Fachhochschule Nordwestschweiz
Hochschule für Technik

Algoria – Allegorien für Algorithmen
Skizzenerkennung für Tablet-PCs

MSE Projekt P7a – Symbolerkennung

September 2010

Autor: Raphael Schweizer

Advisor: Prof. Dr. Christoph Stamm
Institut für Mobile und Verteilte Systeme FHNW

Raphael Schweizer
Hinterer Hafen 348
5224 Unterbözberg

+41 (0)79 378 42 33
fhnw@schweizer-informatik.ch

Im Projekt Algoria wird eine Anwendung zur Skizzenerkennung auf Tablet-PCs entwickelt. Es sollen typische Datenstrukturen aus der Informatik - wie Arrays, Listen, Bäume und Graphen - erkannt werden. Die Skizzen sollen anschliessend „zum Leben erweckt“ werden, so dass häufige Operationen (z.B. Einfügen, Löschen, Suchen, Sortieren, etc.) direkt auf den erkannten Strukturen animiert ausgeführt werden können.

Diese Arbeit beschreibt, wie die Symbolerkennung gelöst wurde. Dafür wird einleitend eine Übersicht der einzelnen Software-Bestandteile und ihrer Aufgaben gegeben. Es folgt eine Analyse der Probleme sowie eine Beschreibung möglicher Lösungsansätze. Schliesslich wird die gewählte Lösung detailliert dargestellt und ein Ausblick auf die weitere Arbeit gegeben.

Das Projekt Algoria ist *Work in Progress* in experimentellem Stadium. Die vorliegende Arbeit entspricht mindestens der Revision 1300 des Codes auf dem zur Entwicklung von Algoria verwendeten Versionierungsserver.

Inhaltsverzeichnis

Akronyme

AST Abstract Syntax Tree

BBN Bayesian Belief Network

DSL Domain Specific Language

GUI Graphical User Interface

MEF Managed Extensibility Framework

PDA Personal Digital Assistant

WPF Windows Presentation Foundation

XAML Extensible Application Markup Language

1. Einleitung

1.1. Ziel

Algoria soll Dozierenden ermöglichen, die Klasse während des gesamten Unterrichts anzuschauen. Bisher wenden sie sich oft der Tafel zu, um Datenstrukturen zu zeichnen. Gerade beim Erklären von Algorithmen verbringen sie viel Zeit mit dem „Kopieren“ von Skizzen. Für Studierende soll der Unterricht spannender werden, sie sollen Algorithmen und Datenstrukturen „in Aktion“ erleben.

1.2. Übersicht

Der Unterricht an Schulen, insbesondere den höheren Stufen, ist in einem ständigen Wandel. Die Wandtafel ist weitgehend von Hellraumprojektoren[1] und Beamern abgelöst worden. Die Studenten benutzen Laptops anstelle von Notizblöcken. Seit Kurzem wird der Laptop selbst vermehrt von Tablet-PCs verdrängt. Geeignete Betriebssysteme[2] ebneten den Weg für alternative Eingabemethoden wie berührungssensitive Bildschirme, Stifteingabe und Handschrift. Die Entwicklung bei Mobiltelefonen, Smartphones und Personal Digital Assistants (PDAs) hat zu einem regelrechten Boom in diesem Bereich geführt.[3] Trotzdem sind Anwendungen, die diese neuen, intuitiven Techniken nutzen, noch Mangelware. Neben *Microsoft Journal* und *Microsoft OneNote* - beides Notizbücher, letzteres aber ungleich mächtiger - existiert kaum Produktivsoftware, die über reine Spielerei hinaus geht.[4]

In diese Lücke stösst Algoria als konsequente Weiterentwicklung von Wandtafel und Prokischreiber. Algoria erlaubt Dozierenden während dem Zeichnen von Datenstrukturen der Klasse zugewandt zu bleiben und sowohl Korrekturen und Annotationen anzubringen, als auch komplizierte Sachverhalte mittels Animationen und Hervorhebungen verständlich darzustellen.

Um die möglichen Einsatzszenarien nicht im Vornherein zu begrenzen, wird Algoria offen und generisch entworfen. Konsequente Trennung von Oberfläche (Graphical User Interface (GUI)) und Applikationslogik, in einer Domain Specific Language (DSL) definierbare Symbole und Plugins für Datenstrukturen und Algorithmen unterstützen diese Anforderung. Die einfachste und schnellste Möglichkeit, eine Windows-Anwendung mit Unterstützung neuartiger Eingabegeräte und ansprechender Oberfläche zu entwickeln, bietet das

[1] auch Tageslichtprojektor, Prokischreiber

[2] Im November 2002 brachte Microsoft *Windows XP Tablet PC Edition* auf den Markt, das zuerst nur vorinstalliert auf einem passenden Computer erhältlich war. Mit dem Service Pack 2 für Windows XP im August 2004 wurde *Windows XP Tablet PC Edition 2005* Bestandteil aller aktuellen Windows Versionen. Mit Ausnahme der abgespeckten Starter und Home Basic Varianten von Windows Vista bzw. Windows 7 sind die Nachfolger der Tablet-Komponenten heute serienmässige Austattung. Linux unterstützt ebenfalls Stifteingabe und berührungssensitive Displays, Projekte zur Handschrifterkennung sind aber rar und können sich nicht mit Microsofts Lösung messen. Apple spezialisiert sich auf die Eingabe mittels Finger und bietet keine stiftbasierten Lösungen.

[3] 2003 machten Tablet-PCs noch nicht einmal ein Prozent der Notebook-Verkäufe aus[Cha03]. Aktuelle Zahlen sind schwer erhältlich, mittlerweile führen aber die meisten grossen Hersteller mindestens ein Tablet-PC-Modell im Sortiment. Asus’ eee PCs und Apples iPad versuchen Tablet-PCs der breiten Masse zugänglich zu machen. Somit darf angenommen werden, dass die Hersteller mit steigendem Absatz rechnen.

[4] Eine Ausnahme bilden Kassen- und andere Industrieanwendungen, die schon seit geraumer Zeit meist auf Embedded-Systemen mit Touchscreen eingesetzt werden.

.NET Framework.[5] Für die Oberfläche wurde Windows Presentation Foundation (WPF) gewählt, welche durch Extensible Application Markup Language (XAML) eine deklarative Beschreibung von GUIs erlaubt.

Im Folgenden liegt der Schwerpunkt auf der Symbolerkennung und somit auf den zwei Haupt-Aufgaben:

1. Umwandeln von Stiftbewegungen (Aufsetzen, Ziehen, Absetzen) in geometrische Primitive (Punkt, Strecke, Pfad, Polygon, Ellipse, Kreis, Kreisbogen)
2. Zusammensetzen dieser Primitive zu bestimmten Symbolen

1.3. Dank

Mein Dank gilt Prof. Dr. Christoph Stamm, meinem Advisor, für seine unerschöpfliche Geduld; Herrn Beat Walti, meinem Algoria-Arbeitskollegen, für unzählige Diskussionen; Herrn Prof. Dr. Dominik Gruntz, meinem MSE-Verantwortlichen, für den administrativen Einsatz und nicht zuletzt Frau Caroline Stöckli, meiner langjährigen Freundin, für das verständnisvolle Ertragen meiner Laune während den Tagen vor der Abgabe.

