Komponente umgesetzt.

Ebene 2

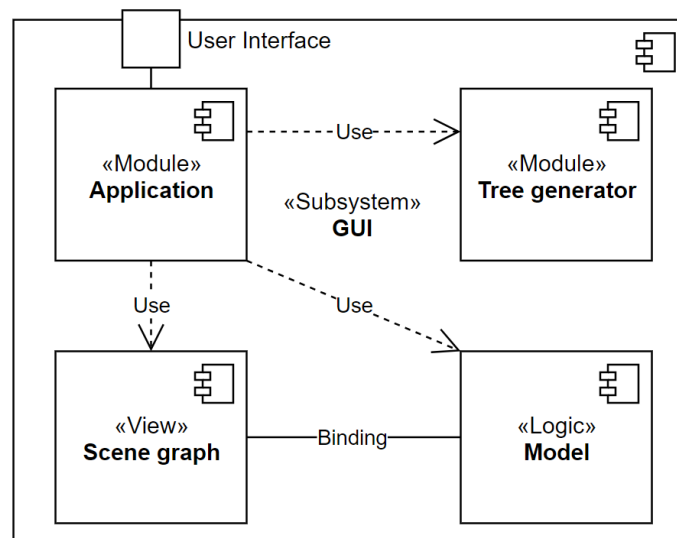


Abbildung 3.5: Subsystem GUI

Laufzeitsicht

Aus der nachfolgenden Abbildung geht hervor, wie sich der Informationsfluss zwischen den einzelnen Subsystemen verhält. Wird der Systemprozess einmal durchlaufen, besteht die Möglichkeit die Ausführung des generalisierten L-Systems mit erneuter Vergabe der Transformationsparameter aus der Häufigkeitsverteilung der Estimator-Komponente zu starten. Der Ablauf des Systems setzt sich wie folgt zusammen:

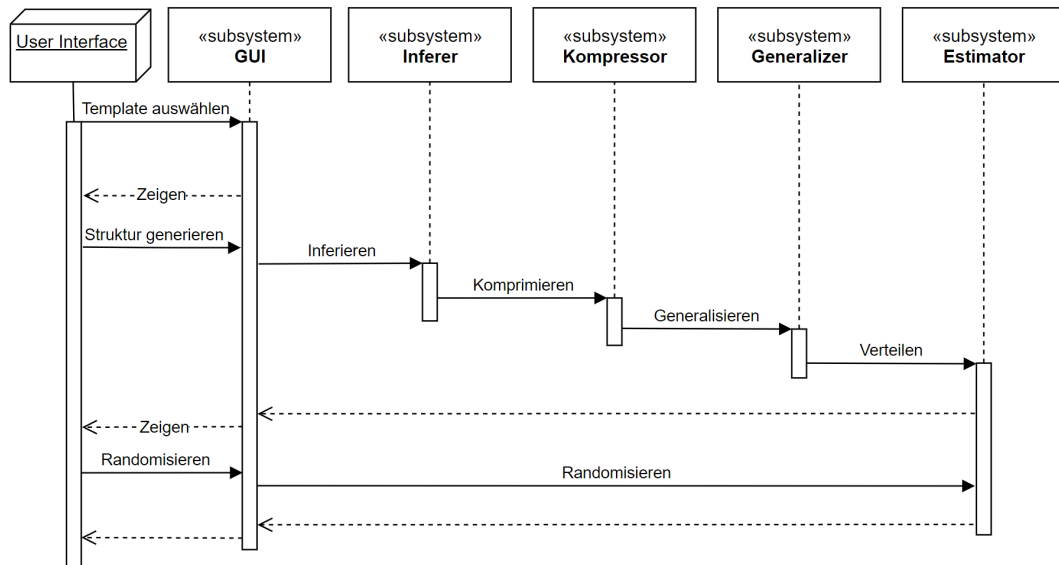


Abbildung 3.6: Laufzeitsicht

Verteilungssicht

Die Ausführung des Programms wird durch ein einfaches Startskript zur Verfügung gestellt. Die folgende Abbildung der Verteilungssicht stellt lediglich die Ausführung auf einer Windows-Maschine dar.

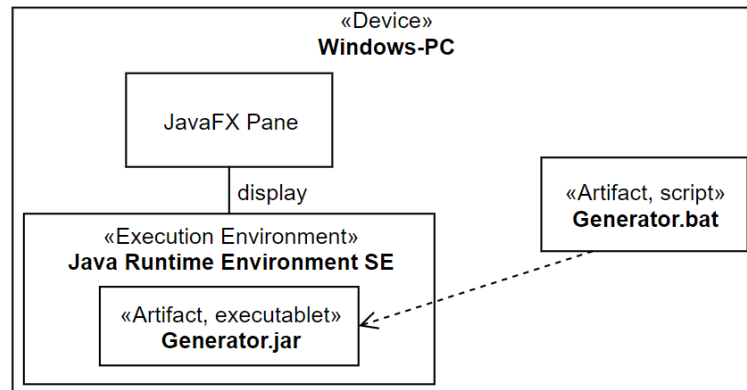


Abbildung 3.7: Infrastruktur Windows-PC

Datenstrukturen

Das wesentliche Datenmodell ist als UML-Diagramm modelliert (siehe Seite 56 Anhang). Die während der Strukturierung entstehende Baumstruktur wird in A.5 gezeigt.

`TreeNode` implementiert das `Iterable`-Interface, sodass durch die Überladung der `iterate`-Funktion ein Iterator zurückgegeben werden kann, der ein Durchgehen der Baumtopologie möglich macht. Dieser Iterator ist durch die Klasse `TreeNodeIterator`, die das `Iterator`-Interface implementiert, definiert. Die überladene `next`-Funktion beschreibt eine Iterierung des Baums durch eine Breitensuche.

A.2 beschreibt den Generierungsprozess im Pipeline-Design-Pattern. Jeder Teilschritt des Prozesses ist durch eine Klasse, die das `Pipe`-Interface nutzt, definiert. Die überladene `process`-Funktion nimmt den `PipelineContext` entgegen, verändert diesen und gibt ihn anschließend zurück. Das `PipelineContext`-Objekt hält die Daten, die die einzelnen Schritte benötigen und zurückgeben (Pipeline-Kontext).

Die Pipes lassen sich in einer festen Reihenfolge durch die überladene `pipe`-Funktion eines `Pipeline`-Objekts einreihen. Die Prozesspipeline wird durch die `execute`-Funktion ausgeführt.

Die verschiedenen Algorithmen zur Manipulation von L-Systemen werden in A.3 durch verschiedene Objekte definiert. Sie werden in der entsprechenden Pipe erstellt und durch eine Funktion ausgeführt, die das veränderte L-System zurückgibt. Im Konstruktor werden benötigte Daten aus dem Pipeline-Kontext an den Algorithmus übergeben.

4 Implementierung

Zur Umsetzung der vorgestellten Konzepte wird im Folgenden auf Softwarepakete, Technologien, Datenspeicherung, Benutzerinteraktion und Arbeitsablauf der erstellten Software eingegangen. Darüber hinaus wird ein Überblick über Quellcode und einige Implementierungsentscheidungen gegeben. Auf Implementierungsdetails zur Nutzung der vorgestellten Technologien wird nicht im Detail eingegangen. Quellcode wird im Wesentlichen gezeigt und anhand von Zeilenangaben (z.B. **5**) erläutert. Softwareumgebungsspezifische Implementierungsdetails werden unter Angabe von drei Punkten (...) weggelassen.

4.1 Projektstruktur

Das Softwareprojekt ist nach der Gradle-Source-Code-Konvention organisiert. Das gesamte Programm befindet sich im Ordner `Generator`, der obligatorische Gradle-Dateien und den Source-Ordner enthält. Sowohl die Quelldateien, also auch die Testdateien sind der Paketstruktur **de.haw** untergeordnet. Die folgende Tabelle gibt einen Überblick über grundlegende Pakete und deren Funktion.

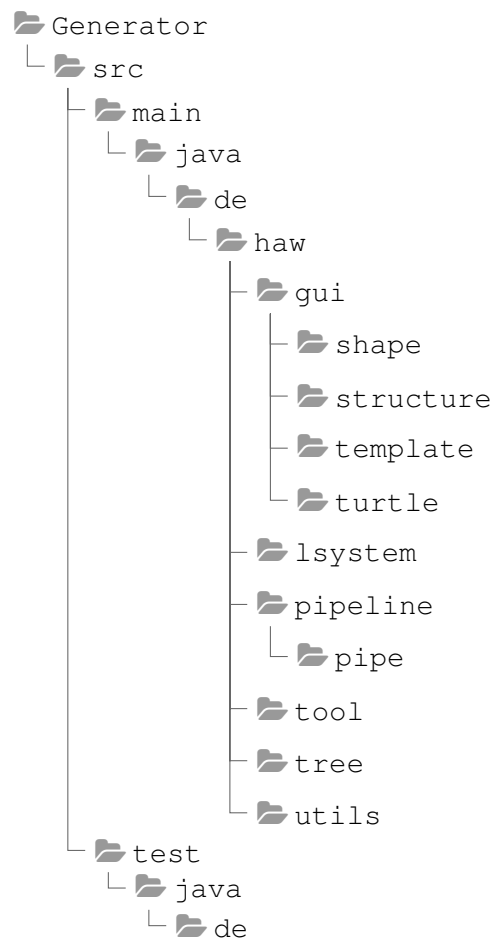


Abbildung 4.1: Softwareprojekt Dateistruktur

Paket	Funktion
<i>gui</i>	Visualisierung des Programms
<i>lssystem</i>	L-System-Repräsentation
<i>pipeline</i>	Umsetzung des Pipeline-Design-Patterns
<i>tool</i>	Methodiken und Algorithmen
<i>tree</i>	Komponenten der Baumstruktur
<i>utils</i>	Hilfskomponenten

Abbildung 4.2: Softwarepakete mit zugehörigen Funktionen

4.2 Technologien

Zur Umsetzung des Softwareprojekts wird eine **Java**-Anwendung für die **Java**-Laufzeitumgebung entwickelt. Sie liegt in der Distribution **Amazon Corretto** in der Version 11.0.2 vor. Grafische Oberflächen werden mit der JavaFX-Spezifikation von **Oracle** in der Version 11.0.2 umgesetzt. Zur Automatisierung von Abhängigkeits- und Buildmanagement wird **Gradle** (Version 6.7) verwendet. Eine Testumgebung, eine Vektorbibliothek, eine Tupelrepräsentation und eine Erweiterung zur mathematischen Standardbibliothek werden über Abhängigkeiten vom Gradle-Framework im Build-Prozess geladen und zur Verfügung gestellt.

Um den test-driven Implementierungsansatz umzusetzen wird **JUnit 5 Jupiter** als Testumgebung genutzt. Sie setzt sich aus einem Programmierschema und einem Erweiterungsmodell zusammen. Das Jupiter-Projekt liefert zudem die Laufzeitumgebung für Softwaretests. **Google Guava** ist eine Bibliothek mit mathematischen Funktionen. Sie wird benötigt, um eine praktikable Lösung zur Mengenmanipulation nutzen zu können (Bsp. Erstellen von Kombinationspaaren einer Menge). Ausschließlich JavaFX wird außerhalb des Projektes installiert und als Modul im Start-Skript des Programmes hinzugefügt.

Weitere Systeme zur Erstellung des Softwareprojekts sind:

- Versionierung via **Git**
- **Dot** zur Visualisierung von Graphen
- **PlantUML** zur Generierung von UML-Diagrammen

4.3 Konzeptumsetzung

Das Skript `Generator.bat`, das zum Starten der Anwendung innerhalb eines Windows-Betriebssystems genutzt werden kann, fügt dem Programm alle externen Module hinzu, die während der Laufzeit genutzt werden, und führt die angegebene jar-Datei aus:

```
1 java --module-path .\javafx-sdk-11.0.2\lib --add-modules
   ➔ javafx.controls,javafx.fxml,javafx.graphics -jar
   ➔ Generator-1.0.jar
2 pause
```

Abbildung 4.3: Startskript `Generator.bat`

Beim Start der Anwendung findet der Benutzer die grafische Oberfläche vor.

