

Diplomarbeit

**Thema: Entwurf einer XML-Sprache
zur programmiersprachenunabhängigen
Formulierung von Algorithmen und einer
Transformationskomponente für Java und C++**

vorgelegt von: Frank Seidel (frank@f-seidel.de)

...

Datum: 10. September 2004

Die Urheberrechte und Copyrights dieser Arbeit liegen bei dem Autor (Frank Seidel). Bei Interesse zur Verwertung oder Verbreitung des Werkes bittet der Verfasser um vorherige Kontaktaufnahme.

Inhaltsverzeichnis

1	Einleitung	1
I	Theorieteil	3
2	XML und Algorithmen	4
2.1	Warum XML?	5
2.2	XSLT als Programmiersprache	5
2.3	MathML für mathematische Ausdrücke	6
3	XAlgo	7
3.1	Motivation	7
3.2	Entwurfsziele	8
3.3	Formale Eigenschaften	8
4	Bezug zu Syntaxbäumen	9
5	Vorzüge und Nachteile von XAlgo	12
5.1	Vorteile	12
5.1.1	Formaler (als) Pseudocode	12
5.1.2	Leichte Weiterverarbeitbarkeit	13
5.1.3	Leicht erweiterbar	14
5.1.4	Kombinierbarkeit mit anderen XML-Anwendungen	14
5.2	Nachteile	14
5.2.1	XML-Vorkenntnisse	14
5.2.2	Wortreich	14
5.2.3	Unpräziser als Programmiersprachen	15
5.2.4	Zusätzlicher Lernaufwand	16
5.3	Fazit	16
6	Dokumentation mit „Literate Programming“	17
6.1	Literate Programming	17
6.2	Vergleich zu üblichen Dokumentationsmethoden	18

7	Rendering-Möglichkeiten	19
7.1	Detaillierte Dokumentation	19
7.2	Erweitertes Coderendering	19
8	Verwendungsmöglichkeiten	21
8.1	Validierung über RELAX NG-Schema	21
8.2	Generierung von Programmgerüsten	21
8.3	Kombination mit anderen XML-Techniken	22
8.3.1	Signierung von Algorithmen	22
8.3.2	Verschlüsselung	22
II	Praxisteil	23
9	Entwurf von XAlgo	24
9.1	Präzisierung des Sprachentwurfs	24
9.1.1	Das Typsystem	24
9.1.2	Bezeichner	25
9.1.3	Sichtbarkeit und Lebensdauer	26
9.1.4	Subalgorithmen	26
9.1.5	Klassen in XAlgo	26
9.1.6	Prozedurale Elemente	27
9.1.7	Elemente mit Nebenwirkungen	27
9.1.8	Dokumentationselemente	27
9.2	Festlegung über RELAX NG-Schema	28
9.2.1	Besondere Herausforderungen	28
9.2.2	Zwei Versionen: Restriktiv oder offen	29
10	Beschreibung von XAlgo	30
10.1	Grundaufbau von XAlgo-Dokumenten	30
10.2	Elemente für Programmdeklarationen	31
10.2.1	Namensdeklaration	32
10.2.2	Parameter des Algorithmus	32
10.2.3	Klassendeklarationen	34
10.2.4	Konstanten	34
10.2.5	Variablen	34
10.2.6	Klasseninstanzen	35
10.2.7	Rückgabewert	35
10.2.8	Typbeschreibungen	36
10.3	Beschreibung der prozeduralen Elemente	38
10.3.1	Null-Anweisung	39
10.3.2	Sequence-Struktur	39
10.3.3	Schleifen	40
10.3.4	Bedingte Ausführung	40

10.3.5	Zuweisungen	44
10.3.6	Subalgorithmen-Aufruf	44
10.3.7	Rückkehranweisung	45
10.3.8	Systemanweisungen	46
10.4	Ausdrücke	48
10.4.1	Mathematische Ausdrücke	48
10.4.2	boolesche Ausdrücke	49
10.4.3	Pfade zu Klassenelementen	49
10.4.4	Text	51
10.4.5	Arrayelemente	51
10.4.6	Arraylitterale	52
10.4.7	Sonstige	52
10.5	Dokumentationselemente	53
10.5.1	Metainformationen	53
10.5.2	Entwicklungs-Dokumentation	55
10.5.3	Erklärungskomponente	56
10.5.4	Einfache Kommentare	57
11	Umsetzung der Transformationskomponenten	58
11.1	XSLT als Transformationssprache	58
11.2	Besondere Problemstellungen	59
11.2.1	Namensumsetzung für Bezeichner	59
11.2.2	MathML-Prozessor	59
11.2.3	Abbildung von Namen auf Reihenfolgen	60
11.3	Stylesheet für Java	60
11.3.1	Besondere Entwicklungshürden	60
11.3.2	Erreichtes Ergebnis	61
11.4	Stylesheet für C++	61
11.4.1	Besondere Entwicklungshürden	61
11.4.2	Erreichtes Ergebnis	61
12	Beispielalgorithmen	62
12.1	Zähler	62
12.2	Quicksort	65
12.3	Klassendemo	69
13	Benutzung von Schema und Stylesheet	73
13.1	RELAX NG-Schemas für XAlgo	73
13.1.1	Jing	74
13.1.2	MSV	75
13.2	Transformationskomponenten	76
13.2.1	XAlgo nach Java	77
13.2.2	XAlgo nach C++	77

