Fachhochschule Nordwestschweiz
Hochschule für Technik

Algoria

MSE P8 – Kombination von Datenstrukturen

März 2012

Autor: Raphael Schweizer

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

Fachhochschule Nordwestschweiz
Hochschule für Technik

Raphael Schweizer
Hinterer Hafen 348
5224 Unterbözberg

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

Satz: XeLaTeX

Algoria ist eine Anwendung zur Skizzenerkennung auf Convertibles. Es können typische Datenstrukturen aus der Informatik – wie Arrays, Listen und Bäume – erkannt werden. Auf den „zum Leben erweckten“ Strukturen können anschliessend häufige Operationen (z.B. Einfügen, Löschen, Suchen, Sortieren, etc.) animiert ausgeführt werden.

Dieses Dokument beschreibt die für das P8 MSE-Projekt durchgeführten Arbeiten mit besonderem Augenmerk auf einfachere Anwendbarkeit von Algoria sowie besserer Wartbarkeit des über Jahre gewachsenen Quellcodes.

Das Projekt Algoria ist *Work in Progress* in experimentellem Stadium. Die vorliegende Arbeit entspricht mindestens der Revision 4190 des Codes auf dem zur Entwicklung von Algoria verwendeten Versionierungsserver.

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Die Geburtsstunde von Algoria liegt bereits über drei Jahre zurück. Nach anfänglicher Finanzierung durch die Hasler Stiftung erfolgt die Weiterentwicklung seit einem Jahr im Rahmen von Studierendenprojekten (Bachelor- und Masterstufe). Wie bei solchen Projekten üblich (und erwünscht) werden in erster Linie die interessanten Probleme behandelt, die Implementierung der erarbeiteten theoretischen Lösungen lässt dabei oft den letzten Schliff vermissen, vor allem bedingt durch die Tatsache, dass der entsprechende Code nach Abschluss der Arbeiten kaum mehr gewartet wird.

1.2 Ziel

Die vorliegende Arbeit verfolgt die Verbesserung der Softwarequalität[1] von Algoria. Dazu werden die folgenden zwei Hauptziele definiert:

- Verbesserung der Codebasis in Hinblick auf Wartbarkeit, Zuverlässigkeit und Effizienz sowie funktionaler Vollständigkeit fragmenthafter Implementierungen
- Entwicklung weiterführender Konzepte für die bessere Benutzbarkeit, insbesondere im Hinblick auf kombinierte Datenstrukturen und natürlicheres Skizzieren von Symbolen

1.3 Vorgehen

In einer ersten Phase wurde die bestehende Codebasis konsolidiert, anschliessend die Erweiterung in Angriff genommen. Da zu Beginn des Projektes nicht abschliessend beurteilt werden konnte, welche Arbeiten mit welchem Aufwand verbunden sind und flexibel auf sich ändernde Anforderungen eingegangen wurde, erfolgte eine ständig aktualisierte Planung mittels häufiger, je nach Bedarf kurzfristig anberaumter, informellen Sitzungen.

Bemerkung Um während der Entwicklung die parallel laufenden Studierendenprojekte nicht zu behindern, wurde auf dem Versionierungsserver der neue Branch `Algoria-NG` erzeugt. Dieser benötigt eine aktuelle Vorab-Version von Visual Studio 11[2]. Sämtliche Änderungen bis auf die .NET 4.5 und VS 11 spezifischen Anpassungen wurden aber auch in den Hauptentwicklungszweig `Algoria-UI-branch` eingepflegt.

[1] vgl. ISO/IEC 9126
[2] `https://www.microsoft.com/visualstudio/11/en-us`

2 Konsolidierung

Um möglichst effizient eine nachhaltige Verbesserung der Codequalität zu erreichen, wurden für jedes betrachtete Merkmal einige konzentrierte Massnahmen ergriffen, die im Folgenden grob beschrieben werden sollen.