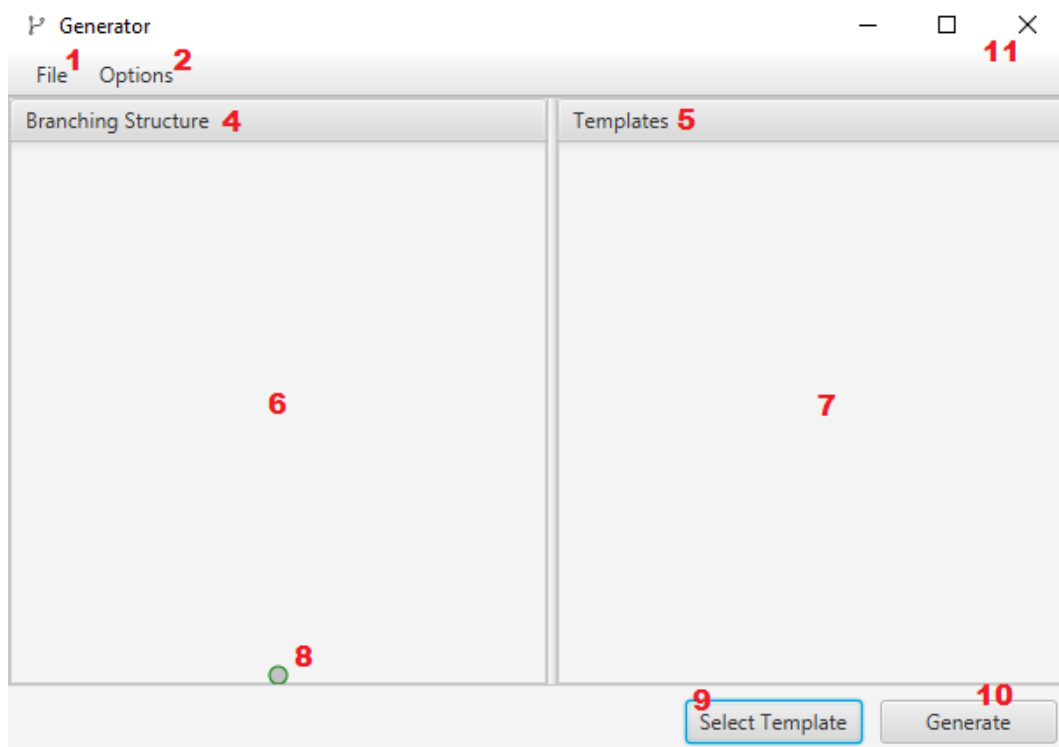
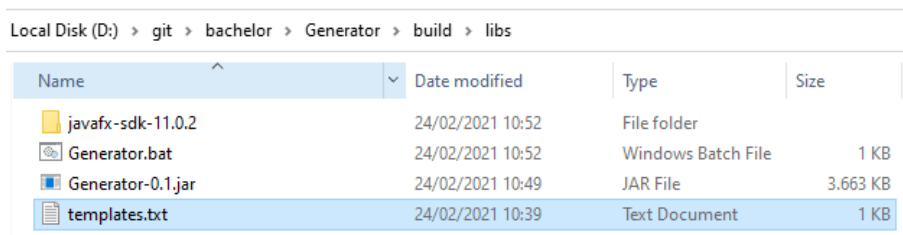


Abbildung 4.4: Programm nach Ausführung des Start-Skripts

Der Menüeintrag **1** bietet Funktionen zum Öffnen des Dateipfades und zum Laden der angelegten Templates-Datei. **2** dient zur Konfiguration der Umgebungsparameter für die Anzahl Iterationen der Ausführung und die Anzahl an Ähnlichkeitsstrukturen, die aus dem resultierenden L-System abgeleitet werden können. Die Gewichtungparameter der vorgestellten Algorithmen können ebenfalls hier eingestellt werden. Die Strukturen **4** und **5** gliedern das Programm in die Ansicht der Verzweigungsstruktur (**6**), die vom Benutzer angelegt wird, die Übersicht zur Auswahl eines Templates (**7**) und eine Sicht zur Setzung von Transformationsparametern (**7**). Der Kreis **8** stellt einen Anker dar, an welchen eine Template-Instanz angehängt werden kann. Sowohl über einen Doppelklick, also auch über einen Button (**9**) kann ein Template aus der Tempalte-Liste (**7**) ausgewählt werden. Ist die Verzweigungsstruktur vom Benutzer fertiggestellt, kann die Synthese zur Erstellung der Ähnlichkeitsabbildungen mit **10** erfolgen. Wird der `Generate`-Button ohne eine Basisstruktur gedrückt, lässt sich das Beispiel aus dieser Arbeit erzeugen (ohne Visualisierung der Basisstruktur). **11** schließt die Anwendung.

Templates

Eine Datei `template.txt` wird im selben Verzeichnis wie die ausführbare jar-Datei hinterlegt. Sie enthält je eine Template-Zeichenkette pro Zeile, welche vom Programm eingelesen und als Template zur Verfügung gestellt wird.



Name	Date modified	Type	Size
javafx-sdk-11.0.2	24/02/2021 10:52	File folder	
Generator.bat	24/02/2021 10:52	Windows Batch File	1 KB
Generator-0.1.jar	24/02/2021 10:49	JAR File	3.663 KB
templates.txt	24/02/2021 10:39	Text Document	1 KB

Abbildung 4.5: Templates-Datei zum Einlesen der Template-Strukturen

Visualisierung

Die grafische Oberfläche liegt als MVC-Pattern in Form des Application State (Model), der XML-basierten FXML-Datei (View) und dem JavaFX-Controller vor. Der Arbeitsablauf zur Erstellung der Verzweigungsstruktur kann Schritt für Schritt umgesetzt werden, nachdem die Templates geladen wurden. Dies wird an folgendem Beispiel deutlich. Die Parameter **Number of iterations**, **Number of generations**, **Rule application ratio** und **Merge application ratio** werden auf die Werte **5**, **5**, **0.5** und **0.5** gesetzt (Standardeinstellung).

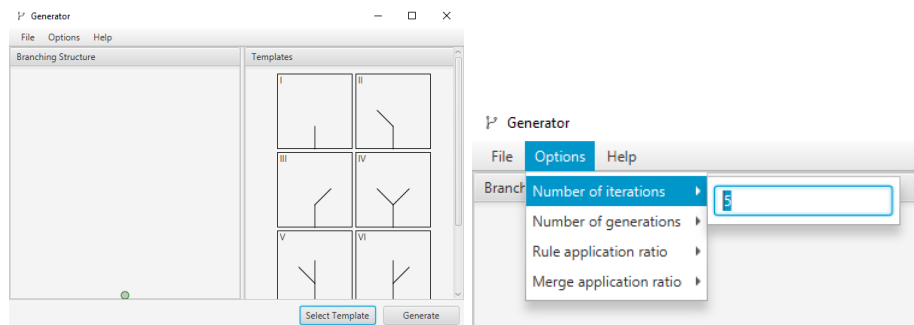


Abbildung 4.6: Erster Anker ist vorselektiert (links) & ein gesetzter Parameter (rechts)

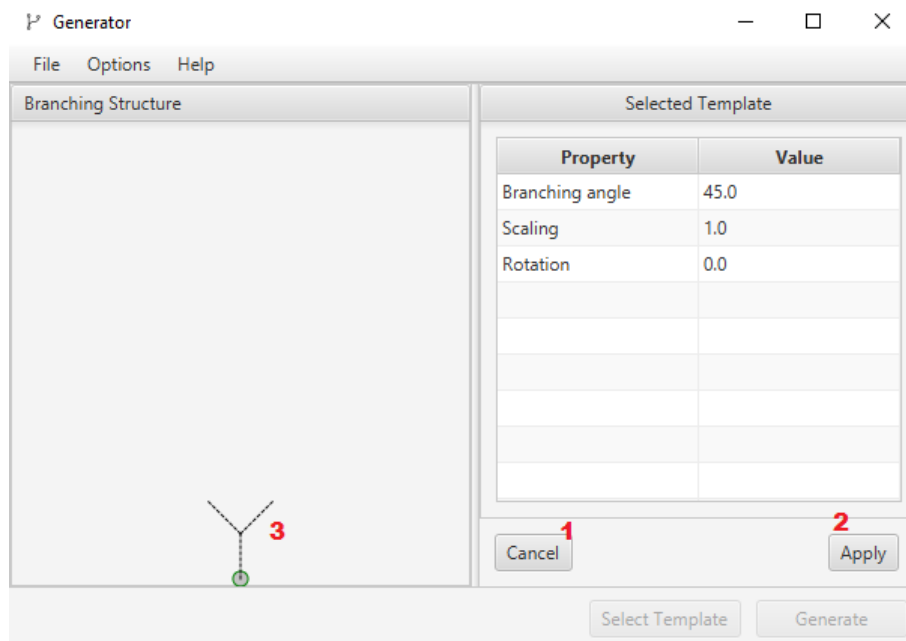


Abbildung 4.7: Auswahl des ersten Templates

Nachdem ein Template ausgewählt wird, können die Transformationsparameter gesetzt (Doppelklick) und bestätigt (**2**) oder der Vorgang abgebrochen werden (**1**). **3** zeigt einen Entwurf der Template-Instanz, die mit den aktuellen Transformationsparametern angepasst wurde. Mit der Bestätigung (**2**) wird die Template-Instanz der Basisstruktur hinzugefügt und der Benutzer gelangt wieder in die Übersicht der zur Verfügung stehenden Templates.

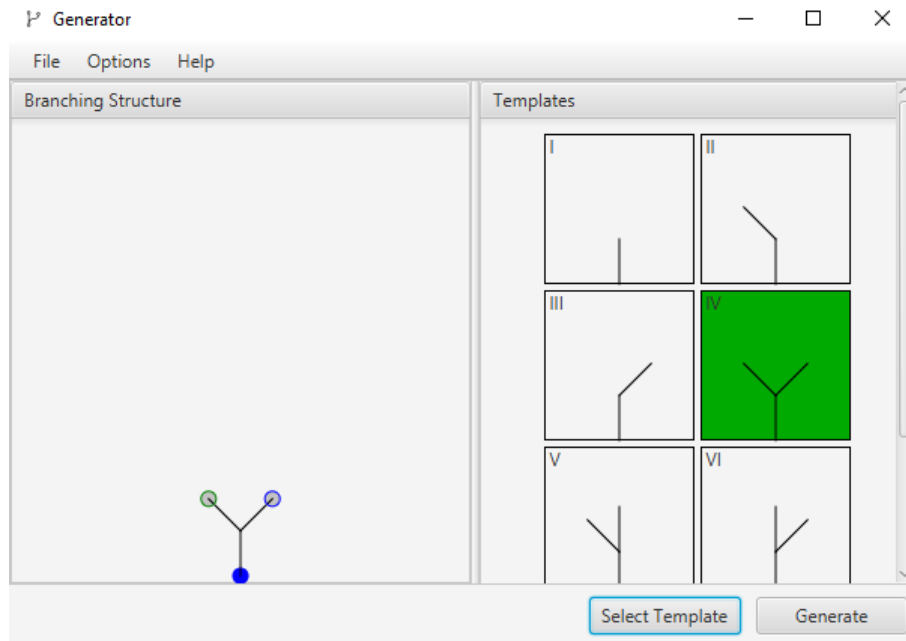


Abbildung 4.8: Der Verzweigungsstruktur hinzugefügte Template-Instanz

Bis zur gewünschten Fertigstellung kann der Benutzer den Vorgang beliebig oft wiederholen.