[5] Ab Version 3.5, verfügbar ab Windows XP

2. Symbolerkennung

Die Symbolerkennung baut auf den Erkenntnissen von T.A. Hammond und R. Davis auf. Sie stellen fest, dass die besten Resultate von mehrphasigen Systemen erreicht werden. [HD03, Ham07] Diese Systeme erkennen zuerst kontextfrei geometrische Primitive und setzen diese anschliessend zu komplexeren Symbolen zusammen. So wird eine weitgehend benutzerunabhängige und trainingsfreie Erkennung von Symbolen erreicht. Dabei können die Symbole in einer Beschreibungssprache spezifiziert werden, die nahe der natürlichen Sprache ist, d.h. eine untrainierte Versuchsperson beschreibt Symbole ähnlich wie sie für solche Systeme spezifiziert werden müssen. Das .NET Framework stellt einen Grossteil der für die im Folgenden beschriebenen ersten Phase benötigten Funktionalität zur Verfügung.

2.1. Primitive

Die Symbolerkennung beginnt mit dem Zeichnen eine Linie (`Stroke`). Diese wird durch den Digitizer beim Antippen bzw. Ziehen mit dem Stift oder mit der Maus erzeugt. Die `Strokes` werden analysiert und in Punkte, Linien(züge), n-Ecke, Kreis(segmente) und Pfade eingeteilt. Diese Geometrien sind atomare Symbole und bilden die Grundlage der anschliessenden Symbolsynthese.

2.2. Symbolsynthese

Sollen einzelne Symbole aus geometrischen Primitiven zusammengesetzt werden, so stellt sich zunächst die Frage nach einer geeigneten Beschreibungssprache für Symbole.

2.2.1. Symbolbeschreibung

Für Algoria wird eine einfache, an LADDER [Ham07, HD03] angelehnte, Sprache verwendet. Der Einfachheit halber wird diese im Folgenden ebenfalls LADDER genannt. Mit ihr ist es möglich, Symbole ähnlich der natürlichen Beschreibung zu spezifizieren. Ein Rechteck z.B. kann wie Listing 2.1 zeigt charakterisiert werden. Das Einlesen der LADDER-Dateien erfolgt mit ANTLR[6] und den in A.1 aufgeführten Grammatiken.

Neben Namen und Langtext besteht eine Symbolbeschreibung aus Komponenten (`components`) und Einschränkungen (`constraints`) sowie Vererbungs- und Typ-Informationen (`isA`, `implements`), Pseudonymen (`aliases`) und Zuweisungen (`mappings`).

Komponenten

Die Komponenten sind die Bestandteile des Symbols. Dies können Primitive (s. 2.1) wie Linien (`line`), Punkte (`point`) und Ellipsen (`ellipse`) oder -Segmente (`arc`) aber auch ganze Symbole sein. So können komplexe Symbolen aus mehreren einfachen zusammengesetzt werden, was die Wiederverwendbarkeit und Lesbarkeit fördert. Ausserdem kann die Erkennung bereits nach wenigen gezeichneten Komponenten beginnen. Jede Kompo-

[6] ANother Tool for Language Recognition, `http://www.antlr.org`

nente besitzt eine Menge von Attributen (Linien z.B. ihre beiden Endpunkte), die für Einschränkungen verwendet werden können.

Bemerkung Zum Zeitpunkt dieser Arbeit werden die erkannten Symbole verworfen, sobald sie einer Datenstruktur zugeordnet worden sind. Besteht also ein Symbol *B* aus einem *A* und einigen weiteren Komponenten und der Benutzer zeichnet zuerst *A*, so wird dieses gegebenenfalls einer Datenstruktur zugeordnet. Dadurch wird eine Erkennung von *B* verunmöglicht.

Einschränkungen

Die Einschränkungen beschreiben geometrische Eigenschaften der Komponenten sowie deren relative Lage. Ein `constraint` besteht aus einem Prädikat, einem oder mehreren Operanden und einer optionalen Gewichtung. Häufig verwendete Einschränkungen wie zusammenfallende Punkte (`coincident`), Ausrichtung (`horizontal`, `vertical`) oder relative Lage (`equalLength`, `aboveOf`, `rightOf`, `longer`, `perpendicular`) sind bereits implementiert, weitere können als Plug-In einfach realisiert werden. Eine Einschränkung ist für bestimmte Operanden-Typen definiert, `horizontal` beispielsweise für Linien, `coincident` nur für Punkte.

Die Auswertung einer bestimmten Belegung einer Einschränkung liefert den Übereinstimmungsgrad, d.h. eine Aussage darüber, wie stark die Einschränkung auf konkrete Operanden (Primitive oder vollständige Symbole) zutrifft. Der Übereinstimmungsgrad liegt zwischen 0 und 1 (inklusive).

Das Gewicht wird benutzt um dem unterschiedlichen Einfluss geometrischer Eigenschaften auf die Erkennung eines bestimmten Symbols Rechnung zu tragen. Das Gewicht muss zwischen 0 und 100 liegen, wobei eine mit 100 bewertete Einschränkung zu einem Übereinstimmungsgrad von 1 (vollständige Übereinstimmung) führt.

Bemerkung Da das Gewicht sowohl abhängig von allfälligen weiteren Einschränkungen passend zum Symbol gewählt werden muss, als auch von der Implementation des `IConstraints` abhängig ist, muss der passende Wert experimentell ermittelt werden. Es ist denkbar, dass dieser Wert zukünftig in einer Trainigsphase algorithmisch ermittelt werden kann.

„Eine rechtwinklige Figur mit vier Seiten.“

```
1  (define shape Rectangle
2    (description ”A␣simple␣rectangle”)
3    (components
4      (Line a)
5      (Line b)
6      (Line c)
7      (Line d)
8    )
9    (constraints
10     (coincident a.P2 b.P1 80)
11     (coincident b.P2 c.P1 80)
12     (coincident c.P2 d.P1 80)
13     (coincident d.P2 a.P1 80)
14     (perpendicular a b 60)
15     (perpendicular c d 60)
16   )
17 )
```

Listing 2.1: Beispiel für natürliche bzw. genaue Spezifikation von Symbolen (Rechteck)

Vererbung

Mit `isA` kann ein Symbol als abgeleitete Instanz eines Basis-Symbols definiert werden. Das neue Symbol erbt dabei sämtliche Komponenten und Eigenschaften des Basis-Symbols.

Typen

Das Schlüsselwort `implements` erlaubt, ein Symbol als konkrete Implementation eines abstrakten Typs festzulegen. Als abstrakte Typen werden Knoten, Verbindungen oder andere Symbole verstanden, die verschiedene konkrete Ausprägungen annehmen können. So können z.B. Kreise, Rauten oder Rechtecke als Knoten definiert werden. Eine Verbindung muss einen Ursprung und ein Ziel zur Verfügung stellen (Listing 2.2), ob der Benutzer einen Pfeil, Doppelpfeil oder bloss eine Linie zeichnet, muss aber nicht zwingend von Bedeutung sein.