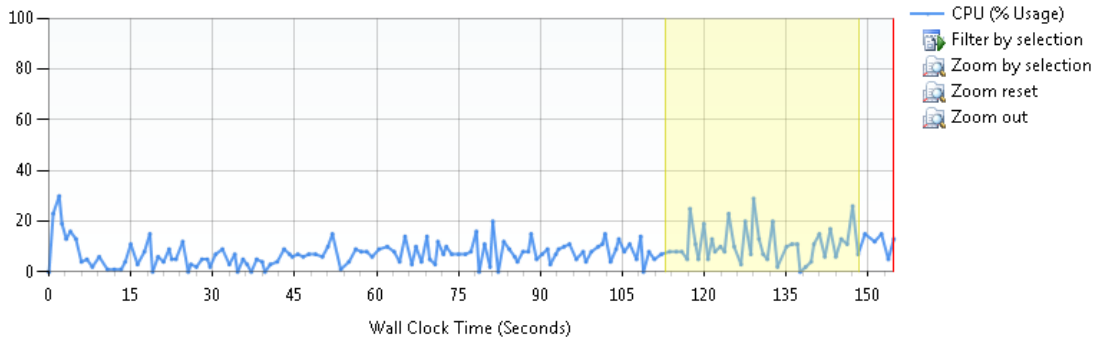
Wartbarkeit In bisherigen Projektarbeiten hat sich insbesondere die Analysierbarkeit als verbesserungswürdig erwiesen. Die Einarbeitung wird neu mit deutlich mehr und besseren Kommentaren und einer Definitionsdatei für den *Sandcastle Documentation Compiler*[3] erleichtert. Eine Kurz-Anleitung für die zur Entwicklung benötigten Tools – zusammen mit sinnvollen Einstellungen für diese – helfen Algoria-Juniorprogrammierern beim Einstieg. Besonders hervorzuheben ist hier *StyleCop*[4], ein Code-Analyse-Werkzeug das (nicht nur!) Java-Umsteiger erzieht und die Navigation in grösseren Klassen erleichtert, da unter anderem die Reihenfolge der Klassenbestandteile vereinheitlicht wird.

Zuverlässigkeit Besonders in den Datenstrukturen-Plugins treten immer wieder Fehler auf, meist entdeckt von neuen Benutzern, die die bekannten Pfade der Plugin-Entwickler verlassen. Aber auch im reiferen Code des Algoria-Kerns sind noch etliche Fehler verborgen. Insbesondere die Verringerung der Komplexität hilft vorgängig Fehler zu vermeiden und diese – beim dennoch unumgänglichen Auftreten – schneller zu finden und zu beheben. In diesem Sinn wurde sowohl Kern-Code als auch die Plugins auf mittlerweile überflüssigen oder redundanten Code untersucht und dieser entfernt. Wo das Aufräumen hoffnungslos erschien, wurde eine verbesserte Version neu implementiert (z.B. LinkedList).

Effizienz Für Algoria besonders wichtig ist die Performance der Skizzenerkennung, d.h. die Zeit ab dem Moment wo der Benutzer seine Eingabe abgeschlossen hat bis zum Erscheinen der erkannten und umgewandelten Symbole. Das Profiling mit einem ‚schwierigen' Symbol (`Star.ldr`, nichtkonvexer, nichtüberschlagener Fünfstern) hat einen unerwarteten Zeitfresser ergeben: die Auflistung der für ein Symbol benötigten Komponenten (Abbildung 2.1). Nach der entsprechenden Optimierung wird jetzt wie erwartet mit Abstand die meiste Zeit für die Auswertung des Entscheidungsbaumes inkl. Constraint-Berechnung und Symbol-Optimierung aufgewendet (Abbildung 2.2).[5] Viel Zeit beansprucht weiterhin die Suche nach dem richtigen Overload eines Constraints. Eine einfache Lösung wäre die Constraints unterschiedlich zu benennen, allerdings würde sich das Problem auch mit einem allfälligen Caching der berechneten Constraints entschärfen.

Funktionale Vollständigkeit Neben einigen Detailverbesserungen für Steuerelemente und Stift bzw. Mausgesten wurden vor allem die Datenstruktur-Plugins stark überarbeitet.

[3] `https://sandcastle.codeplex.com/`
[4] `https://stylecop.codeplex.com/`
[5] Absolute Zahlen können für diesen Anwendungsfall nicht ermittelt werden, da sich das Skizzieren und Erkennen nicht genügend gut und einfach automatisieren lassen.