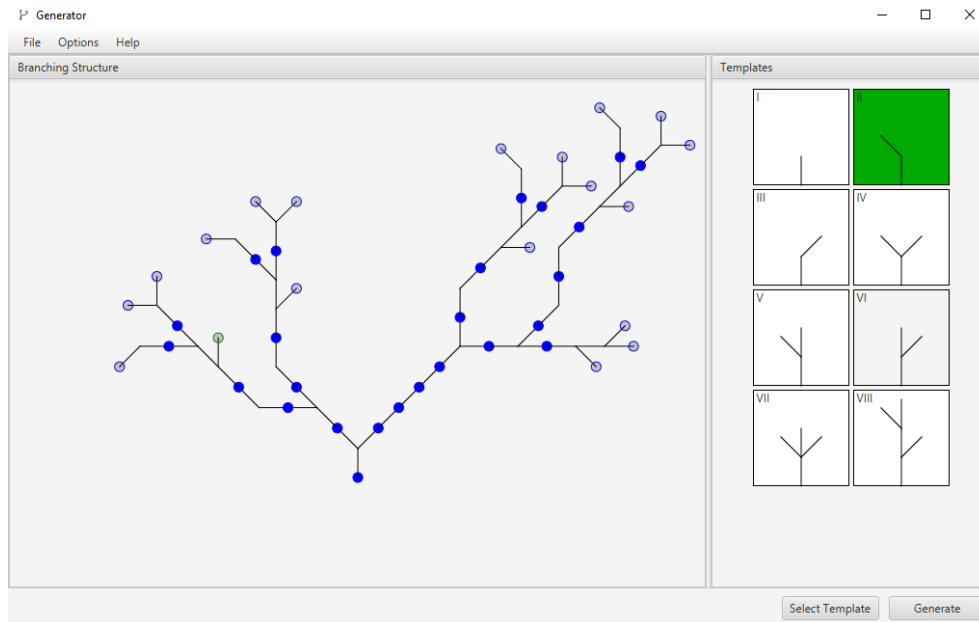


Abbildung 4.9: Vom Benutzer fertiggestellte Verzweigungsstruktur

Über den `Generate`-Button kann nun die Synthese gestartet werden. Es ergeben sich 5 (**Number of generations**) Ähnlichkeitsstrukturen (siehe A.4)

Baumtopologie

Die Klasse `TreeNode` wird verwendet, um eine iterierbare Baumstruktur aufzubauen (siehe A.1). `TreeNode` implementiert die `Iterable`-Schnittstelle, was ein Überladen der `iterator`-Funktion möglich macht. Sie stellt einen Iterator des Baumes zur Verfügung. Dieser Iterator (`TreeNodeIterator`) implementiert die `Iterator`-Schnittstelle und definiert so die Funktionen `hasNext` und `next`. Die Funktion `next` zeigt, wie eine Iteration des Baumes als Breitensuche implementiert ist. Diese wird beim Inferieren eines L-Systems aus der Baumstruktur benötigt (siehe A.2).

Das folgende Beispiel zeigt eine erstellte Baumstruktur (links) und deren erstellte Baumtopologie (rechts).

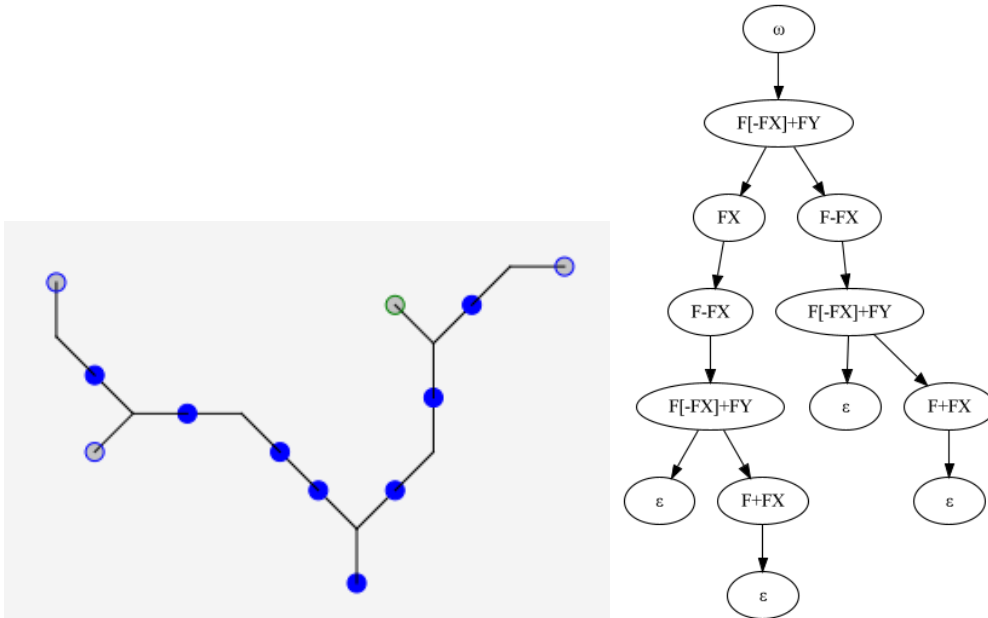


Abbildung 4.10: Verzweigungsstruktur & dazugehörige Baumstruktur

Prozesspipeline

Das verwendete Pipeline-Design-Pattern zur Umsetzung der Inferierung, Komprimierung und Generalisierung von L-Systemem, sowie die Verteilung von Transformationsparametern bei deren Ausführung, setzt sich aus der Pipeline-Klasse und dem Pipe-Interface zusammen (siehe A.3 und A.4).

Die eigentliche Umsetzung der Prozesspipeline zur Synthese der Verzweigungsstrukturen wird wie folgt implementiert. Hierbei implementieren die Klassen `InfererPipe`, `CompressorPipe`, `GeneralizerPipe` und `EstimatorPipe` das `Pipe-Interface` und geben die Reihenfolge des Prozesses an. Das `ctx`-Objekt wird als Kontextobjekt in die Pipeline gegeben und bei jedem Prozessschritt angepasst (siehe A.5).

L-System Manipulation

Die Inferierung des L-Systems findet in der Klasse `Inferer` statt, deren Objekt in der `InfererPipe`-Klasse erstellt wird. Sie nimmt die Baumstruktur entgegen und stellt eine Funktion `infer` zum Inferieren des L-Systems zur Verfügung (siehe A.6).

Das erstellte L-System (8) wird während des Inferierens aktualisiert und am Schluss als Ergebnis zurückgegeben. Dann werden die initialen Symbole F und S hinzugefügt und das Axiom auf S gesetzt (10-12). Die initiale Produktionsregel $\alpha : S \rightarrow A$ wird erstellt und der Produktionsregelmenge beigefügt (14-15). Es wird der erste zu untersuchende Knoten des Baumes über seinen Iterator zurückgegeben und als β gesetzt (17). Am Ende der Initialisierung fügt die Methode `addModuleNotPresentInAlphabet` dem Alphabet ein neues Symbol (`gamma`) hinzu, das dort nicht vorhanden ist (19). Dies geschieht in lexikografischer Reihenfolge.

Nach der Erstellung des `Inferer`-Objektes kann in der zugehörigen Pipe `InfererPipe` das L-System inferiert werden (siehe A.7).

Für die Implementierung des Inferierungsalgorithmus siehe A.8.

Die `infer`-Methode extrahiert die Zeichenkette (Wort) der im aktuellen Baumknoten gespeicherten Template-Instanz (9) und speichert sie zur Bearbeitung in `delta`. Delta wird nach Verzweigungsvariablen durchsucht (11-14) und diese dann durch ein neues Symbol, das dem Alphabet hinzugefügt wird (18), ersetzt (20). Eine neue Produktionsregel, die `gamma` auf das veränderte Wort abbildet, wird dem L-System beigefügt (24). 31-35 sucht nach Symbolen im Alphabet, die noch kein Ziel einer Produktionsregel sind und hält diese in der Variablen `gamma`. Wird ein solches Symbol nicht gefunden, terminiert die Methode und damit der Algorithmus (44). Der aktuelle Knoten wird durch den Iterator der Baumstruktur mit dem nächsten Knoten nach Breitensuche ersetzt (46). Zum Schluss wird das inferierte L-System zurückgegeben (48).

Die übrigen Prozessschritte sind ähnlich aufgebaut. Auch die Klassen `Compressor`, `Generalizer` und `Estimator` implementieren eine Funktion zum Ausführen der jeweiligen Algorithmen, während die Initialisierung im Konstruktor ausgeführt wird. Dabei wird die Implementierung der Algorithmen eng an der Konzeptionierung gehalten (siehe Kapitel 3.2). Alle weiteren Klassen der Basisalgorithmen werden in A.9 bis A.25 gezeigt. Der Algorithmus `editDistanceOptimized` aus A.21 ist aus Praktikabilitätsgründen aus [1] entnommen.

5 Evaluierung

Die Synthese von Verzweigungsstrukturen wird in den vorangegangenen Kapiteln konzeptionell betrachtet und in einem Softwareprojekt umgesetzt. Im Folgendem werden Teilaspekte an einem fortlaufenden Beispiel evaluiert und bewertet. Diese Aspekte umfassen:

- das Nutzen von Templates
- die Erstellung und Visualisierung von Verzweigungsstrukturen
- Aufbau einer Baumstruktur
- Extrahierung von Regeln und Mustern
- Komprimierung von L-Systemen
- Erweiterung von L-Systemen
- Synthese von Ähnlichkeitsstrukturen
- Auswirkung der Gewichtungparameter

Templates

Die Nutzung von Templates als Terminale einer Grammatik findet Anwendung in vielen wissenschaftlichen Arbeiten. ALIAGA ET AL. liefert hierzu eine umfassende Übersicht [3]. Mit der Verwendung einer allgemeinen Repräsentation mithilfe von *turtle*-Befehlen, wird eine Wiederverwendbarkeit der genutzten Templates sichergestellt. Die in dieser Arbeit erstellte Software nutzt ein minimalistisches System zum Einlesen der Templates aus einer textbasierten Datei. Die Forschung zur inversen prozeduralen Modellierung zeigt wiederum den Einsatz von neuronalen Netzen als vielversprechende Methode Strukturen zu erkennen und Regeln abzuleiten.

Während neuronale Netze eine Eingabestruktur analysieren und in einer baumähnlichen Struktur organisieren, knüpft diese Arbeit hier an und nutzt aus Praktikabilitätsgründen stattdessen eine benutzergeführte Anordnung von Templates zu einer Verzweigungsstruktur.

Beispiel: Die Template-Zeichenketten haben folgende Form:

```
1   FX
2   F-FX
3   F+FX
4   F [-FX] +FY
5   F [-FX] FY
6   F [FX] +FY
7   F [-FX] [FY] +FZ
8   F [+FX] F [-FY] FZ
```

Abbildung 5.1: templates.txt

F , $-$, $+$, $[$ und $]$ sind aus dem Turtle-Algorithmus bekannte Befehlssymbole. Alle anderen Symbole (z.B. X , Y , Z) stellen Verzweigungsvariablen dar, um anzuzeigen, an welcher Stelle der Templatestruktur eine neue Verzweigung abgehen kann.

Erstellung und Visualisierung

Durch die Ausführung bestimmter *turtle*-Befehle der template-basierten Struktur, wird diese visualisiert. Deshalb bietet es sich an, einfache grafische Elemente zu nutzen, um Verzweigungsstrukturen sichtbar zu machen (z.B. Canvas). Das Programm nutzt stattdessen interaktive Elemente innerhalb eines JavaFX Pane gegenüber statischen Elementen, um die Strukturen zu zeichnen. Der Prozess der Erstellung ist benutzergeführt und muss somit interagirbar sein. So können Elemente genutzt werden, mit denen der Benutzer kommunizieren kann (z.B. klickbare Kreise). Es ist nun möglich Verzweigungsstrukturen in einem simplen Arbeitsablauf zu erstellen.

Beispiel: Aus der Erstellung der Verzweigungsstruktur ergibt sich folgende Abbildung:

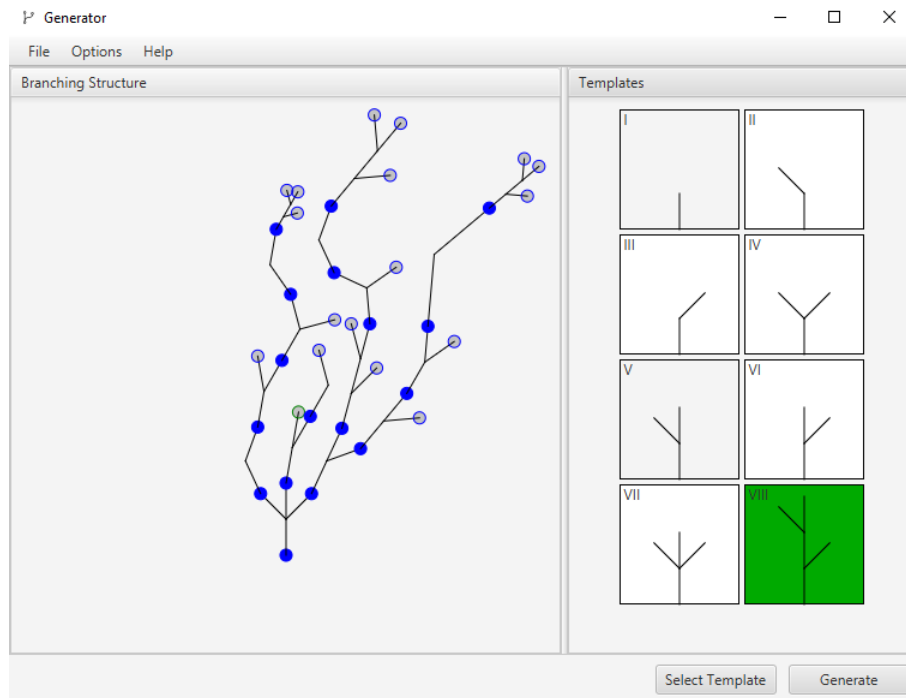


Abbildung 5.2: Grafische Benutzeroberfläche nach der Erstellung einer Basisstruktur

Aufbau einer Baumstruktur

Baumähnliche Strukturen eignen sich gut, um Transformationsparameter und topologische Anordnung von Templates datenstrukturell zu trennen.