Abstrakte Symbole Wird ein Symbol als `abstract` definiert, können ausser den Komponenten keine weiteren Eigenschaften festgelegt werden.

```
(define abstract shape Link
  (components
    (Point from)
    (Point to)
  )
)
```

Listing 2.2: Beispiel für ein abstraktes Symbol

Zuweisungen

Mittels Zuweisungen werden konkrete Komponenten(teile) eines Symbols als Eigenschaften des implementierten abstrakten Symbols festgelegt (Listing 2.3).

```
(define shape Arrow
  (implements Link)
  (components
    (Line shaft)
    (Line head1)
    (Line head2)
  )
  (constraints
    //...
    (coincident shaft.P1 head1.P1 50)
    (coincident shaft.P1 head2.P1 50)
  )
  (mappings
    (Link.from shaft.P1)
    (Link.to shaft.P2)
  )
)
```

Listing 2.3: Beispiel für die Implementation eines abstrakten Symbols

Pseudonyme

Pseudonyme tragen zur besseren Lesbarkeit von Symbolbeschreibungen bei, indem einfache Bezeichner anstelle längerer Komponententeile verwendet werden können (Listing 2.4).

```
1   //...
2   (aliases
3     (Pseudonym Langer.und.komplizierter.Komponent.Teil)
4   )
```

Listing 2.4: Beispiel für die Verwendung von Pseudonymen

2.2.2. Matching

Der entscheidende Schritt für die Symbolerkennung ist das Matching, d.h. die Bewertung von Primitiven anhand der Einschränkungen. Dieser Prozess muss schnell und genau sein, so dass keine Zwangspausen entstehen oder falsche Symbole erkannt werden. Für die Evaluation der Einschränkungen wird ein Entscheidungsbaumverfahren genutzt.

In einer ersten Iteration wurde eine naive Version des Entscheidungsbaumes implementiert, um einen Eindruck der grundlegenden Funktionalität zu erhalten. Obwohl die Erkennungsrate und die Geschwindigkeit subjektiv bereits ausreichend erschien, wurden Überlegungen in Richtung Bayesian Belief Network (BBN) gemacht, insbesondere um die verschiedenen Entscheidungsbäume für die einzelnen Symbole verschmelzen zu können. Der zusätzliche Aufwand und die erhöhte Komplexität scheinen den gewonnenen Nutzen der grösseren Flexibilität aber zu überwiegen, besonders da die Geschwindigkeit der aktuelle Implementation in den bisherigen Anwendungsfällen mehr als ausreichend ist. In der jetzigen Lösung kann der Bewertungsmechanismus dennoch leicht ausgetauscht werden, um z.B. eine benutzerdefinierte Trainingsphase der Symbolerkennung zu verwirklichen. Aus den Diskussionen rund um die BBNs entstand die Idee der gewichteten Einschränkungen, die in der vorhandenen Heuristik einbezogen werden. Vorschläge zur weiteren Verbesserung der Symbolerkennung sind im Abschnitt 3 zu finden.

Entscheidungsbaum

Der Entscheidungsbaum erfüllt zwei Aufgaben:

- Zuweisung von gezeichneten Primitiven bzw. bereits erkannten Symbolen zu Operanden der Einschränkungen des zu bewertenden Symbols
- Bewertung des Symbols anhand der Einschränkungen

Dazu wird pro zu erkennendem Symbol ein Baum erzeugt. Eine Ebene des Baumes entspricht einer Einschränkung und enthält als Knoten alle möglichen Paarungen ⟨Komponenten, Operanden⟩. Für jede dieser Paarungen wird anschliessend der Übereinstimmungsgrad berechnet. Abbildung 2.1 zeigt einen Ausschnitt dieses Baumes für ein Rechteck, das wie in Listing 2.1 spezifiziert ist. Der Übereinstimmungsgrad einer gegebenen Menge von Komponenten mit den gegebenen Zuweisungen für ein bestimmtes Symbol ist dann das (gewichtete) Produkt der Übereinstimmungsgrade entlang der Kanten eines solchen Baumes. Der Gesamtübereinstimmungsgrad für das Symbol ist somit das Maximum aller Produkte.

Es ist klar, dass ein solcher Baum sehr schnell sehr gross werden kann. Für ein Dreieck z.B., das gegen vier gezeichnete Linien abgeglichen werden soll, wird ein Baum erzeugt, der 192 Knoten enthält. Für das Rechteck sind es bereits 384 Knoten.[7]

[7] Für ein n-Eck gegen m Linien müssen aus den m Linien n ausgewählt werden. Diese können auf $n!$ verschiedene Arten den Komponenten zugeordnet werden. Jede Linie wiederum hat zwei Orientierungen:

$$\binom{m}{n} \cdot n! \cdot 2^n = \frac{m!}{(m-n)!} \cdot 2^n$$

Heuristik Gesucht ist also eine Heuristik, die sowohl die Breite des Baumes als auch die Tiefe einzelner Äste verringert. Offensichtlich[8] gilt:

- Der Übereinstimmungsgrad einer Einschränkung liegt zwischen 0 und 1
- Das Gewicht einer Einschränkung liegt zwischen 0 und 1 (0 und 100%)
- Der Übereinstimmungsgrad eines Symbols liegt zwischen 0 und 1
- Die Übereinstimmungsgrade nehmen entlang eines Pfades von der Wurzel gegen die Blätter monoton ab

Ist ein Übereinstimmungsgrad eines Blattes bekannt, müssen Pfade, deren aktueller Übereinstimmungsgrad kleiner oder gleich dem bekannten ist, nicht mehr weiter verfolgt werden. Zudem kann der jeweils erfolgsversprechendste Pfad besucht werden, d.h. derjenige Knoten einer Ebene, welcher den höchsten Übereinstimmungsgrad besitzt.

Damit die Äste möglichst nah an der Wurzel abgeschnitten werden können, müssen sich die Übereinstimmungsgrade früh stark unterscheiden. Dies kann durch eine Sortierung der Einschränkungen nach Gewichten begünstigt werden. Einschränkungen mit hohem Einfluss auf den Übereinstimmungsgrad werden vor solchen mit kleinem Einfluss berechnet.

Der Erfolg dieser Branch and Bound Heuristik hängt von der Implementation der Einschränkungen, der Symbolspezifikation und den gezeichneten Primitiven ab. Die Auswirkungen dieser Faktoren werden im Abschnitt 2.3.3 besprochen.

2.3. Implementation

Dieser Abschnitt dient als roter Faden durch die an der Symbolerkennung beteiligten Komponenten. Bei der Orientierung hilft dabei die Abbildung 2.2. Die `AlgoriaEngine` ist der Controller für den gesamten Prozess und steuert das Zusammenspiel der einzelnen Komponenten. Plugin-Architektur und Dependency Injection werden mit dem Managed Extensibility Framework (MEF) realisiert, das seit .Net 4.0 Teil der Standardbibliothek ist. Im- und Export werden über Attribute gesteuert (Listing 2.5).