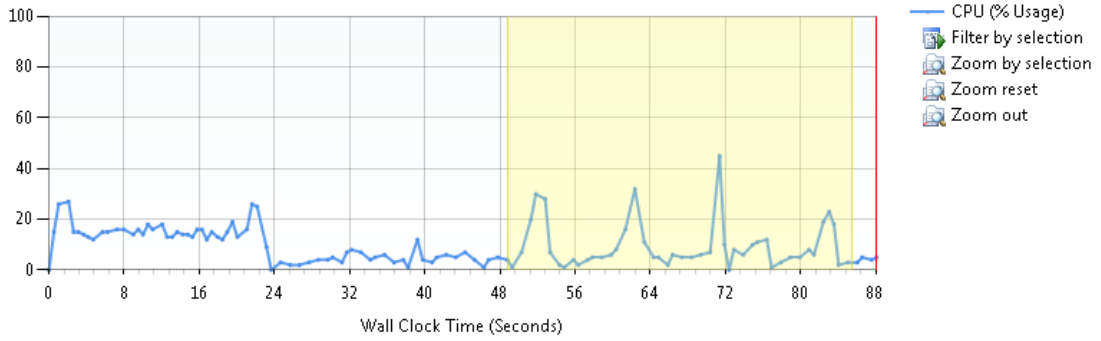
Hot Path

The most expensive call path based on execution times

Function Name	Elapsed Inclusive Time %	Elapsed Exclusive Time %
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.RecursiveCalcConstraints(valuetype S...	72.90	0.05
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.RecursiveCalcConstraints(valu...	**24.14**	**0.01**
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.PrepareAllParams(class IMVS....	**19.90**	**0.00**
IMVS.Algoria.Symbols.SymbolDescription.get_AllComponents()	**14.37**	**8.60**
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.CalculateConstraintConfidence...	**12.70**	**0.04**

Related Views: Call Tree Functions

Abbildung 2.1: Profilinglauf vor der Optimierung



Hot Path

The most expensive call path based on execution times

Function Name	Elapsed Inclusive Time %	Elapsed Exclusive Time %
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.RecursiveCalcConstraints(valuetype S...	49.63	0.04
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.RecursiveCalcConstraints(valuetyp...	41.54	6.20
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.CalculateConstraintConfide...	**11.53**	**0.08**
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.RecursiveCalcConstraints(v...	**9.98**	**0.02**
IMVS.Algoria.Symbols.BranchAndBound.BBHelper.PrepareAllParams(class IMV...	**9.91**	**0.01**

Related Views: Call Tree Functions

Abbildung 2.2: Profilinglauf nach der Optimierung

Insbesondere basieren jetzt alle Plugins auf den animierbaren Datentypen (nicht zu verwechseln mit den Animationen der Datenstrukturen selbst) und sind persistierbar. Erstmals können somit ganze Skizzen und Datenstrukturen gespeichert und wieder geladen werden.

2.1 Probleme

Als im Rahmen dieser Arbeit weiterhin ungelöstes Problem bleibt die Layouting-Mechanik (mit Verschieben und Grössenänderung) von Datenstrukturen bzw. deren Elementen.[6] Die bisher verwendeten Mechanismen (Triggern eines Layoutupdates aus der Business-Logik) scheinen ungeeignet zur verbreiteteren Implementation, ein passender Ersatz muss aber erst noch gefunden werden.[7] Als temporäre Massnahme wurde der fragliche Code auskommentiert und deutlich gemacht, dass dieser nicht weiter verwendet werden soll, bevor eine Lösung gefunden wurde.

Auch an einer anderen Stelle machen sich die Interna von WPF unangenehm bemerkbar: aus noch unbekannten Gründen können mit der jetzigen Implementierung keine Binärbäume zusammengestellt werden, bei der hierarchischen Verlinkung der aus animationstechnischen Gründen von `System.Windows.Media.Animation.Animatable` abgeleiteten Nodes wird eine kryptische Exception geworfen.

Eine erste Möglichkeit für die vertiefte Untersuchung dieser Probleme ist krankheitshalber gescheitert, der Autor musste ein Treffen mit Beat Walti, dem ehemaligen Algoria-WPF-Verantwortlichen, absagen. Ein neuer Termin konnte bis jetzt noch nicht gefunden werden, wird aber baldmöglichst nachgeholt.

[6] So werden z.B. bei der neu implementierten LinkedList je nach Verlinkungs-Reihenfolge einige Elemente abgeschnitten oder an einem falschen Ort angezeigt.

[7] Ein möglicher Ansatz ist die Verwendung eines benutzerdefinierten AttachedProperty pro Datenstruktur, analog zu `Canvas.Top` bzw. `Left`. Dieser Ansatz muss ausserhalb dieser Arbeit erprobt werden.