Ein ganzheitlicher Ansatz zur Analyse von Verzweigungsstrukturen ist die Kombination aus Strukturtopologie und räumlichen Transformationen. Ein weiterer Ansatz ist die separate Betrachtung von Topologie und Transformation. Beide Ansätze sind derzeit Gegenstand der Forschung.

Das von BENES ET AL. vorgestellte System nutzt beispielsweise einen ganzheitlichen Ansatz, um sog. *Guides* zu erstellen, welche Teilsysteme der Eingabestruktur beschreiben. Hier unterscheiden sich Strukturen, die zwar identische Verzweigungen aufweisen, jedoch deren Transformationsparameter (z.B. der Winkel von Verzweigungen) voneinander abweichen. Auch in der Arbeit von STAVA ET AL. zeigt sich eine Organisation von „Clustern“ mit Transformationen, die in eine Signifikanzbewertung einfließen [22].

Bei einer separaten Betrachtung von Topologie und Transformation zeigen NISHIDA ET AL. und GUO ET AL., dass es sinnvoll ist zwei spezialisierte, neuronale Netze zu verwenden, da die räumlichen Transformationen das Erkennen der Topologie nicht signifikant beeinflussen [17, 9].

Diese Arbeit legt den Fokus auf die datenstrukturelle Trennung von Transformationen und topologischer Anordnung und erstellt somit eine Baumstruktur, die Template-Instanzen als Knoten und räumliche Transformationen als eingehende Kanten darstellt.

Beispiel: Aus der Eingabestruktur ergibt sich folgender Baum (die räumlichen Transformationen sind hier nicht visualisiert):

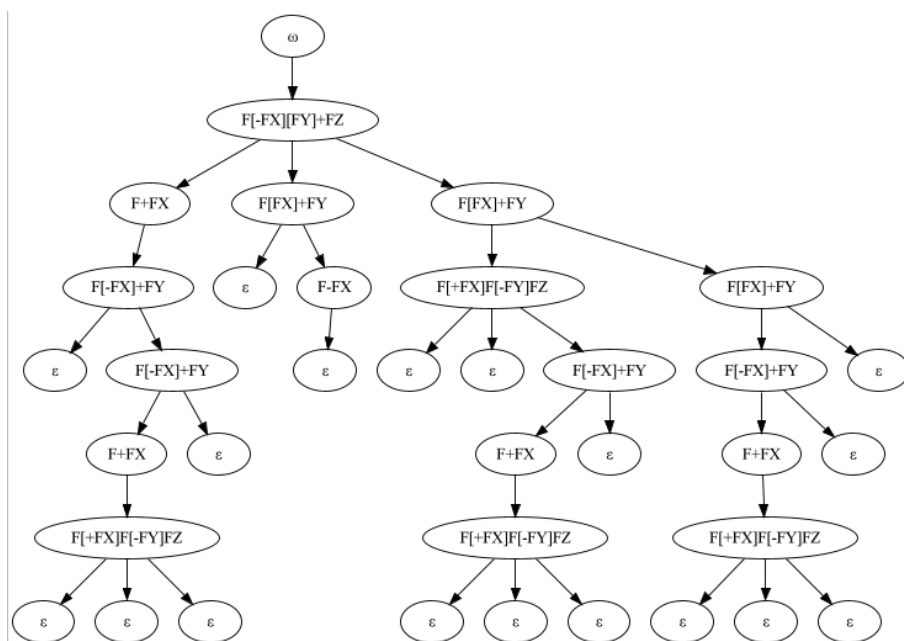


Abbildung 5.3: Baumstruktur der erstellten Verzweigungsstruktur

Extrahieren von Regeln und Mustern

Die in 3.2 vorgestellte Methodik zum Inferieren eines L-Systems aus einer Baumstruktur stellt einen Algorithmus vor, der auf die spezielle Baumstruktur zugeschnitten ist. Die L-Systeme entsprechen lediglich der Eingabestruktur und beinhalten keine Transformationen.

Beispiel: Ausgeführte Ersetzungssysteme werden über einen JavaFX Dialog visualisiert:

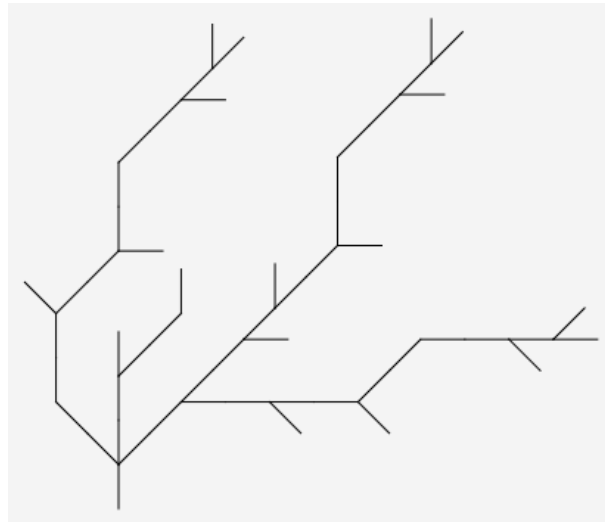


Abbildung 5.4: Inferiertes L-System

Die Zeichenkettenrepräsentation des L-Systems ($\mathcal{L} = \{M, \omega, R\}$) lautet:

```

1 LSystem{
2   [F, S, A, B, C, D, E, G, H, I, J, K, L, M, N, O, P, Q, R, T,
   ➔ U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k],
3   S,
4   [S -> A, A -> F[-FB][FC]+FD, B -> F+FE, C -> F[FG]+FH, D ->
   ➔ F[FI]+FJ, E -> F[-FK]+FL, G -> _, H -> F-FM, I -> F[+FN
   ➔ ]F[-FO]FP, J -> F[FQ]+FR, K -> _, L -> F[-FT]+FU, M ->
   ➔ _, N -> _, O -> _, P -> F[-FV]+FW, Q -> F[-FX]+FY, R ->
   ➔ _, T -> F+FZ, U -> _, V -> F+Fa, W -> _, X -> F+Fb, Y
   ➔ -> _, Z -> F[+Fc]F[-Fd]Fe, a -> F[+Ff]F[-Fg]Fh, b -> F
   ➔ [+Fi]F[-Fj]Fk, c -> _, d -> _, e -> _, f -> _, g -> _,
   ➔ h -> _, i -> _, j -> _, k -> _]
5 }

```

Das inferierte L-System zeigt, dass die Eingabestruktur richtig in eine Grammatik überführt wurde. Dabei steht der Unterstrich ($_$) für das leere Wort ε und damit für die leeren Knoten der aufgebauten Baumstruktur.

Komprimieren des L-Systems

Die Zeichenkettenrepräsentation des inferierten L-Systems zeigt einige Redundanzen auf. Zum Beispiel bilden

```
1 P -> F [-FF+FF [+F] F [-F] F] +F
```

und

```
1 Q -> F [-FF+FF [+F] F [-F] F] +F
```

das gleiche Muster. Um solche Redundanzen zu entfernen, wird das L-System in der vorgestellten Komprimierungs-Pipe reduziert. Hierbei werden identische, maximale Unterbäume und deren leere Kindknoten zusammengefasst. Der Gewichtungparameter für das Beispiel wird auf 0.5 gesetzt. Es ergibt sich folgender Baum:

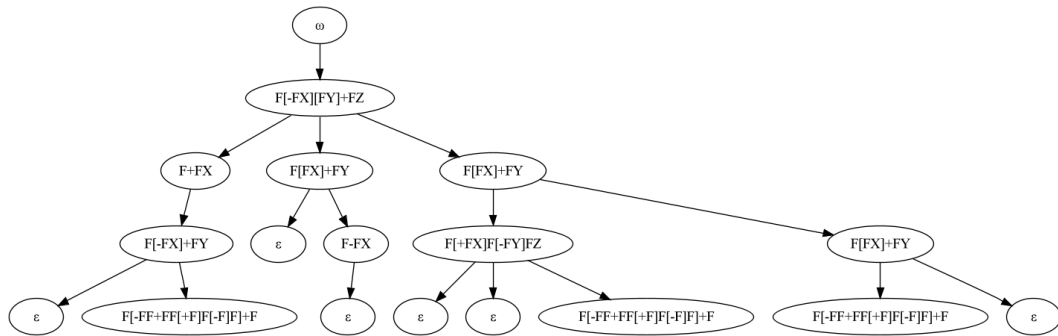


Abbildung 5.5: Komprimierter Baum

Die Baumstruktur zeigt, dass der Komprimierungsalgorithmus die maximalen Subbäume erfolgreich reduziert. Das komprimierte L-System setzt sich wie folgt zusammen:

```
1 LSystem{
2   [F, S, A, B, C, D, E, G, H, I, J, K, L, M, N, O, P],
3   S,
4   [S -> A, A -> F[-FB] [FC]+FD, B -> F+FE, C -> F[FG]+FH, D ->
   ➤ F[FI]+FJ, E -> F[-FK]+FL, G -> _, H -> F-FM, I -> F[+FN
   ➤ ]F[-FO]FP, J -> F[FL]+FG, K -> _, L -> F[-FF+FF[+F]F[-F]
   ➤ ]F]+F, M -> _, N -> _, O -> _, P -> F[-FF+FF[+F]F[-F]F
   ➤ ]+F]
5 }
```

Generalisieren des L-Systems

Der vorgestellte Generalisierungsalgorithmus verändert das L-System der Eingabestruktur, indem Produktionsregeln miteinander verbunden und mit einer Wahrscheinlichkeit versehen werden. Der Gewichtungparameter wird für das Beispiel auf 0.5 gesetzt. Es ergibt sich das finale L-System:

```

1 LSystem{
2   [F, S, E, G, K, M, N, A],
3   S,
4   [S -> A, E -> F[-FK]+FA, G -> _, K -> _, M -> _, N -> _, A
      ➡ -> F-FM, A -> F[-FA][FA]+FA, A -> F[FG]+FA, A -> F[FA]+
      ➡ FG, A -> F+FE, A -> F[+FN]F[-FA]FA, A -> F[-FF+FF[+F]F
      ➡ [-F]F]+F, A -> F[FA]+FA, A -> _]
5 }
```

Die Produktionsregelmengemenge zeigt einige Regeln auf, die dasselbe Ziel haben. Sie werden bei ihrer Anwendung zufällig ausgewählt.

Synthese

Führt man das generalisierte L-System aus, ergeben sich die Ähnlichkeitsstrukturen. Eine Evaluierung wird stichprobenartig durchgeführt, erweist sich jedoch als schwierig, da es bei der Erstellung der Ausgabestrukturen um Wahrscheinlichkeiten beim Auftreten gewisser Muster handelt. Da die Algorithmen eine akzeptable Lösung eines schwierigen Problems liefern sollen, ist die Bewertung der Ergebnisse subjektiv.