13.3	Beautiflier	78
13.4	Beispiel	79
13.4.1	Validierung des XAlgo-Dokumentes	79
13.4.2	Übersetzung in Java	80
13.4.3	Optionale Umformatierung	80
13.4.4	Compilierung mit javac	80
14	Offene Ergänzungen	81
14.1	Schemas	81
14.2	Transformationskomponenten	81
14.2.1	Typüberprüfung	81
14.2.2	Vollständige MathML-Unterstützung	82
14.2.3	Kommandozeilenparameter	82
14.2.4	file -Attribut bei Ein- und Ausgaben	82
14.2.5	Code-Säuberung	82
15	Schlusswort	83
A	Sprachreferenz von XAlgo	85
A.1	Grundelemente	85
A.2	Deklarative Elemente	86
A.3	Prozedurale Elemente	88
A.4	Elemente für Ausdrücke	91
A.5	Dokumentationselemente	92
B	Datenträger	94
B.1	Beschreibung	94
B.2	Medium	95
	Abbildungsverzeichnis	96
	Tabellenverzeichnis	97
	Listings	98
	Abkürzungsverzeichnis	100
	Literaturverzeichnis	101
	Index	103

Kapitel 1

Einleitung

„Some languages are designed to solve a problem; others are designed to prove a point.“¹

(Dennis Ritchie² [[Withopf 1999](#)])

Dieses Zitat gibt einen wichtigen Hinweis darauf, warum die Zahl der Programmiersprachen auf eine inzwischen unüberschaubare Zahl gestiegen ist. Die im Zuge dieser Arbeit erstellte Sprache versteht sich jedoch nicht nur als Machbarkeitsbeweis, sondern vielmehr als ein Vorschlag, eine aus meiner Sicht immer noch bestehende Lücke in der Gruppe der XML-Sprachen zu schließen. So entstand eine XML-basierte Sprache zur Formulierung von Algorithmen, samt einem vollständigen Schema zur Überprüfung von in dieser Sprache verfassten Dokumenten, sowie Compilern für die Zielsprachen Java und C++. XML ist eine Empfehlung des WWW Consortiums (kurz W3C³) für eine ganze Familie von so genannten Auszeichnungssprachen. Compiler sind dabei im Grunde Programme, die ein in der Quellsprache geschriebenes Programm in ein äquivalentes Programm einer anderen Zielsprache übersetzen [[Aho et al. 1999](#), Seite 1].

Die entworfene Sprache bedient dabei im wesentlichen zwei Punkte. Zum ersten ist sie „programmiersprachenunabhängig“, was bedeuten soll, dass die hierin formulierten Algorithmen auf einer höheren Abstraktionsebene beschrieben werden. Durch diese allgemeinere Beschreibung fehlt ihnen der Detailgrad von normalen Programmiersprachen. Eine Klassifizierung als formalisierte Pseudocode-Sprache wäre daher angemessen. Der zweite und wichtigste Aspekt dieser Sprache ist der Einsatz von XML als Syntaxgrundlage. Als – bisher noch inoffizielles – Mitglied der XML-Familie profitiert sie aus einer noch nicht absehbaren Anzahl von Kombinations- und Ergänzungsmöglichkeiten.

¹Übersetzt vom Verfasser der Arbeit: Manche Sprachen sind dafür entworfen, ein Problem zu lösen; andere sind dafür entworfen, etwas zu beweisen.

²„... ‘Miterfinder’ der Sprache C ...“ [[Withopf 1999](#)]

³für Details siehe Abkürzungsverzeichnis

Nach meinen Recherchen – sowohl in der Fachliteratur als auch im Internet – existierte zum Zeitpunkt der Erstellung dieser Arbeit keine andere XML-Sprache, um auf einfache und direkte Weise Algorithmen beschreiben zu können. Auch der schon erwähnte Pseudocode ist bisher nicht standardisiert [uni-protokolle.de]. Das Ziel war es also, einen XML-Sprachvorschlag zu entwickeln, der genau diese Lücke füllt.

Das Thema wählte ich, weil mich die Möglichkeiten, die einem XML-Dokumente bieten, schon lange Zeit während meines Informatikstudiums fasziniert haben. So hatte ich beispielsweise auch schon in dem Fach Contentsicherheit die Gelegenheit, mich intensiv mit digitalen XML Signaturen zu beschäftigen. Dafür habe ich damals sowohl ein Referat als auch das erste Kommandozeilenprogramm (in Java) für die damalige Version der Apache-Programmbibliothek erstellt. Doch bereits seit meinem ersten intensiven Kontakt mit dieser Sprachfamilie – in einem XML-Blockkurs der Hochschule – ist mir das Fehlen einer Beschreibungssprache aufgefallen, wie sie diese Arbeit behandelt.

Grundsätzlich sind für das Verständnis des vorliegenden Werkes lediglich Kenntnisse über gängige Begriffe der Informatik – wie z. B. Bäume, Hierarchien, Internet etc. – nötig. Darüber hinaus sind Erfahrungen auf den Gebieten XML- und Programmiersprachen hilfreich, aber nicht zwingend erforderlich.

Auf die Untersuchung der exakten Analysemöglichkeiten durch diese Sprachform, sowie weitere Kombinationsmöglichkeiten mit anderen XML-Sprachen und auch auf das Rendering musste aus zeitlichen Gründen verzichtet werden.