3 Kombinierte Datenstrukturen

Der zweite Teil der Arbeit widmet sich der Benutzbarkeit von Algoria. Um attraktiv zu sein muss Algoria vollständiger (bezüglich der beschreibbaren Symbole) und konformer (bezüglich erwarteter Skizzierungsmöglichkeiten) werden.

Insbesondere sollen kombinierte Datenstrukturen einfach beschrieben werden können. Als kombinierte Datenstruktur verstehen wir z.B. eine Hashmap, d.h. eine Datenstruktur, die selbst aus mehreren (einfachen) Datenstrukturen (aus einem Array und Listen) aufgebaut ist. Weitere Beispiele sind jagged Arrays oder gerichtete Graphen (aus einfach verketteten Listen) mit $\forall v \in V : Deg_{out}(v) \leq 1$.

Für die Realisation in Algoria gibt es folgende grundsätzlichen Möglichkeiten:

- Bis auf die Erkennung von einfachen Symbolen wird alle Funktionalität in Plugins implementiert.
- Algoria erkennt selbständig komplette in LADDER formulierte Datenstrukturen, für die Abbildung in In-Memory-Strukturen und Animationen sind Plugins zuständig.
- LADDER wird so erweitert, dass auch logische Struktur und Verhalten modelliert werden können, das Plugin enthält so wenig Funktionalität wie möglich.

Aktuell ist die erste Möglichkeit realisiert. Mit dieser Strategie ist es schwierig, eine automatische, statuslose Erkennung von Datenstrukturen zu realisieren, da Algoria keine verwertbaren Informationen über den internen, geometrischen Aufbau der Datenstrukturen hat und somit nicht entscheiden kann, welche Kombination von Arrays, Listen und Trees ein paar Rechtecke, Ellipsen und Pfeile darstellen.

Die dritte Möglichkeit erscheint wenig attraktiv, da eine absehbar sehr komplexe DSL entwickelt werden müsste, die in einer .NET-Sprache geschriebene Plugins doch nicht ersetzen kann.

Als einzige sinnvolle Lösung bleibt somit die Erweiterung von LADDER, so dass auch komplexere Symbol-Gruppen einfach erkannt werden können. Gleichzeitig muss Algoria Mehrdeutigkeiten bei der Erkennung auflösen können, bzw. dem Benutzer Hilfen zur manuellen Auflösung bieten.

Schliesslich bleibt noch die konkrete Realisation von kombinierten Datenstrukturen. Da die Plugins keine generischen Datenstrukturen implementieren können, ist es schwierig und wenig intuitiv aus vorhandenen Datenstrukturen automatisch neue zusammenzusetzen.[8] Es scheint daher angebracht, für solche kombinierten Datenstrukturen ebenfalls eigene Plugins anzulegen. Wieviel Funktionalität wiederverwendet wird, kann dem Entwickler überlassen werden.

[8] Das Array beispielsweise ist aus verschiedenen Gründen fix als `int[]` bzw. `AnimIntArray` realisiert, dieses mittels Vererbung oder Komposition als Array für eine Hashmap zu verwenden ist nicht ohne weiteres möglich.

3.1 Erweiterung von LADDER

Bisher (Abbildung 3.1) unterstützt LADDER optionale Komponenten und Wiederholungen nur indirekt über abstrakte Symbole und Vererbung bzw. Rekursion. Eine Liste (ohne Link am Ende) könnte beispielsweise wie in Listing 3.1 definiert werden. In Listing 3.2 ist dieselbe Liste (mit optionalem Link) in neuem, erweitertem LADDER (Abbildung 3.2) aufgeführt. Ein Array kann sehr einfach wie in Listing 3.3 beschrieben werden, etwas komplexer ist die HashMap in Listing 3.4.

Die Semantik der Produktionen ist natürlich etwas komplizierter als im ursprünglichen LADDER. Komponenten können nicht mehr nur einzelne Symbole, sondern auch Gruppen sein, die wiederum ähnlich wie Symbole aufgebaut sind (die Attribute sind für zukünftige Anwendungen reserviert[9]).