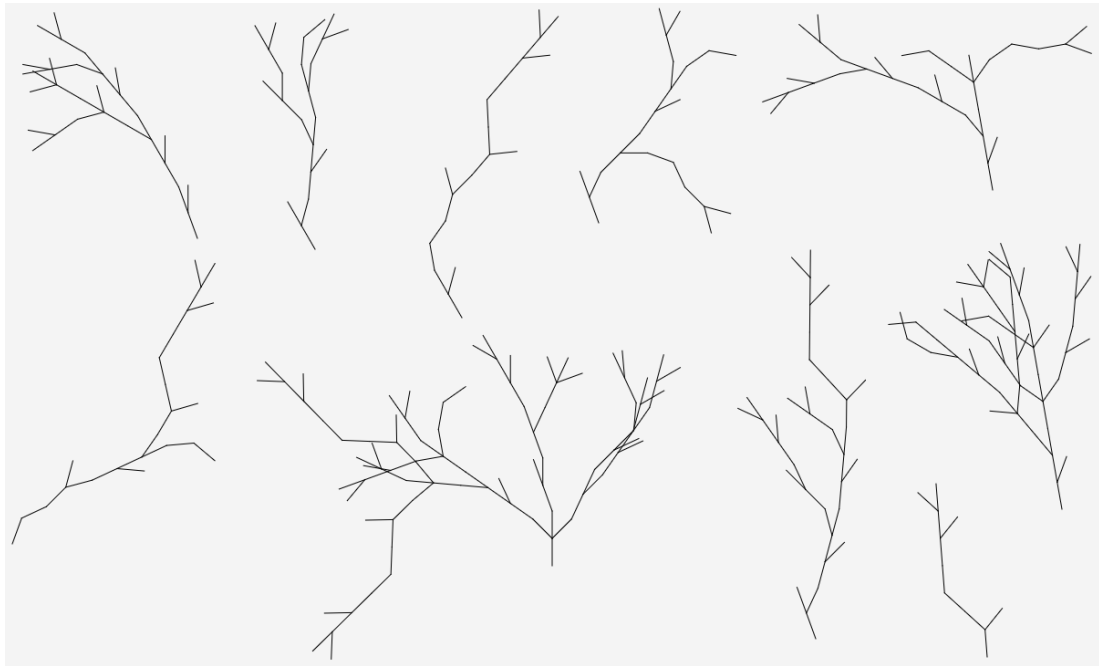


Abbildung 5.6: Synthetisierte Verzweigungsstrukturen

Vergleicht man die Eingabestruktur mit den synthetisierten Strukturen, weisen diese gewisse Eigenschaften auf (siehe folgende Abbildung):

- Wiederholtes Auftreten einer Struktur, die beim Synthetisieren als maximaler Unterbaum erkannt wurde (rot)
- Anwendung von benutzerdefinierten Transformationsparametern (blau)
- Ähnliche Häufigkeiten (grün)
- Isoliertes Auftreten von Templates des maximalen Subbaums, die über den Unterbaum hinaus einzeln vorkommen (gelb)

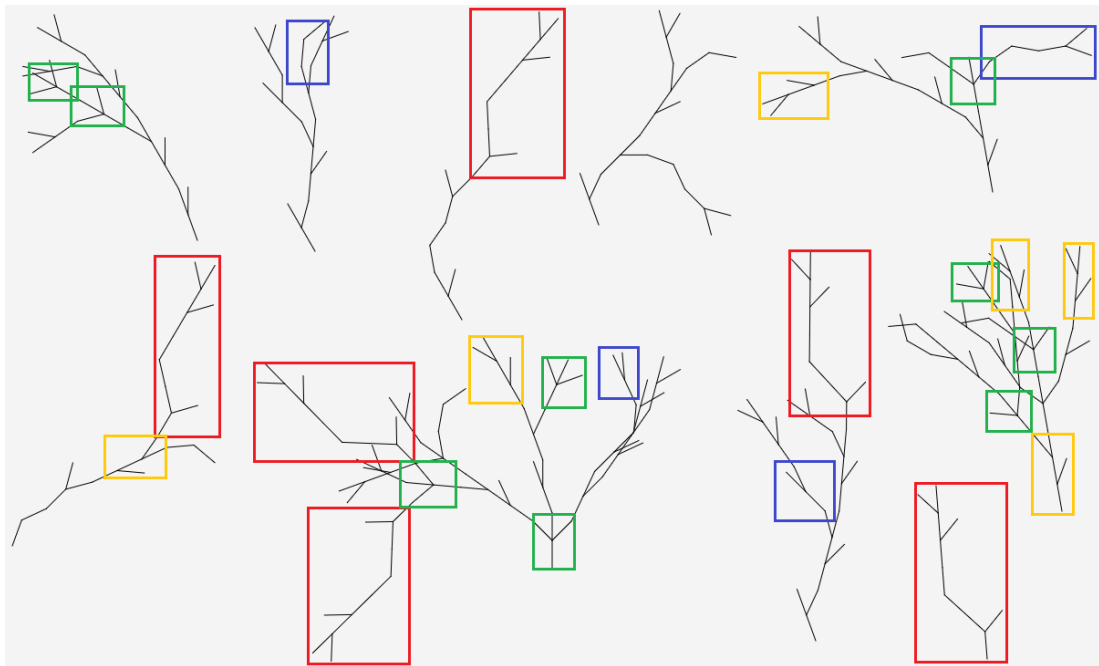


Abbildung 5.7: Untersuchung der synthetisierten Verzweigungsstrukturen

Auswirkung der Gewichtungparameter

Der Parameter w_l des vorgestellten Komprimierungsalgorithmus gewichtet eine Funktion, welche die Kosten eines L-Systems ermittelt, sodass der Algorithmus L-Systeme mit unterschiedlich großer Produktionsregelmenge anders behandelt. Hierbei bewertet ein Wert von 0.0 eine große Produktionsregelmenge höher. Entspricht w_l einem Wert von 1.0, wird eine kleinere Produktionsregelmenge überbewertet. Der Durchschnittswert 0.5, der im Beispiel verwendet wird, sorgt für eine moderate Menge an Produktionsregeln bei moderater Länge der einzelnen Regel. Der im Generalisierungsalgorithmus vorgestellte Gewichtungparameter w_0 legt fest, inwiefern die Länge oder *String Edit Distance* zweier Grammatiken von Bedeutung sind. Bei einem Wert von 0.0 überbewertet der Algorithmus die Metrik zur Berechnung der Anzahl Operationen, um eine Grammatik in eine zweite zu überführen. Der Wert 1.0 entspricht einer Überbewertung der Längenmetrik. Im Beispiel wird der Durchschnittswert 0.5 verwendet.

w_l und w_0 sind im Programm als *Rula application ratio* und *Merge application ratio* bezeichnet und können vom Benutzer vor der Generierung festgelegt werden.

6 Fazit und Ausblick

Diese Arbeit beschreibt die Konzeption und Implementierung eines prozeduralen Systems zur Erstellung von Verzweigungsstrukturen, die einer Eingabestruktur ähneln. Dabei werden Anforderungen untersucht, die zur Erstellung einer individuellen Software nötig sind. Eine intuitiv nutzbare Anwendung zur benutzergesteuerten Strukturierung von eingelesenen Templates und anschließender Generierung von Ähnlichkeitsstrukturen, ist im Rahmen dieser Arbeit entstanden.

Fazit

Diese Arbeit zeigt, wie Ansätze der aktuellen Forschung genutzt werden können, um Verzweigungsstrukturen zu erzeugen. L-Systeme eignen sich gut, um Verzweigungsstrukturen mathematisch zu beschreiben und so in einem Programm zu verwenden. Sie können durch Ausführung und Interpretation mithilfe eines Logo-Turtle-Algorithmus visualisiert und mittels grafischer Elemente sichtbar gemacht werden. Die Steuerung von Gewichtungparametern durch den Benutzer verhindert unnötige *Trial and Error*-Szenarien, da das Ergebnis des Prozesses gesteuert werden kann. Die Erstellung und Integration eines neuronalen Netzes, das laut aktueller Forschung gute Ergebnisse beim Lernen von Regeln aus einer Eingabestruktur liefert, kann in dieser Arbeit aus Praktikabilitätsgründen nicht behandelt werden.

Die umgesetzte Softwaretechnik zur Erstellung des Softwareprojekts stellt sich bedingt als praktikabel heraus. Auf der einen Seite ist ein gewisses Maß an Expertise in der Java-Spezifikation JavaFX nötig, um die Implementierung mit einem test-driven Ansatz zu beginnen. Weiter ist die Erreichung einer optimalen Testabdeckung mit Modultests nur schwer zu erreichen, da einige Module eine lauffähige JavaFX-Instanz benötigen. Auf der anderen Seite hilft sie eine angemessene Testabdeckung und so hochwertigen Quellcode zu erzeugen. Darüber hinaus ist das Softwareprojekt durch den Ansatz gut strukturiert und verringert den Aufbau technischer Schuld.

Ausblick

Die Eingabestrukturen lassen sich ohne erneute Konzeptionierung in andere Strukturen, die ebenfalls einer Baumtopologie entsprechen, überführen. L-Systeme beschreiben Baumstrukturen grundlegend und können diese durch deren Ausführung visualisieren. So können bspw. Websites, die durch das *DOM* spezifiziert sind, auf diese Weise interpretiert und erstellt werden. Hierzu muss die Software um eine Bereitstellung des erzeugten, generalisierten L-Systems erweitert werden.

Eine Weiterführung des Softwareprojekts kann durch den Einsatz neuronaler Netze zum Lernen struktureller Regeln und Ableiten von Transformationsparametern erreicht werden. Die Erstellung von Ähnlichkeitsstrukturen lässt sich durch neuronale Netze automatisieren und vereinfachen, was jedoch mit einem initialen Mehraufwand bezüglich der Implementierung verbunden ist.

Weiter lassen sich die Konzepte zur Synthese zweidimensionaler Strukturen in zukünftigen Arbeiten in den dreidimensionalen Raum überführen.

Zum Schluss ist das Erkennen von Überlappungen von Strukturen ein schwieriges Problem der Computergrafik und kann durch zusätzliche Algorithmen umgesetzt werden.

Literaturverzeichnis

- [1] AASHISH1995 (): Edit Distance | DP-5, <https://www.geeksforgeeks.org/edit-distance-dp-5/>, zugegriffen: 25.01.2021.
- [2] ALHALAWANI, S./YANG, Y./LIU, H./MITRA, N. (2013): Interactive Facades Analysis and Synthesis of Semi-Regular Facades, in: Computer Graphics Forum, 32, doi:10.1111/cgf.12041.
- [3] ALIAGA, D. G./DEMIR, I./BENES, B./WAND, M. (2016): Inverse Procedural Modeling of 3D Models for Virtual Worlds, in: ACM SIGGRAPH 2016 Courses, SIGGRAPH '16, Association for Computing Machinery, New York, NY, USA, ISBN 9781450342896, doi:10.1145/2897826.2927323, URL: <https://doi.org/10.1145/2897826.2927323>.
- [4] BENES, B./STAVA, O./MECH, R./MILLER, G. (2011): Guided Procedural Modeling, in: Computer Graphics Forum, 30, S. 325–334, doi:10.1111/j.1467-8659.2011.01886.x.
- [5] BOKELOH, M./WAND, M./SEIDEL, H.-P. (2010): A Connection between Partial Symmetry and Inverse Procedural Modeling, in: ACM Trans. Graph., 29(4), ISSN 0730-0301, doi:10.1145/1778765.1778841, URL: <https://doi.org/10.1145/1778765.1778841>.
- [6] CHARIKAR, M./LEHMAN, E./LIU, D./PANIGRAHY, R./PRABHAKARAN, M./SAHAI, A./SHELAT, A. (2005): The Smallest Grammar Problem, in: Information Theory, IEEE Transactions on, 51, S. 2554 – 2576, doi:10.1109/TIT.2005.850116.
- [7] CHOMSKY, N. (1956): Three models for the description of language, in: IRE Transactions on Information Theory, 2(3), S. 113–124, doi:10.1109/TIT.1956.1056813.
- [8] DEUSSEN, O./LINTERMANN, B. (2010): Digital Design of Nature: Computer Generated Plants and Organics, 1. Aufl., Springer Publishing Company, Incorporated, ISBN 3642073638.