2.3.1. Primitive Komponenten

Eingabe

Das .Net Framework bietet zur Eingabe von digitaler Tinte den `InkCanvas` aus dem System.Windows.Controls Namespace an. Diese Komponente schluckt aber auch alle weiteren Eingaben - wie Klicks oder Mausbewegungen. Der eigene `InkCanvasLight` reicht entsprechende Ereignisse an Kind-Elemente (z.B. bereits erkannte Datenstrukturen) weiter.[9] Ein `Stroke` entspricht einer gezeichneten Linie und besteht aus mindestens einem `StylusPoint` und Attributen (Farbe, Pinselform, Strichdicke, ...) zur Darstellung auf einer *ink-enabled* Komponente, wie dem `InkPresenter`. Jeder `StylusPoint` enthält eine Koordinate und eine Druckinformation sowie weitere herstellerabhängige Eigenschaften (z.B. Ausrichtung des Stiftes, Auflösung des Digitizers). [Mic10]

---

[8] c_i, w_i Übereinstimmungsgrad, Gewicht der Einschränkung i
$c_i, w_i \in \mathbb{R}, 0 \leq c_i \leq 1, 0 \leq w_i \leq 1 \Rightarrow 0 \leq c_i + w_i \cdot (1 - c_i) \leq 1 \Rightarrow 0 \leq \prod(c_i + w_i \cdot (1 - c_i)) \leq 1$
D.h. für jede Ebene l des Baumes gilt $\prod_{i \leq l} \leq \prod_{i \leq l-1}$

[9] Dies wird unter anderem für die Schrift- und Gestenerkennung sowie das Ausführen von Operationen auf Datenstrukturen und -elementen benötigt.

Componizer

Die Eingaben (Strokes) des Benutzers auf dem AlgoriaInkControl werden zuerst dem .Net InkAnalyzer übergeben. Dieser erzeugt einen Baum aus `ContextNodes`. Eine einzelne `ContextNode` kann Text (Schreibfläche, Absatz, Zeile, Wort) oder Zeichnungen enthalten.[10] Der Baum ist dynamisch und ändert mit jedem hinzugefügten oder entfernten `Stroke`. Dabei werden Gestaltprinzipien (wie Nähe, Ausrichtung, ..) berücksichtigt, so dass z.B. zwei bisher unabhängige `Strokes` zu einem Rechteck zusammengefasst werden, wenn die fehlenden Seiten hinzu kommen. Mittels `HintNodes` (selbst auch `ContextNodes`) kann gesteuert werden, ob und wie der `InkAnalyzer` Text erkennt. Für die Symbolerkennung wird die Handschrifterkennung ausgeschaltet. Trotzdem nutzt der `InkAnalyzer` Komponenten des Betriebsystems, die nur in den häufigsten Sprachen der installierten *Language Packs* zur Verfügung stehen. Ist die ‚falsche' Sprache eingestellt, funktioniert die Analyse nicht.

Der .Net `InkAnalyzer` kann Ellipsen, Kreise, Dreiecke (allgemeine, gleichschenklige, gleichseitige, rechtwinklige), Vierecke (allgemeine, Rechtecke, Quadrate), Rauten, Trapezoide, Parallelogramme, Fünfecke und Sechsecke bestimmen und liefert diese als `InkDrawingNode`. Schlägt die Analyse durch .Net fehl, so wird versucht Punkte, Linien, Linienzüge und Kreisbögen zu erkennen. Jetzt noch nicht identifizierte Eingaben werden als Pfade behandelt. Um von den technischen Details zu abstrahieren werden schliesslich sämtliche Primitive in eigenen Objekten (`IComponent`) gekapselt. Bis auf den Pfad sind diese Komponenten bereits vereinfachte Darstellungen des ursprünglich Gezeichneten. Linien z.B. bestehen nur noch aus zwei Punkten. Das Austauschen der Zeichnung durch die erkannten Primitiven gibt unmittelbare Rückmeldung und zeigt dem Benutzer, wie Algoria seine Eingabe verstanden hat (Abbildung 2.3).

Die AlgoriaEngine erzeugt aus den Primitiven einzelne atomare Symbole (`*Part`), die Zugriff auf grundlegende geometrische Eigenschaften bieten. N-Ecke werden in ihre Primitive (d.h. Liniensegmente) aufgespalten um eine einheitliche Symbolsynthese zu ermöglichen. Der Symbolizer kann nun versuchen, die mit LADDER beschriebenen Symbole zusammenzusetzen.

2.3.2. Symbolbeschreibung

LADDER

Für das Einlesen der LADDER-Files wird ANTLR verwendet. Mit ANTLRWorks[11] lassen sich die Grammatiken komfortabel bearbeiten und direkt die Hilfsklassen erzeugen.[12] Aus der Grammatik in Listing A.1 werden Lexer und Parser generiert, die aus den LADDER-Dateien einen Abstract Syntax Tree (AST) aufbauen. Ein Tree-Walker (Listing A.2) instanziiert und initialisiert die passenden Symbol- und Einschränkungsbeschreibungen (`BaseSymbol`, `BaseConstraint`). Der Symbolizer baut daraus die `isA`-Hierarchie der Symbole (z.B. Viereck → Rechteck → Quadrat) auf.

[10] Programmatisch können auch sogenannte *non-ink* Objekte (Text, Bilder) hinzugefügt werden.
[11] `http://www.antlr.org/works/`
[12] Der generierte Code ist unübersichtlich (der Lexer ist über 1000, der Parser über 2400 Zeilen lang) und führt zu zahllosen Warnungen, die aber getrost ignoriert und von der Code Analyse oder Check-In-Policies ausgenommen werden können.

Constraints

Ein `Constraint` ist die Implementation einer Einschränkung und bietet Methoden um den Übereinstimmungsgrad für gegebene Operanden zu berechnen. Einige oft benutzte Constraints wie `coincident`, `equalLength`, `rightOf` und `aboveOf` (Listing 2.6) sind bereits implementiert. Weitere können (dank MEF zur Laufzeit) hinzugefügt werden, indem eine entsprechende DLL erstellt und in das Plugins-Verzeichnis kopiert wird.

2.3.3. Symbolizer

Der Symbolizer versucht atomare Symbole und eventuell bereits erkannte zusammengesetzte Symbole zu neuen Symbolen zu kombinieren. Dazu wird pro potentiell passendem Symbol ein Entscheidungsbaum aufgebaut. Auf jeder Ebene wird dabei der nächste, noch nicht berechnete Constraint mit dem höchsten Gewicht ausgewählt und je ein Zweig für jede mögliche Belegung seiner Operanden erzeugt. Ein Ausschnitt aus dem so aufgebauten Entscheidungsbaum ist in Abbildung 2.4 dargestellt. Der Entscheidungsbaum wird dabei nicht explizit erzeugt, sondern durch rekursive Funktionsaufrufe auf dem Stack abgelegt.

Einflüsse

Die Effizienz des eingesetzten Verfahrens hängt von drei Variablen ab, auf die im Folgenden genauer eingegangen wird.

Implementation der Constraints Grössten Einfluss auf die Anzahl effektiv berechneter Pfade (bei gegebener Symbolbeschreibung) hat die Implementation der Constraints. Von besonderer Bedeutung ist die Trennschärfe, d.h. wie gut der Constraint passende von weniger passenden Parametern unterscheidet. Für den `coincident`-Constraint z.B. heisst das, dass die maximale Entfernung zweier Punkte (da wo die Übereinstimmung null wird) möglichst klein gewählt werden soll. Einfacher sind binäre Constraints (`rightOf`, `aboveOf`), komplizierter multivariate wie `equalLength`: ist die Übereinstimmung abhängig von der absoluten Länge der Linien, soll eine Quantisierung oder Thresholding angewandt werden? Differenziert ein Constraint schlecht, so trägt er wenig zur Entscheidungsfindung bei. Es gilt, eine möglichst hohe Trennschärfe zu erreichen, ohne Benutzererwartungen zu enttäuschen.[13]

Spezifikation in LADDER Eine gute Beschreibung in LADDER hilft, den Entscheidungsbaum klein zu halten. Die Symbolspezifikation sollte weder unter- (ungenau) noch überdeterminiert (zu restriktiv) sein. Die Beispiele in Listing 2.7 verdeutlichen den Zusammenhang. Ersteres führt zu falsch erkannten Symbolen (im Beispiel alle Vierecke), letzteres zu übergrossen Entscheidungsbäumen. Zudem erschweren grosse Unterschiede in der Anzahl von Constraints die Vergleichbarkeit der Übereinstimmungsgrade verschiedener Symbole.