Constraints besitzen einen Sichtbarkeitsbereich, ist ein Constraint innerhalb einer Gruppe definiert so bezieht er sich auf die entsprechenden Elemente innerhalb der Gruppe, ist er ausserhalb definiert bezieht er sich gruppenübergreifend auf deren Elemente[10]. Ausserdem können Elemente, die auf ein bestimmtes Property-Muster passen, gesammelt werden, indem das Property in geschweifte Klammern gesetzt wird; der Ausdruck in eckigen Klammern selektiert von dieser Gruppe eine bestimmte Wiederholung oder ein ausgezeichnetes Element (z.B. `first` oder `last`, möglicherweise auch `everyOther`, d.h. jedes zweite).

Ein Quantifier schliesslich hat dieselbe Bedeutung wie in Regulären Ausdrücken, d.h. `?` macht die Komponente optional, `+` bedeutet sie muss mindestens einmal vorkommen, `*` steht für beliebig viele Vorkommnisse und die Zahlen in Klammern stehen für `{von, bis}`, eine genaue bzw. minimal oder maximal eine bestimmte Anzahl.

3.2 Natürlicheres Skizzieren

Als letzte Massnahme zur Attraktivitätssteigerung wurde schliesslich ein Proof-of-Concept eines Features implementiert, das es zukünftig erlauben soll, z.B. Arrays als zwei horizontale Linien und beliebig viele vertikale ‚Feldtrenner' zu skizzieren, statt wie bisher einzelne Kästchen zu zeichnen. Dazu werden die skizzierten Linien nach der Erkennung durch den Componizer miteinander geschnitten und als so zerteilte, eigenständige Liniensegmente dem Symbolizer übergeben. In der aktuellen Version müssen die vertikalen Feldtrenner noch doppelt gezeichnet werden (für das jeweils benachbarte Feld), mit der vorgeschlagenen Erweiterung (Abschnitt 3.1) von LADDER und einer entsprechenden Symbolizer-Implementierung, wäre auch das hinfällig. Für das Problem ‚ungewollter' Schnitte bietet sich eine alternative Formulierung der Symboldefinitionen in LADDER an, d.h. die Symbole müssen ‚planar' beschrieben werden, beim Skizzieren entstehende Schnitte müssen bereits im Vorfeld berücksichtigt werden. Ob und ab welcher Distanz Linien, die sich bloss ‚fast' schneiden, getrennt werden sollen, ist eine offene Frage. Um die Symbolizer-Performance zu verbessern, werden sehr kleine abgeschnittene Teile (z.B. Feldtrenner-Schnipsel oberhalb der oberen horizontalen Linie beim Array) verworfen. Trotzdem dauert die Erkennung viel zu lange, behelfsmässig kann die automatische Umwandlung aktiviert

[9] Denkbar wäre z.B. ein `@reusable`, das einen vertikalen Arrayfeld-Trennstrich für seinen linken und rechten Nachbarn verfügbar macht.

[10] Ein `parallel`-Constraint beispielsweise sorgt innerhalb der Gruppe dafür, dass die entsprechenden Elemente einer jeden Gruppe für sich genommen parallel sind, steht er ausserhalb so müssen alle entsprechenden Elemente parallel sein, unabhängig davon, in welcher Gruppe sie sich befinden.