- [9] GUO, J./JIANG, H./BENES, B./DEUSSEN, O./ZHANG, X./LISCHINSKI, D./HUANG, H. (2020): Inverse Procedural Modeling of Branching Structures by Inferring L-Systems, in: *ACM Trans. Graph.*, 39(5), ISSN 0730-0301, doi:10.1145/3394105, URL: <https://doi.org/10.1145/3394105>.
- [10] DE LA HIGUERA, C. (2010): *Grammatical Inference: Learning Automata and Grammars*, Cambridge University Press, USA, ISBN 0521763169.
- [11] HRUSCHKA, D. P./STARKE, D. G. (): arc42 Softwarearchitektur-Template, <https://arc42.de/template/>, pragmatisches Muster für die Erstellung, Dokumentation und Kommunikation von Software- und Systemarchitekturen.
- [12] LINDENMAYER, A. (1968): Mathematical models for cellular interactions in development. I. Filaments with one-sided inputs, in: *Journal of theoretical biology*, 18(3), S. 280—299, ISSN 0022-5193, doi:10.1016/0022-5193(68)90079-9, URL: [https://doi.org/10.1016/0022-5193\(68\)90079-9](https://doi.org/10.1016/0022-5193(68)90079-9).
- [13] MARTINOVIC, A./VAN GOOL, L. (2013): Bayesian Grammar Learning for Inverse Procedural Modeling, S. 201–208, doi:10.1109/CVPR.2013.33.
- [14] MERRELL, P./SCHKUFZA, E./LI, Z./AGRAWALA, M./KOLTUN, V. (2011): *Interactive Furniture Layout Using Interior Design Guidelines*, Association for Computing Machinery, New York, NY, USA, ISBN 9781450309431, URL: <https://doi.org/10.1145/1964921.1964982>.
- [15] MÜLLER, P./WONKA, P./HAEGLER, S./ULMER, A./VAN GOOL, L. (2006): Procedural Modeling of Buildings, in: *ACM Trans. Graph.*, 25, S. 614–623, doi:10.1145/1141911.1141931.
- [16] MULLER, M. M./TICHY, W. F. (2001): Case study: extreme programming in a university environment, in: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, S. 537–544, doi:10.1109/ICSE.2001.919128.
- [17] NISHIDA, G./GARCIA-DORADO, I./ALIAGA, D. G./BENES, B./BOUSSEAU, A. (2016): Interactive Sketching of Urban Procedural Models, in: *ACM Trans. Graph.*, 35(4), ISSN 0730-0301, doi:10.1145/2897824.2925951, URL: <https://doi.org/10.1145/2897824.2925951>.
- [18] PARISH, Y./MÜLLER, P. (2001): *Procedural Modeling of Cities*, Bd. 2001, S. 301–308, doi:10.1145/1185657.1185716.

- [19] PRUSINKIEWICZ, P. (1986): Graphical Applications of L-Systems, in: Proceedings on Graphics Interface '86/Vision Interface '86, S. 247–253, Canadian Information Processing Society, CAN.
- [20] PRUSINKIEWICZ, P./LINDENMAYER, A. (1990): The Algorithmic Beauty of Plants, Springer-Verlag, Berlin, Heidelberg, ISBN 0387972978.
- [21] SMELIK, R. M./TUTENEL, T./BIDARRA, R./BENES, B. (2014): A Survey on Procedural Modelling for Virtual Worlds, in: Comput. Graph. Forum, 33(6), S. 31–50, ISSN 0167-7055, doi:10.1111/cgf.12276, URL: <https://doi.org/10.1111/cgf.12276>.
- [22] STAVA, O./BENES, B./MECH, R./ALIAGA, D./KRISTOF, P. (2010): Inverse Procedural Modeling by Automatic Generation of L-systems, in: Computer Graphics Forum, 29, S. 1467–8659, doi:10.1111/j.1467-8659.2009.01636.x.
- [23] STAVA, O./PIRK, S./KRATT, J./CHEN, B./MECH, R./DEUSSEN, O./BENES, B. (2014): Inverse Procedural Modelling of Trees, in: Comput. Graph. Forum, 33(6), S. 118–131, ISSN 0167-7055, doi:10.1111/cgf.12282, URL: <https://doi.org/10.1111/cgf.12282>.
- [24] TALTON, J./YANG, L./KUMAR, R./LIM, M./GOODMAN, N./MECH, R. (2012): Learning design patterns with Bayesian grammar induction, in: UIST'12 - Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, doi:10.1145/2380116.2380127.
- [25] VAZQUEZ, P./SIDDI, F. (2017): About Blender, <https://www.blender.org/about/>, blender Foundation (2002).
- [26] ZHANG, S.-H./ZHANG, S.-K./LIANG, Y./HALL, P. (2019): A Survey of 3D Indoor Scene Synthesis, in: Journal of Computer Science and Technology, 34, S. 594–608, doi:10.1007/s11390-019-1929-5.

A Anhang

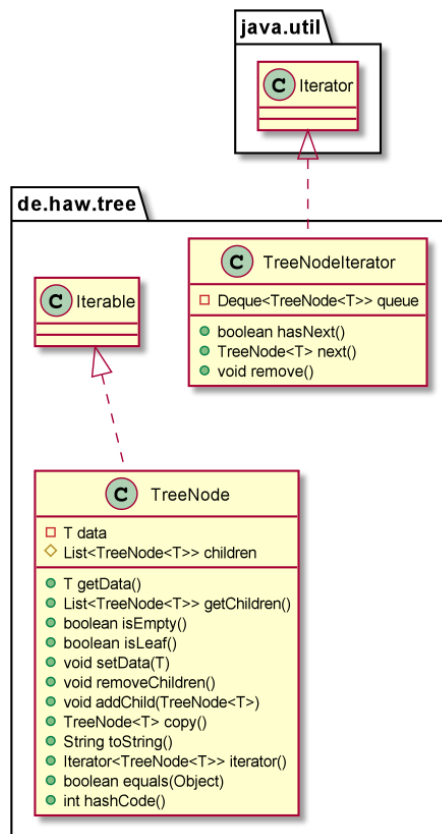


Abbildung A.1: Baumstruktur

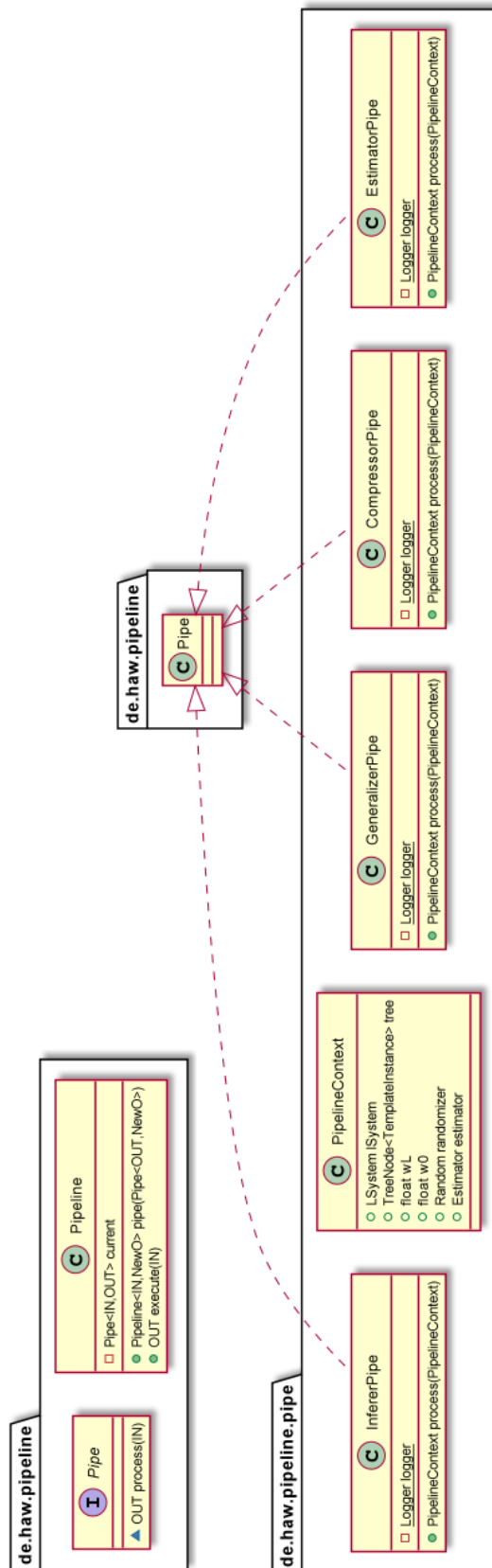


Abbildung A.2: Pipeline

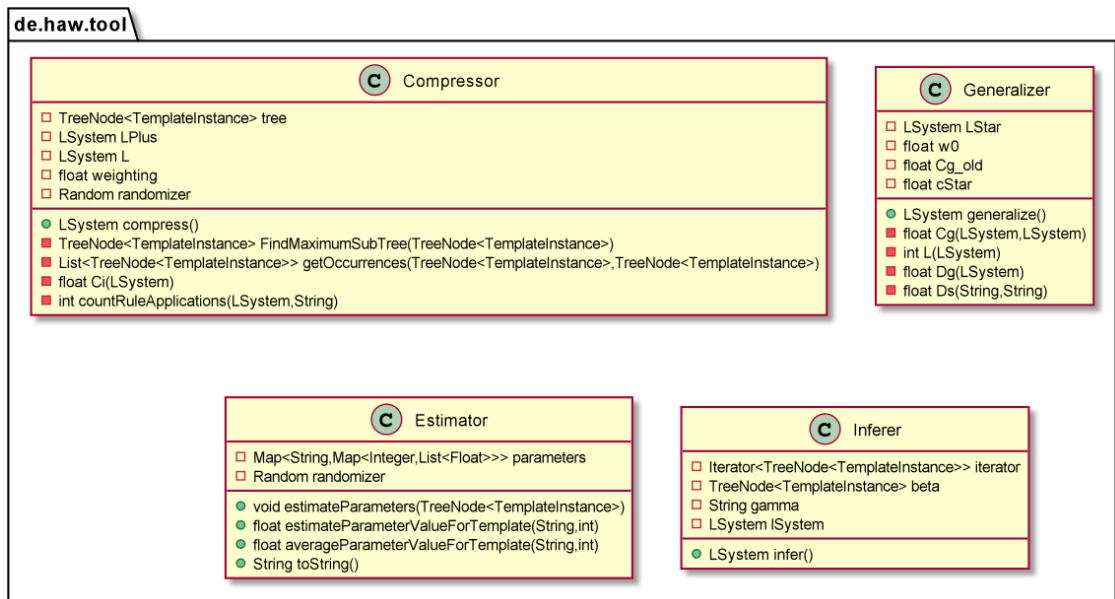


Abbildung A.3: Subsysteme der Generierungs-Pipeline

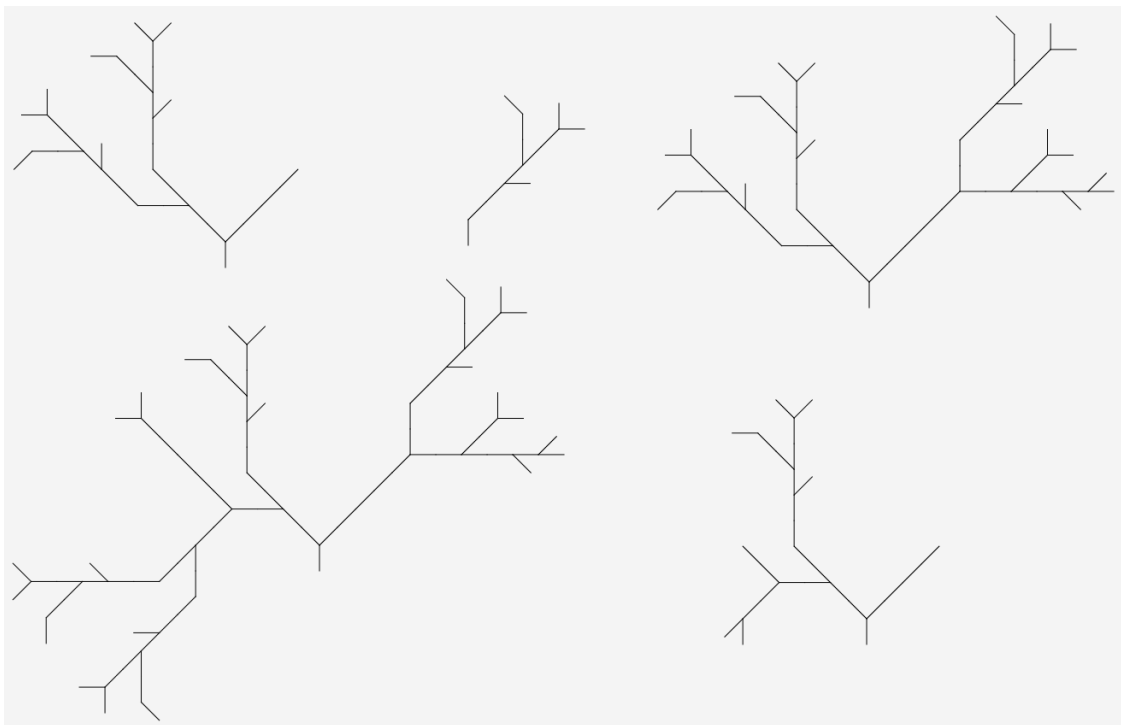


Abbildung A.4: Generierte Verzweigungsstrukturen

Listing A.1: Klasse `TreeNode` zur Erstellung einer Baumstruktur

```
/**
 * Iterable tree node structure containing a payload and children
 * as tree nodes. Every node represents a whole tree
 */
public class TreeNode<T> implements Iterable<TreeNode<T>> {
    private T data; // Payload
    protected List<TreeNode<T>> children; // Child nodes
    ...
    @Override
    public Iterator<TreeNode<T>> iterator() {
        return new TreeNodeIterator<>(this);
    }
    ...
}
```