Konkrete Skizze Letzlich ist die Grösse des Entscheidungsbaumes auch von der konkreten, zu bewertenden Skizze abhängig. Grundsätzlich können mehr Pfade abgeschnitten werden, wenn die Skizze dem zu erkennenden Symbol (d.h. der Symbolbeschreibung) genauer entspricht, da dann die Auswertung der Constraints klarer differenziert.

[13] Soll die Übereinstimmung für `perpendicular` bei 45 oder bei 0 Grad oder irgendwo dazwischen 0 betragen? Stehen zwei Linien im Winkel von 80° senkrecht aufeinander?

Resultate

Aufgrund der im vorhergehenden Kapitel dargelegten Einflüsse ist eine Quantifizierung der durch die eingesetzte Heuristik erreichten Beschränkung des Entscheidungsbaumes schwierig. Einige empirisch ermittelte Werte werden in Tabelle 2.1 aufgeführt. Falls ein Orakel für die Zuweisung von gezeichneten Symbolen zu Operanden der Constraints zur Verfügung stehen würde, müssten jeweils nur die tatsächlich verwendeten Einschränkungen berechnet werden. Entlang eines Pfades werden im besten Fall jeweils nur die pfadlokalen Constraints ausgewertet, um zu entscheiden, welcher Ast weiter verfolgt werden soll. Die Spalte *mit Heuristik* schliesslich führt die empirisch ermittelten Werte für die vorliegende Implementation auf. Dass für das Rechteck rund doppelt so viele Constraints ausgewertet werden, wie auf den ersten Blick erwartet, liegt an der Symmetrie und der Gewichtung der Constraints für das Rechteck. Entlang eines Pfades nimmt die Übereinstimmung ab, wodurch ein weiterer Pfad berechnet wird, dessen Übereinstimmungsgrad ebenso abnimmt… Ein Ausweg ist, die Symbolbeschreibung nicht-symmetrisch zu gestalten, indem z.B. ein `leftOf` Constraint verwendet wird.

	Anzahl berechnete Constraints			
Symbol	ganzer Baum	mit Orakel	ein Pfad	mit Heuristik
Viereck	1008	4	55	55
Rechteck	1776	6	57	120
Gleichschenkliges Dreieck	126	4	13	13

Tabelle 2.1.: Resultate der Entscheidungsbaum-Heuristik (Erläuterung s. Abschnitt 2.3.3)