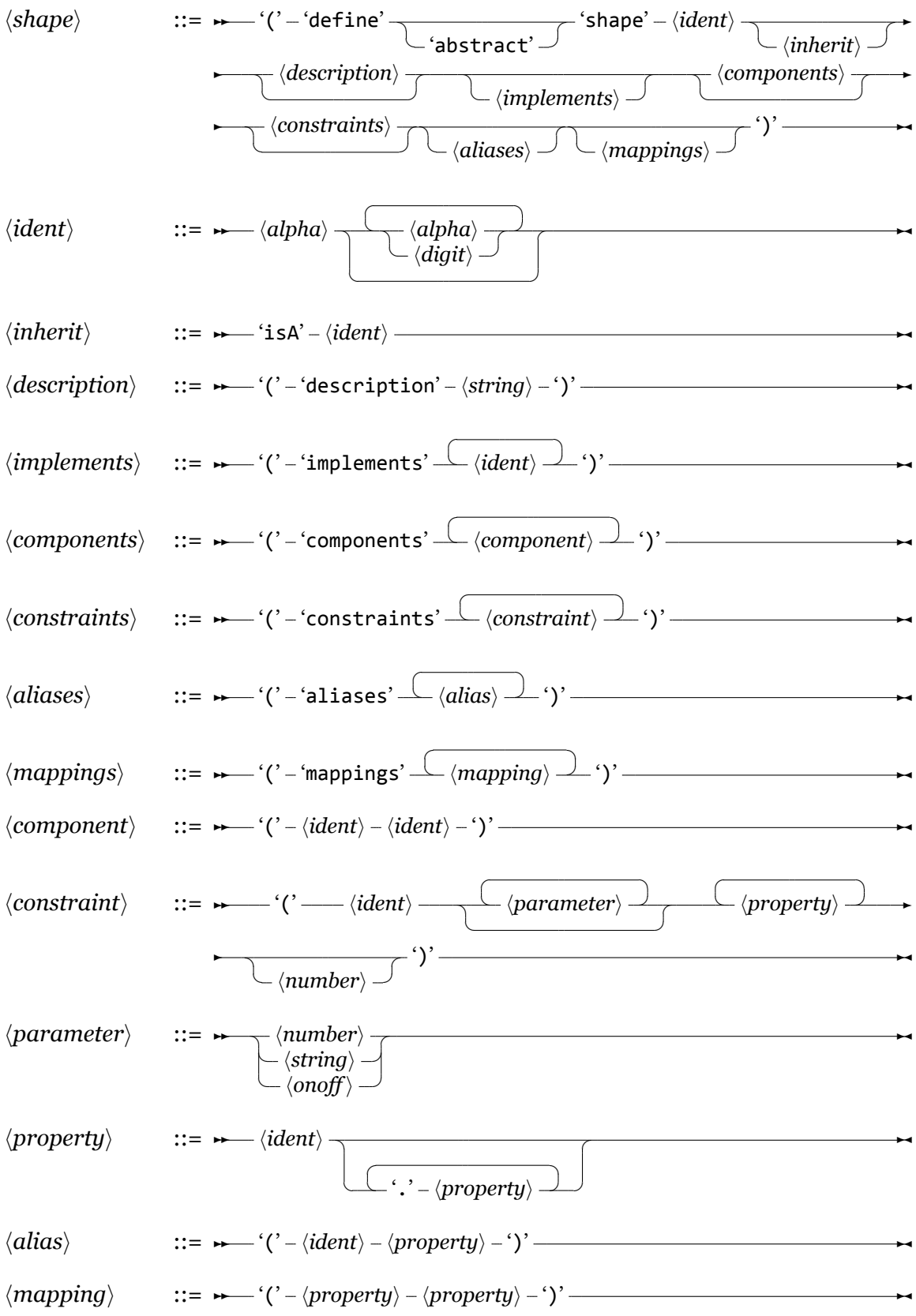


Abbildung 3.1: Ursprüngliche LADDER Grammatik

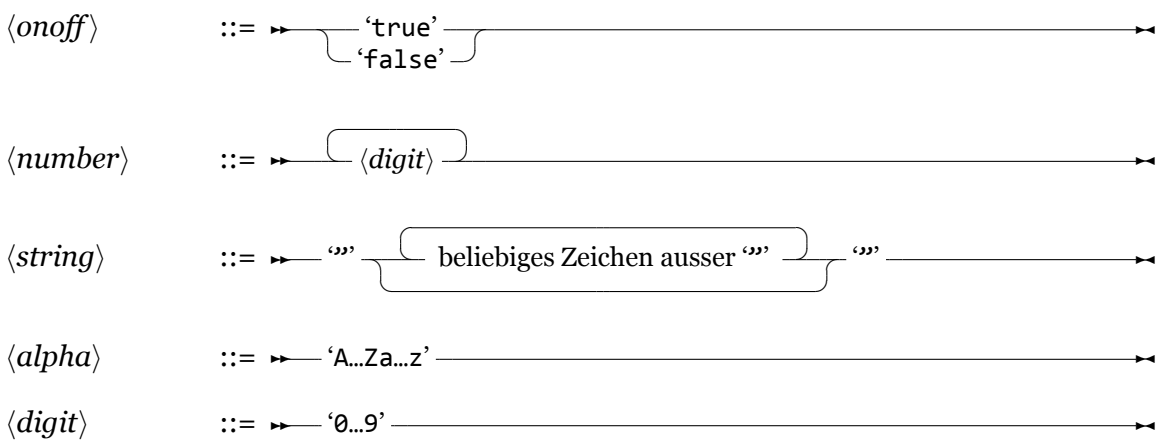


Abbildung 3.1 (Fortsetzung): Ursprüngliche LADDER Grammatik

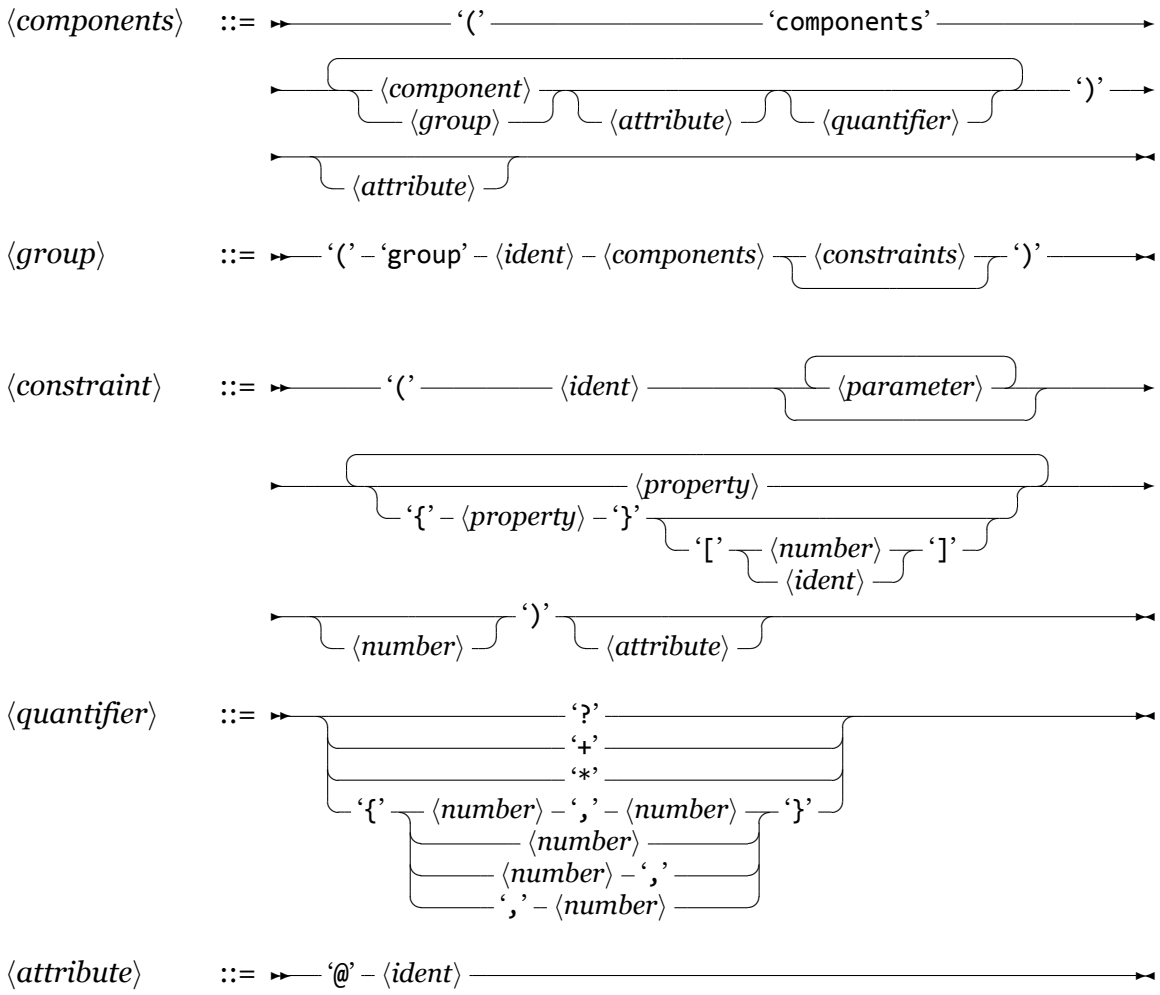


Abbildung 3.2: Vorschlag für neue LADDER Grammatik (Auszug)

```
(define abstract shape List
  (description ”Marker for lists”)
)

(define shape ListNode isA Rectangle
  (implements List)
)

(define shape ClosedList
  (implements List)
  (components
    (ListElement head)
    (Link link)
    (ListElement tail)
  )
  (constraints /* Hier unwichtig */)
)
```