Am Anfang der Arbeit standen zunächst umfangreiche Recherchen in der Fachbibliothek der Fachhochschule und intensive Suchen im Internet. Beim anschließenden Entwurf der Sprache orientierte ich mich stark an den beiden Programmiersprachen Java und Pascal. Die Sprache selbst habe ich dann in einer noch sehr neuen Schemasprache präzisiert, mit deren Hilfe man selbsterstellte Dokumente auf ihre strukturelle Richtigkeit prüfen lassen kann. Als der Sprachumfang einen stabilen Stand erreicht hatte, konnte ich damit beginnen, zunächst den Compiler bzw. die Transformationskomponente für die Zielsprache Java zu formulieren. Nach der Überwindung zahlreicher Schwierigkeiten war es mir schließlich doch möglich, diese Komponente mit einer ebenfalls in XML formulierten Transformationssprache zu vollenden. Auch für C++ war es mir nach weiteren Ergänzungen möglich, einen funktionierenden Compiler zu erstellen.

Für die besondere Unterstützung möchte ich mich bei meinem Betreuer Prof. Dr. Christian Schiedermeier bedanken.

Teil I

Theorieteil

Kapitel 2

XML und Algorithmen

XML¹ ist in den letzten Jahren zu einer immer bedeutenderen Kerntechnologie der Informatik herangewachsen. Viele Internetseiten, Ver- und Bearbeitungsprogramme und XML-basierte Auszeichnungssprachen für Internet-, Firmen- und Privatanwendungen zeugen von der breiten Akzeptanz dieser Sprachempfehlung des W3C².

Bei XML handelt es sich um ein Metaformat, das vorgibt, wie konkrete XML-Sprachen³ aufgebaut sein dürfen. Trotz des Namens ist XML selbst keine eigene Auszeichnungssprache [Ray 2001, Seite 2]. Wie auch der Vorläufer SGML setzt XML auf Elemente und Attribute, die in einem Dokument eine eindeutige und hierarchische Struktur bilden. SGML fand zuvor nur sehr eingeschränkt Anwendung, da es zwar viel weniger Beschränkungen unterlag, jedoch die Software zur Verarbeitung auch viel komplizierter und teurer war [Ray 2001, Seite 11].

Aufgrund der inhärenten hierarchischen Struktur von XML ist es besonders gut geeignet, strukturierte oder auch strukturierbare Daten zu beschreiben. Hier findet es vielfach Anwendung, z. B. auf Webseiten, die aus XML-Dokumenten mit XSLT-Stylesheets die in Webbrowsern darstellbaren XHTML-Seiten generieren.

In welchem Zusammenhang nun aber XML und Algorithmen stehen, ergibt sich hieraus bisher noch nicht eindeutig. Algorithmen werden häufig entweder direkt in Programmiersprachen, in ungenauem Pseudocode oder Mischformen beider festgehalten und auch in dieser Form zur Ausbildung und in Büchern verwendet (z. B. [Güting 1992] mit Modula-2). Das Stichwort „strukturierte Programmierung“ bildet nun die entscheidende Brücke zu XML. Seit den 70er Jahren versucht man – weg vom so genannten „Spaghetti-Code“ – hin zu strukturierten, hierarchisch

¹siehe <http://www.w3.org/xml/> und im Abkürzungsverzeichnis

²siehe <http://www.w3.org> und im Abkürzungsverzeichnis

³häufig auch XML-Anwendung, XML-Vokabular oder allgemeiner Auszeichnungs- oder Markup-Sprache

aufgebauten Programmstrukturen zu kommen, angefangen bei N. Wirth's Pascal bis hin zu fast allen modernen Programmiersprachen (nach [Schiedermeier 1996]). Nun liegt die Überlegung nahe, auch hierarchische Programm- bzw. Algorithmenstrukturen in einem eigenen XML-Vokabular zu erfassen.

2.1 Warum XML?

Um strukturierte Programme zu beschreiben, gibt es schon heute eine fast unüberschaubare Anzahl von Programmiersprachen, die genau dies leisten, könnte man argumentieren. Dies ist sicher richtig, doch spricht man bei XML nicht umsonst oft von der XML-Familie [W3C XML 2003, Punkt 5].

Die große Vielfalt an XML-Sprachen – standardisierte wie selbstkreierte – lassen sich nahtlos und konfliktfrei (über XML-Namensräume) miteinander kombinieren. Hierdurch erlangt man fast beliebig erweiterbare Möglichkeiten bei der Beschreibung von Daten und nun auch von Algorithmen.

Auch die zahlreichen und zum Teil sogar freien Verarbeitungsprogramme für XML-Dokumente können somit zur Erstellung, Analyse oder Weiterverarbeitung von Algorithmen genutzt werden.

Aufgrund dieser Rahmenbedingungen scheint es schon fast verwunderlich, dass es bisher nach den durchgeführten Recherchen keinerlei andere Entwicklung in diese Richtung gegeben hat.

2.2 XSLT als Programmiersprache

Mit XSLT existiert bereits eine XML-Ausprägung, die man als vollwertige (touringmächtige) Programmiersprache bezeichnen darf. Jedoch liegt bei dieser Sprache das Hauptaugenmerk auf der Transformation von XML-Dokumenten, wie der Name „Extensible Stylesheet Language for Transformation“ schon vermuten lässt.