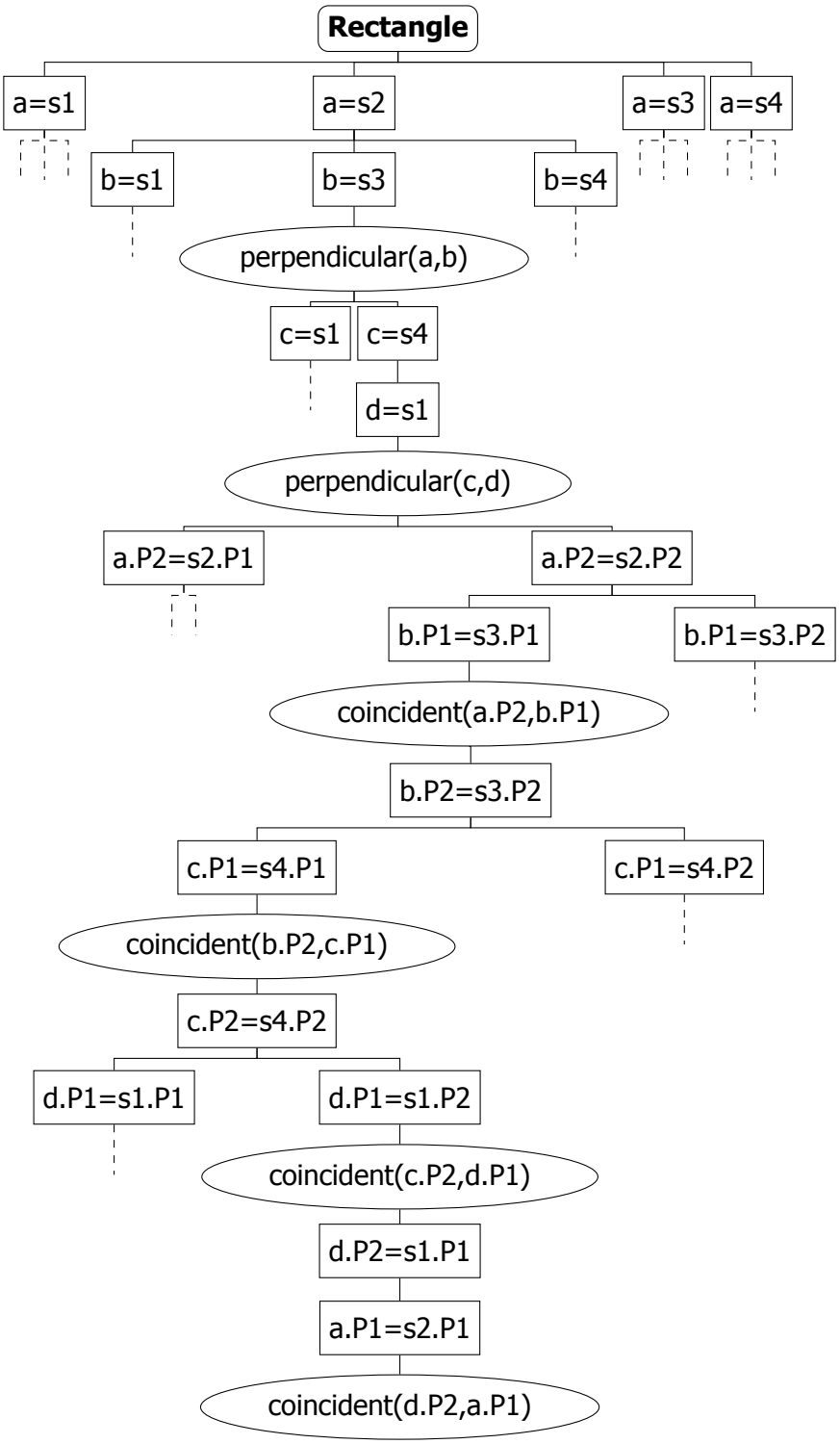

Abbildung 2.1.: Ausschnitt aus dem Entscheidungsbaum des Rechtecks

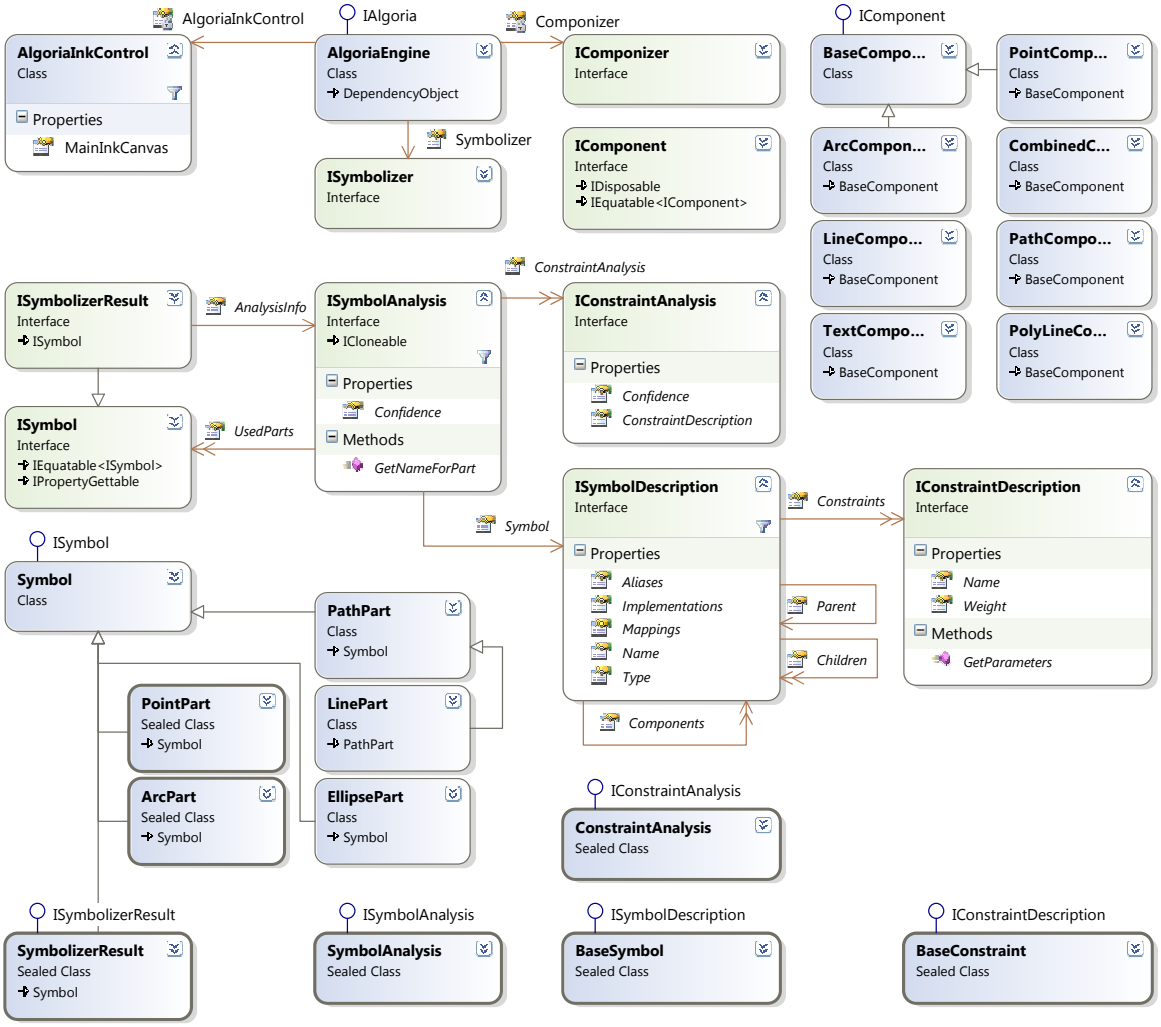

Abbildung 2.2.: Übersicht der an der Symbolerkennung beteiligten Klassen und Interfaces

```
1  [Export]
2  class Initializer //...

4  [Export(typeof(IDependencyObject))]
5  class Foo : IDependencyObject //...

7  [Export(typeof(IItem))]
8  [PartCreationPolicy(
9   CreationPolicy.NonShared
10 )]
11 class Bar : IItem //...
```

```
1  class DependentObject{
2   [ImportingConstructor]
3   DependentObject(Initializer i) //...

5   [Import]
6   IDependencyObject DepObj {get; set;}

8   [ImportMany]
9   private ICollection<IItem> items;
10 }
```

Listing 2.5: Dependency Injection mit MEF

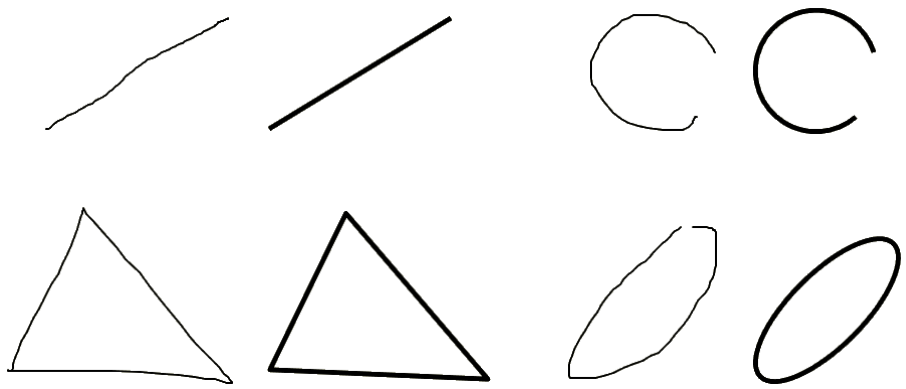

Abbildung 2.3.: Gezeichnete und vom Componizer erkannte Primitive.

```
1  using System;
2  using System.ComponentModel.Composition;

4  namespace IMVS.Algoria.Plugin.Constraints
5  {
6      [Export(typeof(IConstraint))]
7      [PartCreationPolicy(CreationPolicy.Shared)]
8      public class AboveOf : IConstraint
9      {
10         public string Name
11         {
12             get { return ”aboveOf”; }
13         }

15         public Type[] ComponentTypes
16         {
17             get { return new[] { typeof(ISymbol), typeof(ISymbol) }; }
18         }

20         /// <summary>
21         /// Calculates whether a symbol is above another symbol
22         /// </summary>
23         /// <param name=”components”>two symbols [A, B]</param>
24         /// <returns>
25         /// 1 if the bottom of A is at the same height or above the top of B
26         /// 0 if A’s center axis is at the same height or below B’s center axis
27         /// a linear interpolation inbetween
28         /// </returns>
29         public double CalculateConstraint(params ISymbol[] components)
30         {
31             ISymbol above = components[0];
32             ISymbol below = components[1];

34             double midAbove = above.BoundingBox.Top + above.Height/2;
35             double midBelow = below.BoundingBox.Top + below.Height/2;
36             double dNorm = (midBelow - midAbove) / (above.Height/2 + below.Height/2);
37             return Math.Min(Math.Max(dNorm, 0), 1);
38         }
39     }
40 }
```