Listing 3.1: Liste in ursprünglichem LADDER

```
(define shape List
  (components
    (group LinkedNode
      (components
        (Rectangle node)
        (Link link)?
      )
      (constraints
        (contains node link.From)
      )
    )+
  )
  (constraints
    (contains node link.To)
    /*
      Falls die Gruppe LinkedNode mehr als einmal gematcht wird,
      so muss der Link auf eine weitere Gruppe zeigen.
    */
  )
)
```

Listing 3.2: Liste in erweitertem LADDER

```
(define shape Array
  (components
    (Rectangle field)+
  )
  (constraints
    (collinear {field})
  )
)
```

Listing 3.3: Array in erweitertem LADDER

```
(define shape HashMap
  (components
    (group Bucket
      (components
        (Rectangle key)
        (Link el)?
        (group Entry
          (components
            (Rectangle value)
            (Link l)?
          )
          (constraints
            (contains value l.From)
          )
        )*
      )
      (constraints
        (contains key el.From)
        (contains value el.To)
        (contains value l.To)
        (collinear {Entry})
      )
    )+
  )
  (constraints
    (collinear {Bucket})
  )
)
```

Listing 3.4: HashMap in erweitertem LADDER

werden, so dass Symbole direkt nach jedem Schnitt erkannt und als z.B. Array-Felder verwendet werden, was für den Benutzer aber verwirrend sein kann.

Durch die Implementation der Linienschnitte ist ein weiterer Fehler in Algoria aufgedeckt worden, der sich bemerkbar machen kann, wenn mehrere Datenstrukturen beinahe gleichzeitig erzeugt werden.[11] Der betreffende Code-Abschnitt wurde seinerzeit eingefügt um einen Spezialfall in der Array-Implementierung abzufangen, das Konzept aber so generalisiert, dass es für alle Datenstrukturen gilt. In der verbleibenden Zeit konnte dieser Fehler nicht mehr behoben werden. Sinnvoll wäre die Implementation des Arrays so zu ändern, dass der betreffende Codeabschnitt im `AlgoriaCore` stark vereinfacht werden kann.

[11] Eine Race zwischen UI- und Background-Thread mit `NullReferenceException` in `AlgoriaEngine.TryMerge()`.

4 Fazit

Die vorliegende Arbeit wurde als äusserst zwiespältig erlebt. Einerseits konnten doch einige, seit Längerem auf ihre Behebung wartende, Probleme gelöst werden, andererseits war und ist die langwierige Fehlersuche sehr entmutigend und liess den Autor mehr als einmal an seinen Fähigkeiten als Software-Entwickler zweifeln.

Der Vorsatz, Baustellen abzuschliessen und möglichst keine neuen zu eröffnen, konnte nicht eingehalten werden. Zu uneben sind die Übergänge von altem und neuem Strassenbelag, zu verlockend interessante Abzweigungen in Nebenstrassen, die sich nur allzu oft als Sackgassen herausstellen. Vom in der Projektklärung geplanten Arbeitsvorrat konnte somit vieles erledigt werden, das Titel-Thema wurde jedoch – mit Ansage – verfehlt. Grund dafür ist wohl die Tatsache, zuviel Zeit in die Behebung (vermeintlicher) Fehler und Unzulänglichkeiten gesteckt zu haben, was einerseits natürlich durchaus einen positiven Effekt auf den Code hat, andererseits ein subjektiv schlechtes Nutzen-Aufwand-Verhältnis aufweist. Die richtige Mischung von Detailversessenheit und grosszügigen Würfen zu finden, ist in diesem Projekt nicht gelungen.

Ein Beispiel dafür ist die Liniensegment-Schnitt-Funktion. Anstatt mit einer naiven, quadratisch-langsamen Implementation zu starten liegt nun eine Bentley-Ottmann Sweep-Line Entwicklung vor (inklusive eigenem AA-Tree), dafür ist die Integration in Algoria noch nicht fertig bzw. fehlerhaft. Dabei ist das Laufzeitverhalten bei den zu erwartenden Segmentzahlen sowieso gutartig und im Vergleich zum restlichen Code nebensächlich.

Abbildungsverzeichnis

Listings

Fachhochschule Nordwestschweiz
Hochschule für Technik

Ehrlichkeitserklärung

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

Unterbözberg, März 2012

Raphael Schweizer