Listing A.2: Klasse `TreeNodeIterator` als Iterator für die Baumstruktur

```
/**
 * Tree node iterator for iterating nodes of a tree
 * @param <T> Payload of the tree nodes
 */
public class TreeNodeIterator<T> implements Iterator<TreeNode<T>>{
    ...
    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    @Override
    public TreeNode<T> next() {
        if (queue.isEmpty()) return null;
        var node = queue.pop();
        queue.addAll(node.getChildren());
        return node;
    }
}
```

Listing A.3: Pipeline Klasse zur Organisation von Prozessen

```
/**
 * Pipeline class to set up a pipeline design pattern.
 * Classes implementing the Pipe interface can be executed in a specific
 * order while updating a pipeline context
 * @param <IN> Pipeline input type
 * @param <OUT> Pipeline output type
 */
public class Pipeline<IN, OUT> {
    private final Pipe<IN, OUT> current;

    public Pipeline(Pipe<IN, OUT> pipe) {
        current = pipe;
    }

    public <NewO> Pipeline<IN, NewO> pipe(Pipe<OUT, NewO> next) {
        return new Pipeline<>(input -> next.process(current.process(input)));
    }

    public OUT execute(IN input) {
        return current.process(input);
    }
}
```

Listing A.4: Pipe Interface als Vorlage zur Erstellung eines Teilprozesses einer Pipeline

```
/**
 * Pipe class as a process to be part of a pipeline
 * @param <IN> Pipe input type
 * @param <OUT> Pipe output type
 */
public interface Pipe<IN, OUT> {
    OUT process(IN input);
}
```

Listing A.5: Erstellen des Pipeline Kontextes

```
...
var ctx = new PipelineContext();
...
// Execute pipeline
var result = new Pipeline<>(new InfererPipe())
    .pipe(new CompressorPipe())
    .pipe(new GeneralizerPipe())
    .pipe(new EstimatorPipe())
    .execute(ctx);
...

```

Listing A.6: Inferer Klasse zur Inferierung eines L-Systems aus einer Baumstruktur

```
1 /**
2  * Inferer to infer a L-System out of a tree-like data structure
3  */
4 public class Inferer {
5     ...
6     public Inferer(TreeNode<TemplateInstance> tree) {
7         ///// Initializing
8         lSystem = new LSystem();
9         // M = {F, S}
10        lSystem.addModule("F", "S");
11        // w = S
12        lSystem.setAxiom("S");
13        // R ← { alpha: S → A }
14        var alpha = new ProductionRule("S", "A");
15        lSystem.addProductionRule(alpha);
16        // beta = next node
17        beta = iterator.next();
18        // M ← gamma in { A, B, ..., Z }, with gamma not in M
19        gamma = lSystem.addModuleNotPresentInAlphabet();
20    }
21    ...
22 }

```

Listing A.7: InfererPipe Klasse als Teilprozess der Pipeline

```
1  /**
2   * Pipe for executing the infer algorithm.
3   * It takes the pipeline context, set it accordingly to the result
4   * and returns it to the next pipe
5   */
6  public class InfererPipe implements Pipe<PipelineContext, PipelineContext>,
7     ↳ Logging {
8     ...
9     @Override
10    public PipelineContext process(PipelineContext input) {
11    ...
12    // Update pipeline context
13    input.lSystem = new Inferer(input.tree).infer();
14    ...
15    }
16 }
```

Listing A.8: Inferierungsalgorithmus der Inferer Klasse

```
1  /**
2   * Return a L-System inferred from the given tree structure
3   * @return Inferred L-System
4   */
5  public LSystem infer() {
6    var done = false;
7    while (!done) {
8      // delta = word of beta
9      var delta = (beta == null || beta.isEmpty()) ? "" : beta.getData().
10     ↳ getTemplate().getWord();
11     // For all variables in delta
12     var variableMatches = Pattern.compile("[A-EG-Z]")
13     .matcher(delta)
14     .results()
15     .collect(Collectors.toList());
16     for (var v : variableMatches) {
17       var index = delta.indexOf(v.group(0));
18       // Add new module not present in the alphabet
19       String zeta = lSystem.addModuleNotPresentInAlphabet();
20       // Replace variable with new module not present in alphabet
21       delta = delta.substring(0, index) + zeta + delta.substring(
22     ↳ index + 1);
23     }
24 }
```



```
23     // R ← { gamma → delta }
24     lSystem.addProductionRule(new ProductionRule(gamma, delta));
25
26     // Find an eta that is in the alphabet but not part of the LHS of
27     //   ↳ any production rule
28     var modules = lSystem.getAlphabet();
29     for (var eta : modules) {
30         if (eta.equals("F")) continue;
31         if (eta.equals("S")) continue;
32         var lhs = lSystem.getProductionRules().stream()
33             .map(ProductionRule::getLhs)
34             .filter(x -> x.equals(eta))
35             .findFirst()
36             .orElse(null);
37
38         if (lhs == null) {
39             gamma = eta;
40             break;
41         }
42
43         // There is no symbol in the alphabet not being part of a lhs
44         //   ↳ of a production rule?
45         if (modules.indexOf(eta) == modules.size() - 1) done = true;
46     }
47     beta = iterator.next();
48 }
}
```

Listing A.9: Klasse Compressor zur Komprimierung eines L-Systems

```
1  /**
2   * Compressor class for compressing a L-System.
3   * The algorithm searches for identical, maximal subtrees
4   * and replaces them with combined instances
5   */
6  public class Compressor {
7      ...
8      public Compressor(TreeNode<TemplateInstance> tree, LSystem lSystem,
9          ↪ float wL, Random randomizer) {
10         ///// Initializing
11         this.tree = tree.copy();
12         this.LPlus = lSystem;
13         // L = {}
14         this.L = new LSystem();
15         // wL in [0, 1]
16         weighting = wL;
17         this.randomizer = randomizer;
18     }
19     ...
20 }
```

Listing A.10: Komprimierungsalgorithmus der Compressor Klasse

```
1  /**
2   * Return a compressed L-System by finding maximum sub-trees
3   * and replacing them
4   * @return Compressed L-System
5   */
6  public LSystem compress() {
7      // T' ← T
8      var subtree = FindMaximumSubTree(tree);
9      while (subtree != null && !subtree.isEmpty()) {
10         // Get extended string representation from (repetitive) sub tree
11         var subTreeDerivation = new Inferer(subtree).infer().derive();
12         // Data to be set in the tree to replace old node structure
13         ↪ representing the subtree
14         Template template = new Template(subTreeDerivation);
15         // Estimate parameters for new template instance derivation
16         var estimator = new Estimator(randomizer);
17         var occurrences = getOccurrences(subtree, tree);
18         // Average parameter
19         for (var o : occurrences) {
20             estimator.estimateParameters(o);
21         }
22     }
23 }
```

```
20     }
21     // Replace occurrences of the sub tree
22     for (var o : occurrences) {
23         var derivationInstance = new TemplateInstance(template);
24
25         int scalingSum = 0, rotationSum = 0, branchingAngleSum = 0;
26         int counter = 0;
27         for (var node : o) {
28             if (node.isEmpty()) continue;
29             counter++;
30             scalingSum += estimator.averageParameterValueForTemplate("
                ↳ Scaling", node.getData().getTemplate().getId());
31             rotationSum += estimator.averageParameterValueForTemplate("
                ↳ Rotation", node.getData().getTemplate().getId());
32             branchingAngleSum += estimator
                ↳ averageParameterValueForTemplate("Branching_angle",
                ↳ node.getData().getTemplate().getId());
33         }
34
35         derivationInstance.setParameter("Scaling", (float) (scalingSum
                ↳ / counter));
36         derivationInstance.setParameter("Rotation", (float) (
                ↳ rotationSum / counter));
37         derivationInstance.setParameter("Branching_angle", (float) (
                ↳ branchingAngleSum / counter));
38
39         o.setData(derivationInstance);
40         o.removeChildren();
41     }
42     L = new Inferer(tree).infer().minimize();
43     if (Ci(L) >= Ci(LPlus)) {
44         break;
45     }
46     // T ← T'
47     subtree = FindMaximumSubTree(tree);
48     // L+ ← L
49     LPlus = L;
50 }
51
52 return LPlus.clean();
53 }
```

Listing A.11: Algorithmus zum Finden eines maximalen Unterbaums

```
1  /**
2   * Search and for maximum sub-tree that appears more than one time
3   * in the tree and return it
4   * @param tree Tree to be searched
5   * @return Maximum sub-tree
6   */
7  private TreeNode<TemplateInstance> FindMaximumSubTree(TreeNode<
      ↳ TemplateInstance> tree) {
8      var globalIterator = tree.iterator();
9      globalIterator.next();
10     // Iterate
11     while (globalIterator.hasNext()) {
12         var globalNode = globalIterator.next();
13         var localIterator = tree.iterator();
14         var l = localIterator.next();
15         while (l != globalNode) l = localIterator.next();
16         while (localIterator.hasNext()) {
17             var localNode = localIterator.next();
18             if (!globalNode.isLeaf()) {
19                 // Check for equality / check for appearance > 1 in the tree
20                 if (Trees.compare(globalNode, localNode)) return globalNode;
21             }
22         }
23     }
24     return null;
25 }
```

Listing A.12: Algorithmus zum Finden aller Vorkommen eines Unterbaums

```
1  /**
2  * Find all occurrences of a sub-tree in a tree
3  * @param subtree Sub-tree to be searched for
4  * @param tree Tree that contains the sub-tree
5  * @return List of sub-trees
6  */
7  private List<TreeNode<TemplateInstance>> getOccurrences(TreeNode<
    ↳ TemplateInstance> subtree, TreeNode<TemplateInstance> tree) {
8      var iterator = tree.iterator();
9      iterator.next();
10     // Store occurrences
11     var occurrences = new ArrayList<TreeNode<TemplateInstance>>();
12     // Iterate through the tree
13     while (iterator.hasNext()) {
14         var node = iterator.next();
15         if (Trees.compare(node, subtree)) {
16             // Subtree to be replaced found
17             occurrences.add(node);
18         }
19     }
20     return occurrences;
21 }
```

Listing A.13: Funktion zur Berechnung der Kosten eines L-Systems

```
1  /**
2  * Return the costs of a L-System combining the length of the alphabet
3  * and the number of rule applications of the production rules
4  * @param lSystem L-System to be measured
5  * @return Costs of the L-System
6  */
7  private float Ci(LSystem lSystem) {
8      var costs = 0;
9      for (var rule : lSystem.getProductionRules()) {
10         costs += weighting * rule.getRhs().length() + (1 - weighting) *
            ↳ countRuleApplications(lSystem, rule.getLhs());
11     }
12     return costs;
13 }
```