Listing 2.6: AboveOf Constraint

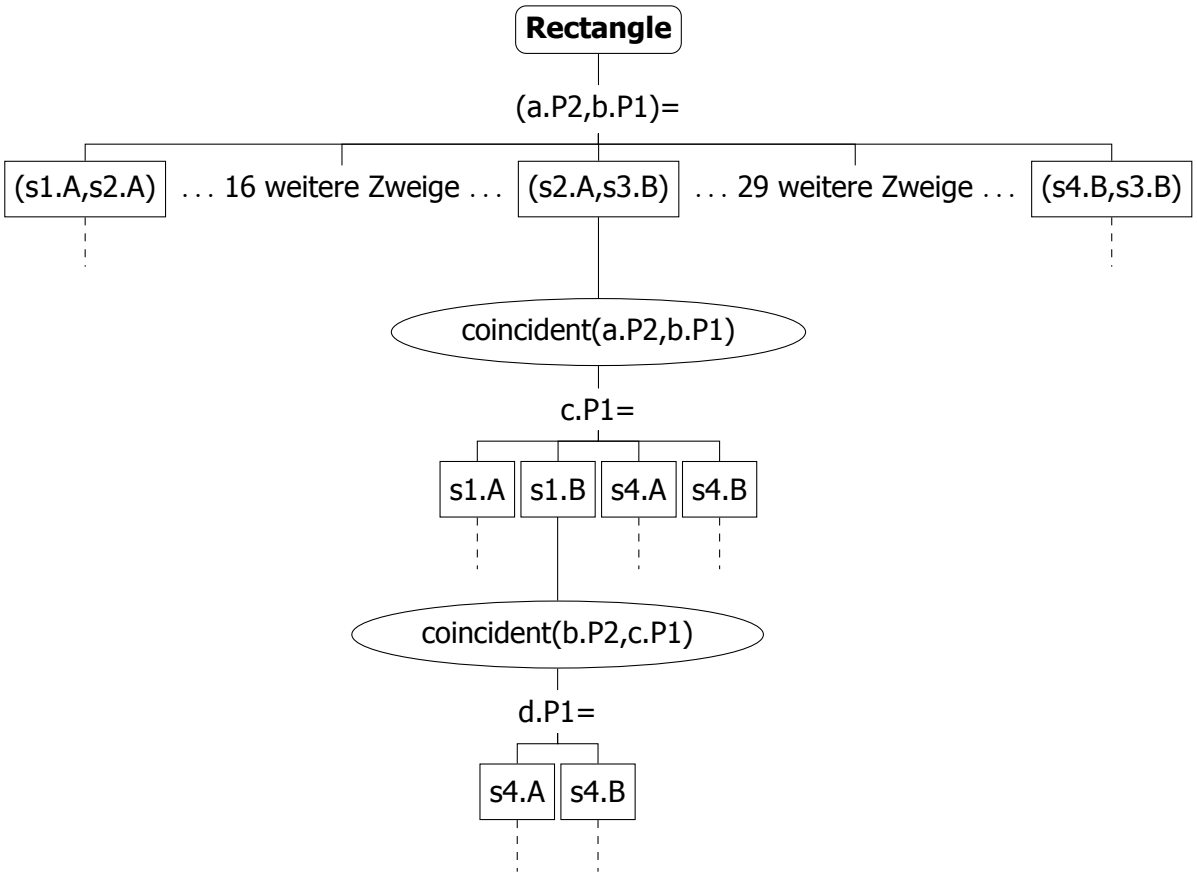

Abbildung 2.4.: Ausschnitt des durch die Implementierung erzeugten Entscheidungsbaumes des Rechtecks

```
(constraints
  (coincident a.P2 b.P1 80)
  (coincident b.P2 c.P1 80)
  (coincident c.P2 d.P1 80)
  (coincident d.P2 a.P1 80)
)
```

```
(constraints
  (coincident a.P2 b.P1 80)
  (coincident b.P2 c.P1 80)
  (coincident c.P2 d.P1 80)
  (coincident d.P2 a.P1 80)
  (sameLength a c 60)
  (sameLength c d 60)
  (perpendicular a b 60)
  (perpendicular c d 60)
)
```

Listing 2.7: Unter- bzw. überdeterminierte Symbolspezifikation des Rechtecks

3. Schlusswort

Das Resultat dieser Arbeit ist eine, nach dem aktuellen Stand von Algoria mehr als ausreichend effiziente und genaue, Lösung für die Symbolerkennung. Neben diesem unmittelbaren Ergebnis haben sich folgende Überlegungen ergeben bzw. sind weitere Optimierungsmöglichkeiten offenbar geworden, die Raum für kommende Entwicklungen bieten.

- Inkrementeller Branch and Bound
  Anstatt für jedes Symbol mindestens einen kompletten Pfad zu berechnen, kann der Übereinstimmungsgrad schrittweise ermittelt werden, indem nur für das jeweils beste Symbol eine weitere Ebene des Baumes erstellt wird.
- Symmetrien
  Wie Tabelle 2.1 zeigt können bestimmte Konfigurationen dazu führen, dass mehr Constraints als unbedingt notwendig berechnet werden. Für das Rechteck z.B. werden die acht rotationssymmetrischen und gespiegelten Formen berechnet. Als Workaround kann z.B. die Sortierung der Constraints so geändert werden, dass zuerst die gezeichneten Linien den Komponenten zugewiesen werden. Eine elegante Lösung, die Symmetrien ausnutzt, muss noch gefunden werden.
- Training der Gewichte für die Constraints
  Das manuelle Ermitteln bzw. Schätzen der Gewichte für die Constraints ist unbefriedigend. Denkbar wäre eine automatisierte Lösung, die die Gewichte schätzt und selbstlernend anpasst. In einem ersten Schritt könnten auch Gewichtsklassen (z.B. `relaxed`, `normal`, `hard`) genutzt werden.
- Bayesian Belief Network
  Der Ansatz mit BBNs scheint vielversprechend und interessant. Möglich wäre z.B. eine Realisation mit Hilfe von Infer.NET [MWGK09], einem Microsoft Research Framework für logisches Schliessen.
- `aliases` und `mappings`
  Pseudonyme und Zuweisungen werden zwar geparst, Algoria weiss aber noch nichts mit ihnen anzufangen und ignoriert sie stillschweigend.
- Alternative Symbole
  Von der Texteingabe mittels Stift bereits bekannt sind die Listen mit Alternativen für erkannte Wörter. Mit steigender Anzahl Symbole und somit abnehmender relativer Erkennungsgenauigkeit, könnte der Bedarf, in Algoria etwas ähnliches anzubieten, zunehmen.
- Validierung von LADDER
  Die Symbolspezifikationen werden nur auf syntaktische Korrektheit überprüft. Falsche Komponentennamen oder fehlende Constraints werden erst bemerkt, wenn ein Symbol erkannt werden soll. Eine statische Analyse könnte auch den erzeugten Entscheidungsbaum darstellen oder Hinweise auf über- oder unterdeterminierte Spezifikationen geben.

Glossar

ContextNode
: Resultat einer Analyse eines oder mehrer `Strokes` durch den `InkAnalyzer`. Kann u.a. Text, geometrische Figuren (`InkDrawingNode`) oder eine Sammlung von unerkannten Objekten beinhalten.

HintNode
: `ContextNode` die das Verhalten des `InkAnalyzers` steuert

InkAnalyzer
: .NET Komponente zur Analyse von `Strokes`. Enthält u.a. Texterkennung

InkDrawingNode
: Von der `ContextNode` abgeleitete Klasse für geometrische Figuren.

Stroke
: Gezeichnete Linie bestehend aus `StylusPoints`

StylusPoint
: Durch den Digitizer erzeugten Punkt eines Zeigegerätes (Stift, Maus)

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Cha03] ChannelPartner. Absatzzahlen von Tablet-PCs weiterhin ein Trauerspiel, 2003.
`http://www.channelpartner.de/news/209592/index.html`.