Zwar werden viele auch in gängigen Programmiersprachen übliche Konstrukte angeboten, jedoch sind die Einschränkungen sehr gravierend. Zum Teil sind selbst einfachste Algorithmen nur schwer zu formulieren und später für Menschen auch nur wieder schwer lesbar. Zum Beispiel lassen sich schon Schleifen mit einer errechneten Wiederholungsanzahl nur mit Tricks realisieren. Dies entspricht den Anforderung an eine XML-Sprache zur Formulierung von Algorithmen natürlich in keinsten Weise.

Aber selbst wenn XSLT für die Erfassung von strukturierten Algorithmen nicht besonders geeignet scheint, so wird es bei dieser Arbeit dennoch als Programmiersprache für die Compiler eingesetzt.

2.3 MathML für mathematische Ausdrücke

MathML stellt eine weitere XML-Sprache dar, die bei der vorliegenden Arbeit ebenfalls eine ganz besondere Rolle spielt. Mit dieser Sprache können beliebige mathematische Ausdrücke formuliert werden. Dabei setzt sie auf zwei Modelle, um dies zu ermöglichen. Mit Hilfe des Präsentations-Modells wird beschrieben, wie die Darstellung erfolgen soll. Mit dem Content-Modell hingegen wird der mathematische Inhalt festgelegt.

Verständlicherweise kommt für das verfolgte Ziel hier natürlich nur das Content-Modell in Betracht, da für die zu formulierenden Algorithmen nicht die Erscheinung, sondern zunächst nur die Funktion beziehungsweise der Inhalt entscheidend ist.

Der Einsatz von MathML ist dabei nur die erste und nächstliegende Kombination mit Mitgliedern der XML-Familie. Zum Beispiel wäre durchaus auch der Einsatz von XML-Anwendungen wie DocBook oder XSL-FO für die Formatierung von Programmausgaben denkbar.

Kapitel 3

XAlgo - zwischen Pseudocode und Programmiersprache

Die im Zuge dieser Arbeit geschaffene Sprache wurde „XAlgo“¹ benannt. XAlgo soll dabei als offene Sprachempfehlung aufgefasst und frei verwendet werden. Diese neue Sprache ist zwischen üblichem Pseudocode und gängigen prozeduralen Programmiersprachen angesiedelt. Sie soll also keineswegs in Konkurrenz zu Sprachen wie Java oder Perl treten, sondern vielmehr als ein formalisierter Pseudocode aufgefasst werden, mit dessen Hilfe man direkt lauffähige Programmgerüste in verschiedensten Programmiersprachen erzeugen kann. Bei dieser Arbeit wurden hierzu zwei Transformationskomponenten beziehungsweise Compiler erstellt, die XAlgo in lauffähige Java- und C++ Programme übersetzen können.

3.1 Motivation

In der XML-Familie gibt es inzwischen eine fast unüberschaubare Anzahl der verschiedensten XML-Sprachen. Das Spektrum reicht von relativ einfachen Formatbeschreibungen wie XHTML bis hin zu biomedizinischen Daten in BIOML². Die große Flexibilität von XML einerseits, sowie der strenge hierarchische Aufbau andererseits, lassen es wie geschaffen erscheinen, perfekt strukturierte Algorithmen (z. B. aus einem Struktogramm) direkt mit einem passenden XML-Vokabular zu erfassen.

Steht man z. B. vor der Aufgabe, einen gängigen Algorithmus (z. B. Quicksort) in einer bestimmten (imperativen bzw. auch objektorientierten) Programmiersprache zu implementieren, muss man in aller Regel von einer Vorlage entweder Pseudocode konkretisieren oder den Algorithmus aus einer anderen Sprache in

¹X in Anspielung auf XML, Algo für Algorithmen; gesprochen: X-Algo

²siehe „<http://xml.coverpages.org/bioml.html>“

die gewünschte Zielsprache manuell umsetzen. Beide Varianten sind mögliche Fehlerquellen für zu schreibende Programme. Liegt hingegen der gewünschte Algorithmus in XAlgo, sowie eine Transformationskomponenten für die verwendete Programmiersprache vor, kann man sich ein lauffähiges Programmgerüst erzeugen lassen.

Beim Verfassen eines Buches über Algorithmen (z. B. mittels der XML-Sprache DocBook) wäre es denkbar das Buch für mehrere Zielsprachen aus dem gleichen Datenmaterial zu generieren, wenn man entsprechende Transformationskomponenten für XAlgo und die gewünschten Sprachen zur Hand hat.

Es ließen sich viele derartige Beispiele konstruieren, bei denen jedesmal die Verwendung einer programmiersprachenunabhängigen Algorithmenformulierung von zentraler Bedeutung ist. XML ist hierfür eine gute Grundlage.

3.2 Entwurfsziele

XAlgo sollte möglichst einfach aufgebaut sein, sich an gängigen Programmiersprachen orientieren und leicht erlernbar sein. Außerdem stellt die Dokumentationsfähigkeit ein besonderes Ziel dieser Sprache dar, was jedoch noch in Kapitel 6 (auf Seite 17) genauer beleuchtet wird.