Listing A.14: Funktion zur Ermittlung der Anzahl Anwendungen einer Produktionsregel

```
1 /**
2  * Counts the production rule applications in a string and returns it
3  * @param lSystem L-System to be searched
4  * @param lhs LHS of a production rule
5  * @return Number of production rule applications
6  */
7 private int countRuleApplications(LSystem lSystem, String lhs) {
8     int occurrences = 0;
9     var pattern = Pattern.compile(lhs);
10    // Check axiom
11    var axiomMatcher = pattern.matcher(lSystem.getAxiom());
12    while (axiomMatcher.find()) occurrences++;
13    // Check production rules
14    var ruleMatcher = pattern.matcher(lSystem.getProductionRules().stream()
15        .map(ProductionRule::getRhs).collect(Collectors.joining()));
16    while (ruleMatcher.find()) occurrences++;
17    return occurrences;
18 }
```

Listing A.15: Generalizer Klasse zur Generalisierung eines L-Systems

```
1 /**
2  * Generalizer class to add non-deterministic rules to an L-System
3  */
4 public class Generalizer {
5     ...
6     public Generalizer(LSystem lSystem, float w0) {
7         //// Initialization
8         // Generalized grammar L* = L+ (compact grammar)
9         LStar = lSystem.copy();
10        // metric weight balancing
11        this.w0 = w0;
12        // C^old_g = Cg( L + { p }, L )
13        Cg_old = 0;
14        // cStar
15        cStar = 0;
16    }
17    ...
18 }
```

Listing A.16: Generalisierungsalgorithmus der Generalizer Klasse

```
1  /**
2   * Generalize a L-System by adding non-deterministic rules
3   * @return Generalized L-System
4   */
5  public LSystem generalize() {
6      do {
7          // Exclude S → A from set combinations
8          var productionRulesWithoutS = new ArrayList<>(LStar.
9              ▶ getProductionRules());
10         if (productionRulesWithoutS.remove(0) == null)
11             throw new RuntimeException("No_rule_S_→_A_found");
12         // Generate all possible merging rule pairs P → L*
13         var combinations = Sets.combinations(new HashSet<>(
14             ▶ productionRulesWithoutS), 2);
15         var minimalCosts = Float.MAX_VALUE;
16         LSystem minimalLSystem = null;
17         // Find a pair p_i with the minimal Cg(L* + {p_i}, L*)
18         for (var c : combinations) {
19             var LStarMerged = LStar.merge(c);
20             float costs = Cg(LStarMerged, LStar);
21             if (costs < minimalCosts) {
22                 minimalCosts = costs;
23                 minimalLSystem = LStarMerged;
24             }
25         }
26         if (minimalCosts >= 0) break;
27         cStar = minimalCosts - Cg_old;
28         Cg_old = minimalCosts;
29         LStar = minimalLSystem;
30     } while (cStar <= 0);
31     return LStar;
32 }
```

Listing A.17: Kostenfunktion zum Vergleich zweier L-Systeme

```
1 /**
2  * Calculates the costs of editing a grammar leveraging the grammar
3  * length and the grammar edit distance both weighted with a weight
4  * balancing w0      [0.0, 1.0]
5  */
6 private float Cg(LSystem lStar, LSystem lPlus) {
7     return w0 * (L(lStar) - L(lPlus)) + (1 - w0) * Dg(lStar);
8 }
```

Listing A.18: Längenfunktion über ein L-System

```
1 /**
2  * Measure and return the length of a L-System
3  * @param lSystem L-System to be measured
4  * @return L-System length
5  */
6 private int L(LSystem lSystem) {
7     // Set size of the alphabet M
8     var M_size = lSystem.getAlphabet().size();
9     // Sum of RHS symbols over all production rules
10    var sum_rulesRHSs = lSystem.getProductionRules().stream()
11        .map(r -> r.getRhs().length())
12        .mapToInt(Integer::intValue)
13        .sum();
14    return M_size + sum_rulesRHSs;
15 }
```

Listing A.19: Grammar Edit Distance

```
1 /**
2  * Grammar edit distance: Overall costs to convert a grammar to another
3  * by a set of merging operations  $M(L+ \rightarrow L^*)$ 
4  */
5 private float Dg(LSystem lStar) {
6     // Grammar edit distance
7     var editDistance = 0;
8     // Get all multi-module production rules
9     var rules = lStar.getProductionRules().stream()
10        .filter(rule -> rule.getLhs().length() > 1)
11        .collect(Collectors.toList());
12
13    // Go through all merging operations (merging two rules)
14    while (!rules.isEmpty()) {
```



```
15     var operation1 = rules.get(0);
16     // Find corresponding rules
17     var otherOperations = rules.stream()
18         .filter(r -> !r.equals(operation1) && r.getLhs().equals(
19             ↳ operation1.getLhs()))
20         .collect(Collectors.toList());
21
22     rules.remove(operation1);
23
24     if (!otherOperations.isEmpty()) {
25         var iterator = otherOperations.iterator();
26         do {
27             // Ds(M*_A,M*_B)
28             var operation = iterator.next();
29             editDistance += Ds(operation1.getRhs(), operation.getRhs());
30             rules.remove(operation);
31         } while (iterator.hasNext());
32     }
33
34     return editDistance;
35 }
```

Listing A.20: String Edit Distance

```
1 /**
2  * Calculate and return the edit distance between two strings
3  * @param M_A String A
4  * @param M_B String B
5  * @return Edit distance between A and B
6  */
7 private float Ds(String M_A, String M_B) {
8     return Modules.editDistanceOptimized(M_A, M_B);
9 }
```

Listing A.21: Modules Klasse mit Funktion zur Ermittlung der String Edit Distance

```
1 public class Modules {
2     ...
3     /**
4      * Calculate and return the edit distance between two strings
5      * @param str1 String A
6      * @param str2 String B
7      * @return Edit distance between A and B
8      */
9     public static int editDistanceOptimized(String str1, String str2) {
10         int len1 = str1.length();
11         int len2 = str2.length();
12         int [][] DP = new int [2][len1 + 1];
13         for (int i = 0; i <= len1; i++) DP[0][i] = i;
14         for (int i = 1; i <= len2; i++) {
15             for (int j = 0; j <= len1; j++) {
16                 if (j == 0) DP[i % 2][j] = i;
17                 else if (str1.charAt(j - 1) == str2.charAt(i - 1)) DP[i %
18                     2][j] = DP[(i - 1) % 2][j - 1];
19                 else DP[i % 2][j] = 1 + Math.min(DP[(i - 1) % 2][j], Math.
20                     min(DP[i % 2][j - 1], DP[(i - 1) % 2][j - 1]));
21             }
22         }
23     }
24 }
```

Listing A.22: Estimator Klasse zur Erstellung einer Verteilung über Transformationsparameter

```
1 /**
2  * Estimator class to create a distribution of different
3  * transformation parameters
4  */
5 public class Estimator {
6     ...
7     public Estimator(Random randomizer) {
8         /// Initialization
9         parameters = new HashMap<>();
10        this.randomizer = randomizer;
11    }
12    ...
13 }
```

Listing A.23: Verteilungsalgorithmus der Estimator Klasse

```

1  ...
2  /**
3   * Create transformation parameter distribution of a given tree
4   * @param tree Tree the parameters are distributed from
5   */
6  public void estimateParameters(TreeNode<TemplateInstance> tree) {
7      // Determine different parameters
8      tree.getData().getParameters().forEach((key, value) -> parameters.
9          ▶ putIfAbsent(key, new HashMap<>()));
10     // Iterate tree and extract parameters
11     for (var node : tree) {
12         if (node == null || node.isEmpty()) continue;
13         var templateID = node.getData().getTemplate().getId();
14         for (var p : node.getData().getParameters().entrySet()) {
15             var parameterEntry = parameters.get(p.getKey());
16             var templateEntry = parameterEntry.get(templateID);
17             if (templateEntry == null) {
18                 var valueList = new ArrayList<Float>();
19                 valueList.add(p.getValue().floatValue());
20                 parameterEntry.put(templateID, valueList);
21             } else {
22                 templateEntry.add(p.getValue().floatValue());
23             }
24         }
25     }
26     ...

```

Listing A.24: Abrufen eines zufälligen Parameters aus der Verteilung für ein Template

```

1  ...
2  /**
3   * Estimate and return a parameter for a given template by name
4   * @param parameter Parameter name
5   * @param templateID Corresponding template
6   * @return Estimated parameter value
7   */
8  public float estimateParameterValueForTemplate(String parameter, int
9      ▶ templateID) {
10     var entries = parameters.get(parameter).get(templateID);
11     return entries.get(randomizer.nextInt(entries.size()));
12 }
13 ...

```

Listing A.25: Berechnung des Durchschnitts eines Parameters für ein Template

```
1 ...
2 /**
3  * Calculate and return the average value for a parameter for a
4  * given template by name
5  * @param parameter Parameter name
6  * @param templateID Corresponding template
7  * @return Averaged parameter value
8  */
9 public float averageParameterValueForTemplate(String parameter, int
    ↳ templateID) {
10     var entries = parameters.get(parameter).get(templateID);
11     return (float) entries.stream().mapToDouble(v -> v).average().orElse(0);
12 }
13 ...
```

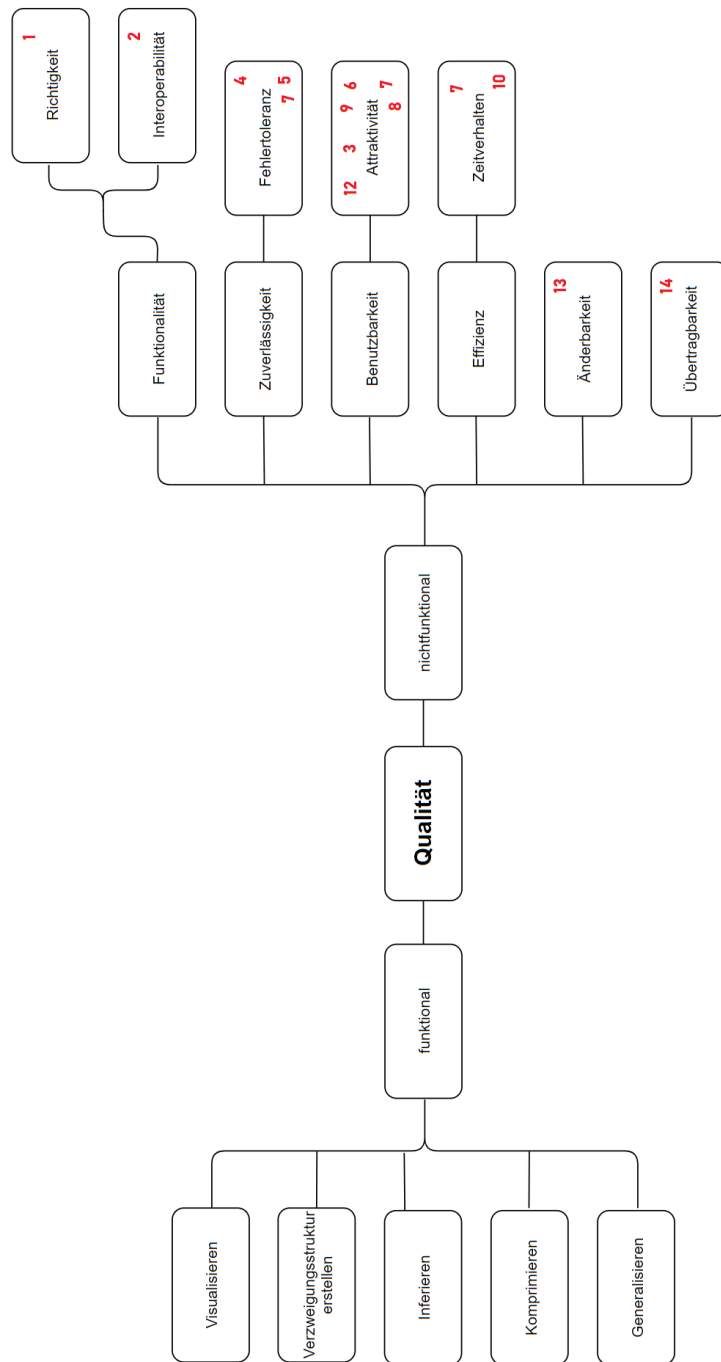


Abbildung A.5: Qualitätsbaum

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Template-basierte Synthese von Verzweigungsstrukturen mittels L-Systemen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original
-----	-------	--------------------------