[Ham07] Tracy Anne Hammond. *LADDER: A Perceptually-based Language to Simplify Sketch Recognition User Interface Development*. PhD thesis, Massachusetts Institute of Technology, 2007.

[HD03] Tracy Anne Hammond and Randall Davis. LADDER: A Language to Describe Drawing, Display, and Editing in Sketch Recognition. In *Proceedings of the 2003 International Joint Conference on Artifical Intelligence*, pages 461–467, 2003.

[Mic10] Microsoft Corporation. *Ink Analysis Overview*, MSDN Library for Visual Studio 2010, 2010.

[MWGK09] T. Minka, J.M. Winn, J.P. Guiver, and A. Kannan. Infer.NET 2.3, 2009. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

A. Anhang

A.1. ANTLR LADDER-Grammatiken

```
grammar Ladder;
options
{
  language=CSharp2;
  output=AST;
}

// Primitives
tokens{
  DEF = 'define' ;
  LPAR  = '(' ;
  RPAR  = ')' ;
  ABSTRACT
    = 'abstract' ;
  SHAPE = 'shape' ;
  DESC  = 'description' ;
  ISA = 'isA' ;
  IMPLEMENTS
    = 'implements' ;
  COMPONENTS
    = 'components' ;
  CONTEXT = 'context' ;
  CONSTRAINTS
    = 'constraints' ;
  ALIASES = 'aliases' ;
  MAPPINGS
    = 'mappings' ;
  NOT = 'not' ;
  QUOTE = '"';
  NOOP ;
}

@lexer::namespace { IMVS.Algoria.Ladder }
@parser::namespace { IMVS.Algoria.Ladder }

// Parser
shape : LPAR DEF ABSTRACT? SHAPE Ident description? isA? implements? components?
    constraints? aliases? mappings? RPAR -> ^(Ident ABSTRACT? description? isA?
    implements? components? constraints? aliases? mappings?) ;
description
  : LPAR DESC String RPAR  -> ^(DESC String) ;
isA : LPAR ISA Ident RPAR -> ^(ISA Ident) ;
implements
  : LPAR IMPLEMENTS Ident+ RPAR -> ^(IMPLEMENTS Ident+) ;
components
  : LPAR COMPONENTS component+ context* RPAR -> ^(COMPONENTS component+ context*) ;
component
  : LPAR n=Ident a=Ident RPAR -> ^($n $a) ;
context : LPAR CONTEXT n=Ident a=Ident RPAR -> ^(CONTEXT $n $a) ;
constraints
  : LPAR CONSTRAINTS constraint+ RPAR -> ^(CONSTRAINTS constraint+) ;
constraint
  : LPAR term RPAR -> term;
term  : predicate property+ Number? -> ^(predicate property+ Number?) ;
predicate
  : Ident ;
aliases : LPAR ALIASES alias+ RPAR -> ^(ALIASES alias+) ;
alias : LPAR Ident property RPAR -> ^(Ident property) ; // TODO
property: Ident ('.' property)? -> ^(Ident property?) ;
mappings: LPAR MAPPINGS mapping+ RPAR -> ^(MAPPINGS mapping+) ;
mapping : LPAR super=property act=property RPAR -> ^(NOOP $super $act) ; // TODO
// Lexer
Ident : (Alpha | ALPHA) AlphaNum* ;
String  : QUOTE .* QUOTE ;
```

```
63  Number  : Digit+ ;
64  Whitespace
65    : (' ' | '\t' | '\n' | '\r' | '\f')+
66    { Skip(); }
67    ;
68  fragment AlphaNum
69    : ALPHA | Alpha | Digit ;
70  fragment ALPHA
71    : 'A'..'Z' ;
72  fragment Alpha
73    : 'a'..'z' ;
74  fragment Digit
75    : '0'..'9' ;
```

Listing A.1: ANTLR LADDER grammar

```
1   tree grammar LadderTree;
2   options {
3     language=CSharp2;
4     tokenVocab=Ladder;
5     ASTLabelType=CommonTree;
6   }
7   @treeparser::namespace { IMVS.Algoria.Ladder }

9   @header{
10  using System.Collections.Generic;
11  using IMVS.Algoria.Symbols;
12  }

14  shape returns [ISymbolDescription desc]
15    : ^(Ident ABSTRACT? description? isA?
16  {
17  if($ABSTRACT != null){
18    return null;
19  }else if($isA.type != null){
20    $desc = new BaseSymbol($Ident.Text, $isA.type);
21  }else{
22    $desc = new BaseSymbol($Ident.Text);
23  }
24  }
25  implements[desc.Implementations]? components[desc.Components]? constraints[desc.
        Constraints]? aliases[desc.Aliases]? mappings[desc.Mappings]?) ;

27  description
28    : ^(DESC String) ;

30  isA returns [string type]
31  : ^(ISA Ident)
32  {
33  $type = $Ident.Text;
34  } ;

36  implements[ICollection<string> imps]
37    : ^(IMPLEMENTS (Ident {$imps.Add($Ident.Text);})+);

39  components[ICollection<ISymbolDescription> parts]
40    : ^(COMPONENTS (component {$parts.Add($component.comp);})+ context*) ;

42  component
43    returns [ISymbolDescription comp]
44    : ^(n=Ident a=Ident)
45  {
46  $comp = new BaseSymbol($a.Text, $n.Text);
47  } ;

49  context : ^(CONTEXT n=Ident a=Ident) ;

51  constraints[ICollection<IConstraintDescription> constr]
52    : ^(CONSTRAINTS (constraint {$constr.Add($constraint.constr);})+) ;

54  constraint
55    returns [IConstraintDescription constr]
```

```
56    : term
57  {
58  if($term.weight != 0){
59    $constr = new BaseConstraint($term.name, $term.prop, $term.weight);
60  }else{
61    $constr = new BaseConstraint($term.name, $term.prop);
62  }
63  } ;

65  term  returns [string name, string[\] prop, int weight]
66    : {List<string> props = new List<string>();}
67      ^(predicate (property {props.Add($property.prop);})+ (Number{int.TryParse($Number.
            Text, out $weight);})?)
68  {
69  $name = $predicate.name;
70  $prop = props.ToArray();
71  } ;

73  predicate
74    returns [string name]
75    : Ident {$name = $Ident.Text;} ;

77  property
78    returns [string prop]
79    : ^(Ident {$prop = $Ident.Text;} (p=property {$prop += ".” + $p.prop;})?) ;

81  aliases[IDictionary<string,string> a]
82    : ^(ALIASES (alias {$a.Add($alias.name, $alias.prop);})+) ;

84  alias returns [string name, string prop]
85    : ^(Ident property)
86  {
87  $name = $Ident.Text; $prop = $property.prop;
88  } ;

90  mappings[IDictionary<string,string> maps]
91    : ^(MAPPINGS (mapping {$maps.Add($mapping.super, $mapping.act);})+) ;
92  mapping returns [string super, string act]
93    : ^(NOOP b=property a=property)
94  {
95  $super = $b.prop; $act = $a.prop;
96  } ;
```

Listing A.2: ANTLR LADDER tree grammar

Ehrlichkeitserklärung

Hiermit bestätigt der unterzeichnende Autor, dass alle nicht klar gekennzeichneten Stellen von ihm selbst erarbeitet und verfasst wurden.

Unterbözberg, September 2010

Raphael Schweizer