3.3 Formale Eigenschaften

XAlgo ist eine prozedurale bzw. imperative Programmiersprache mit einem einfachen objektorientiertem Ansatz. Da es sich weiterhin um eine XML-Anwendung handelt, besitzt die Sprache im eigentlichen Sinne keine eigene Syntax und Grammatik, sondern setzt auf XML-Vorgaben, wodurch quasi direkt die Semantik mit XML-Tags beschrieben wird.

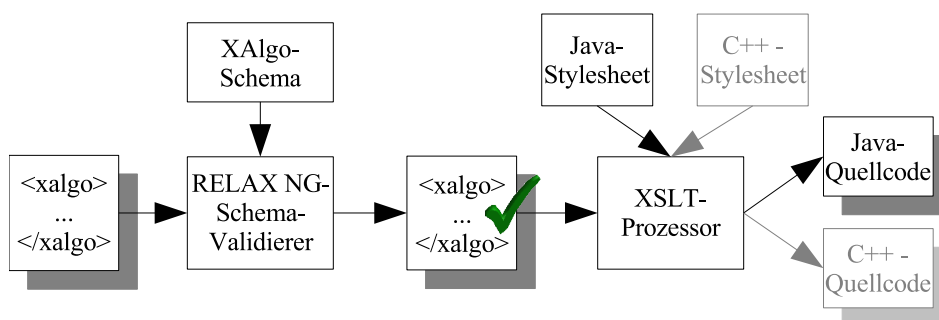


Abbildung 3.1: Grobes Verarbeitungsschema für XAlgo-Dokumente

Kapitel 4

Bezug zu Syntaxbäumen

Compiler – also Übersetzungsprogramme – für heutige prozedurale Programmiersprachen sind in aller Regel aus mehreren aufeinander aufbauenden Teilen bzw. Phasen aufgebaut ([Aho et al. 1999], [Pingel 2003]).

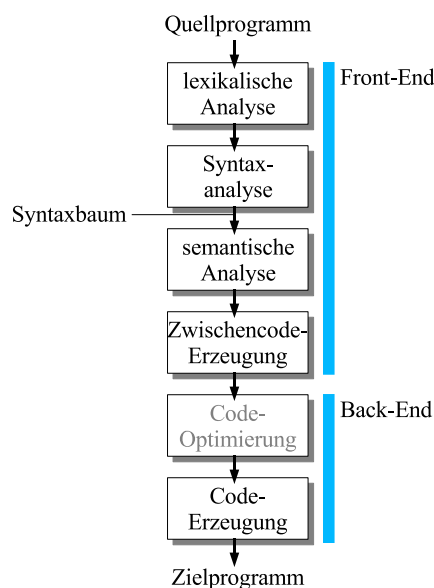


Abbildung 4.1: Phasen eines Compilers (Grafik nach [Aho et al. 1999, Seite 12])

Die Aufgabe der lexikalischen Analyse beziehungsweise des so genannten Scanners ist es, aus den Textzeichen des Quellprogramms die Symbole (lexikalische Objekte wie Bezeichner, Operatoren, Leerräume etc.) zu erkennen und an die nächste Stufe weiterzureichen. Diese Symbole fasst die Syntaxanalyse (oder auch Parser)

zu grammatikalischen Sätzen in Form von abstrakten Syntaxbäumen¹ zusammen [Aho et al. 1999, Seite 9]. In der semantischen Analyse werden die Syntaxbäume auf Zusammenhangsfehler geprüft und Typ-Informationen gesammelt und ausgewertet. Hieraus wird nun – nach einer eventuellen Zwischencodeerzeugung – im so genannten Back-End des Compilers der Code für die Zielsprache erzeugt (siehe dazu auch Abbildung 4.1 auf Seite 9).

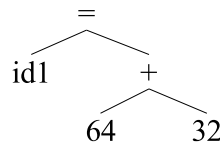


Abbildung 4.2: abstrakter Syntaxbaum einer Zuweisung mit vorheriger Addition

```

<assign>
  <source><math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply><plus /><cn>64</cn><cn>32</cn>
    </apply>
  </math>
</source>
<target>id1</target>
</assign>
  
```

Listing 4.1: XAlgo-Segment einer Zuweisung passend zu Abbildung 4.2

Aufgrund des streng hierarchischen Aufbaus von XML-Dokumenten gibt ein in XML formulierter Algorithmus quasi eine schriftliche Form eines Syntaxbaumes wieder. Zunächst entspricht dieser noch einem so genannten Parse-Baum, der noch viele redundante Informationen enthält (z. B. Vorrangregeln bei Berechnungen, die aber im Baum schon über die Struktur gegeben sind), doch ist dieser in den kompakteren „abstrakten Syntaxbaum“ überführbar [Aho et al. 1999, Seite 60ff][Pingel 2003, Seite 2]. Auch bisherige, datenorientierte XML-Dokumente werden zur Analyse und Verarbeitung häufig über ihre so genannten „DOM-Bäume“² angesprochen, die als eine Ausprägung von Syntaxbäumen aufgefasst werden dürfen. Auch im XAlgo-Segment aus Listing 4.1 ist die Ähnlichkeit mit dem zugehörigen abstrakten Syntaxbaum aus Abbildung 4.2 gut erkennbar.

¹verdichtete Form der Parse-Bäume (oft auch konkreter Syntaxbaum)

²Document Object Model; streng genommen eigentlich nur eine API für Baumzugriffe [Schiedermeier 2004, Seite 17]

Auf abstrakten Syntaxbäumen (oft kurz ASTs) sind eine Vielzahl von Analysen möglich [Pingel 2003, Seite 5]. Dies wird auch in Compilern zur Überprüfung der Semantik eines Programmes genutzt. ASTs enthalten eine zusammengefasste aber vollständige Repräsentation eines Programmes in einer Baumstruktur. Zur Untersuchung und Weiterverarbeitung von solchen Bäumen stehen eine große Zahl von Verfahren zur Verfügung, die sich je nach Anforderungen kombinieren lassen.

Es wird durch die nahe Verwandtschaft von XAlgo zu abstrakten Syntaxbäumen auch noch deutlicher, wie vorteilhaft dies bei der Realisierung und Benutzung dieser Sprache ist. Auf die Entwicklung und Verwendung von eigenen Scanner- und Parser-Programmen kann so vollständig verzichtet werden. Die Benutzung der XML-Syntax, statt einer eigenen neuen, kommt hier voll zum Zuge.

Weiterhin wird die semantische Analyse zum größten Teil – von einer Typüberprüfung abgesehen – von einem RELAX NG-Schemavalidator übernommen³. So konnte selbst für diese weitere Übersetzungsphase ein äußerst effizienter Weg gewählt werden. Auch ist dadurch die Sprache viel leichter auf künftige Erweiterungen und besondere Bedürfnisse möglicher Benutzer eingestellt.

Schon in der Vergangenheit gab es vereinzelt Projekte zur direkten Formulierung von Zwischendarstellungen, die den Syntaxbäumen entsprechen (wie z. B. IML⁴[Pingel 2003]), denen nach Auffassung des Autors jedoch deutlich die mangelnde Flexibilität gegenüber einer XML-basierten Lösung anzumerken ist.

³Im Augenblick wird der Rest noch auf die Analysen der Zielsprachencompiler übertragen

⁴InterMediate Language

Kapitel 5

Vorzüge und Nachteile von XAlgo

Viele seiner positiven Eigenschaften hat XAlgo durch seine Zugehörigkeit zur XML-Familie (wenn auch noch nicht als offizieller Bestandteil). Selbstverständlich kann aber selbst so eine vielseitige und flexible Sprache nicht nur Vorzüge besitzen. Die Vor- und Nachteile sollen deshalb nun genauer beleuchtet werden.

5.1 Vorteile

Fast alle hier erwähnten Vorzüge von XAlgo beruhen auf dem Einsatz von XML. Dennoch finden sie hier Erwähnung, weil sie trotz dieser eher ererbten Eigenschaften trotzdem von Nutzen sind.

5.1.1 Formaler (als) Pseudocode

Pseudocode wird immer noch gerne zur Vermittlung einfacher Algorithmen herangezogen. Allerdings existiert für Pseudocode kein einheitlicher Standard. Nach [uni-protokolle.de] ist Pseudocode lediglich eine halbformale, nicht standardisierte Sprache. Mit dieser kann man die Ablaufstrukturen eines Algorithmus grob durch Texte und Schlüsselwörter, die sich in der Regel an Ausdrücken höherer Programmiersprachen anlehnen, beschreiben.

Schon aufgrund der fehlenden Formalität ergibt sich, dass Pseudocode niemals zur automatisierten Verarbeitung und Analyse geeignet sein wird. XAlgo hingegen beschreibt Programmabläufe auf ähnlicher Ebene wie üblicher Pseudocode, ist jedoch formal exakt über ein RELAX NG-XML-Schema definiert. Aus diesem Grund kann man die hier geschaffene Sprache durchaus als formalisierten beziehungsweise formalen Pseudocode bezeichnen.

5.1.2 Leichte Weiterverarbeitbarkeit

Nicht nur der festgelegte Sprachformalismus, sondern auch die überaus bequemen Weiterverarbeitungsmöglichkeiten sind ein wichtiger Fürsprecher. Aufgrund der vielen existierenden XML-Sprachen und der großen Zahl von – sogar freien – XML-Verarbeitungsprogrammen können XAlgo-Dateien zu einer im Augenblick noch gar nicht vollständig überschaubaren Menge von Verarbeitungs- und Analysenformen genutzt werden.

Analysen

Die in Kapitel 4 schon erwähnten DOM-Bäume lassen sich für Analysen besonders gut verwenden. Über die hierin enthaltenen Daten könnte man z. B. eine Typüberprüfung durchführen. Aber auch Zusammenfassungen des Algorithmienablaufes lassen sich erstellen, indem man Detailschritte beziehungsweise Baumknoten entfernt. Diese Darstellungsform lässt als klassische Zwischendarstellung alle Analyseformen zu, die auch auf abstrakten Syntaxbäumen denkbar sind (siehe auch hier Kapitel 4). Jedoch ist aufgrund der XML-Basiertheit die Durchführung besonders einfach möglich.

Übersetzungen

Wie in dieser Arbeit sogar durch die Erstellung der Transformationskomponenten für Java und C++ bewiesen wird, sind XAlgo-Dokumente durchaus auch in gängige Hochsprachen überführbar (siehe Kapitel 11 auf Seite 58). Natürlich kann eine Sprache, die mehr auf der Ebene der Algorithmen als auf der Maschinenebene angesiedelt ist, nicht die implementatorischen Details heutiger Programmiersprachen nutzen. Jedoch können vollständige und lauffähige Programmgerüste generiert werden, die den Algorithmus fehlerfrei ausführen lassen.

Automatisch generierbare Dokumentation

XAlgo enthält ein System zur einfachen Beschreibung und Dokumentation jedes einzelnen Teiles eines Algorithmus (siehe auch Kapitel 6). Mit Hilfe dieser optionalen Zusatzinformationen zum Algorithmus können ausführliche Dokumentationen zu dem beschriebenen Algorithmus (z. B. in Form von HTML-Seiten oder eines PDF-Dokuments) generiert werden. Leider hätte es den zeitlichen Rahmen dieser Arbeit gesprengt, auch dies anhand einer weiteren Transformationskomponente demonstrierbar zu machen.

5.1.3 Leicht erweiterbar

Aufgrund des offenliegenden Schemas für XAlgo ist es für XML-erfahrene beziehungsweise speziell RELAX NG-erfahrene Benutzer ein leichtes, die Sprache und selbst die Transformationskomponenten um weitere Details und Fähigkeiten zu erweitern.

5.1.4 Kombinierbarkeit mit anderen XML-Anwendungen

Andere XML-Sprachen sind speziell zur Kombination mit wieder anderen XML-Anwendungen ausgelegt. So existiert beispielsweise eine Technik für digitale Signaturen (siehe auch „<http://www.w3.org/TR/xmlsig-core/>“) von XML-Dokumenten. Hiermit ist es besonders einfach, Quellcode vor Veränderungen zu schützen beziehungsweise dies zumindest erkennbar zu machen. Würde man typische digitale Signaturen auf herkömmlichen Quellcode einer gängigen Programmiersprache anwenden, würden selbst irrelevante Änderungen wie zusätzliche Leerzeichen die Authentizitätsprüfung verhindern. Aufgrund einer speziellen Umformung bei XML-Signaturen – der Kanonifizierung – ist dies dort nicht der Fall.

5.2 Nachteile

Selbstverständlich stehen nun aber den genannten Vorteilen auch eine Reihe negativer Punkte entgegen, die auch erwähnt werden sollen. Ähnlich wie bei den Vorzügen, liegen hier zumindest auch die ersten beiden Nachteile im Einsatz von XML.

5.2.1 XML-Vorkenntnisse

Leider lässt es sich bei einer so stark auf XML-basierter Sprache nicht vermeiden, dass man bei der Benutzung zumindest grundlegende Kenntnisse von XML, den gängigsten Sprachen und Softwarewerkzeugen besitzt. Da dieses Themengebiet jedoch umfangreich ist und man derartiges Wissen für andere Programmiersprachen nicht zwangsweise benötigt, ist dies sicherlich eine große Hemmschwelle für viele potentielle Nutzer dieser neuen Sprache.

5.2.2 Wortreich

Es liegt in der Natur von XML, dass Dokumente in dieser Sprache dazu neigen, sehr umfangreich zu werden [[W3C XML 2003](#), Punkt 4]. Durch diese Eigenschaft leiden vor allem größere Algorithmen in ihrer Übersichtlichkeit, wenn sie direkt in normalen Texteditoren betrachtet werden. Allerdings können moderne XML-Betrachter dies dadurch kompensieren, dass sie dem Benutzer entweder

die zugehörige Baumstruktur zum Navigieren anbieten oder zumindest niedrige Hierarchiestufen vorübergehend ausblenden können. Letztere Möglichkeit bieten sogar die beiden Internetbrowser Mozilla¹ (sowie darauf aufbauende Browser, z. B. Firefox) und Microsofts Internet Explorer (siehe Abbildungen 5.1 und 5.2).

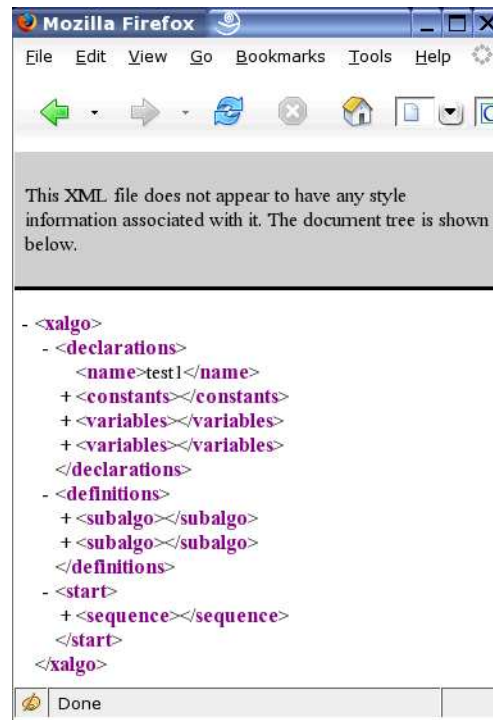


Abbildung 5.1: XAlgo-Dokument in Firefox-Browser mit „eingeklappten“ unteren Hierarchieebenen (mit „+“ gekennzeichnet)

5.2.3 Unpräziser als Programmiersprachen

XAlgo fehlen viele in Programmiersprachen übliche Mittel zum Beschreiben von implementatorischen Details, wie zum Beispiel:

- Möglichkeit zum Informationhiding (vor allem in Klassen)
- Speicherallokationen und Verwendung im Heap
- Enumerationstypen
- Typüberprüfungen zur Laufzeit
- Fehlerbehandlung und vieles mehr

¹freier Internetbrowser; siehe <http://www.mozilla.org>

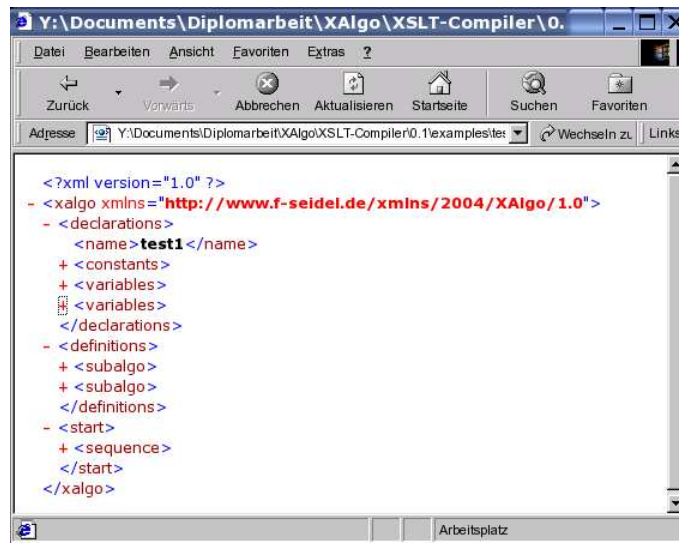


Abbildung 5.2: XAlgo-Dokument (aus Abbildung 5.1) in Microsofts Internet Explorer

5.2.4 Zusätzlicher Lernaufwand

Da es sich bei XAlgo – trotz der XML-Basiertheit – dennoch um eine neue Sprache mit eigenen Element- und Attributnamen handelt, lässt es sich nicht vermeiden, dass einem Benutzer ein zusätzlicher Lernaufwand aufgebürdet wird. Selbst wenn der Anwender vorher schon Erfahrungen mit XML- und gewöhnlichen Programmiersprachen sammeln konnte, lässt sich eine gewisse Einarbeitungszeit nicht verhindern. Zumindest die grundlegende Struktur und die wichtigsten Elementnamen von XAlgo muss man vor einem Einsatz unbedingt kennen.

5.3 Fazit

Zwei der hier genannten Nachteile hängen direkt mit der Verwendung von XML zusammen und sind somit auch bei allen anderen XML-Sprachen präsent. Das zeigt, dass diese keine zu große Hürde darstellen, nachdem dies den anderen Familienmitgliedern keinen Abbruch geleistet hat. Die geringere Präzisionsstärke von XAlgo liegt größtenteils im Zweck der Sprache begründet. Eine programmiersprachenunabhängige Sprache kann natürlich keine programmiersprachliche Details beschreiben. Auch ein zusätzlicher Lernaufwand ist mit jeder neuen – sowohl sehr schlechten als auch nützlichen – Sprache verbunden. Vor allem die Möglichkeit, ein einziges Dokument als Grundlage für viele verschiedene Ziele heranziehen zu können ist herausragend. Aus Sicht des Autors kann man ohne Übertreibung behaupten, dass XAlgo ein zumindest vielversprechender Ansatz ist.

Kapitel 6

Dokumentation mit „Literate Programming“

Der Dokumentation von Programmen und Algorithmen kommt heutzutage, bei immer komplexeren und größeren Strukturen, eine immer bedeutendere Rolle zu. Bei den meisten gängigen Programmiersprachen werden spezielle Kommentarformen für Zusatzprogramme (z. B. Javadoc¹, CSC² oder Doxygen³) genutzt, die von diesen (oder entsprechenden Zusatzprogrammen) ausgewertet werden und so eine mehr oder minder detaillierte Dokumentation – meist als HTML-Seiten – generiert wird.

6.1 Literate Programming

Der Begriff „Literate Programming“⁴ wurde von dem legendären Informatiker Donald E. Knuth⁵ geprägt. In [Knuth 1983, Seite 1] fordert Knuth, dass nicht mehr nur der Programmcode im Mittelpunkt stehen sollte, sondern auf gleicher Ebene auch in menschenlesbarer Form beschrieben werden müsse, was wir damit durchführen lassen wollen. Knuth hat dazu an der Stanford University das „WEB System“ entwickelt, das hauptsächlich eine Kombination aus einer Formatierungssprache (im Stile von TeX) und einer Programmiersprache ist. Der wesentliche Punkt ist aber, dass sowohl das vollständige Programm als auch dessen Dokumentation aus der gleichen Quelle erzeugt werden kann [Knuth 1983, Seite 2].

¹siehe auch <http://java.sun.com/j2se/javadoc>

²in C#-Compiler integriert, liefert keine fertigformatierten Ausgaben

³siehe auch <http://www.stack.nl/~dimitri/doxygen/index.html>

⁴literate: engl. für gebildet

⁵gesprochen wie engl. „Ka-Nooth“; <http://www-cs-faculty.stanford.edu/~knuth/faq.html>

Anders als WEB von Knuth setzt XAlgo bei dieser Form von Literate Programming nicht auf eine formatierungsbezogene sondern auf eine contentbezogene Dokumentation im Programmcode. Wie letztendlich die erzeugte Dokumentationsausgabe formatiert ist, sollte bei einer mehr inhaltebezogenen Sprache wie XML eher durch separate so genannte Stylesheets geregelt werden.

6.2 Vergleich zu üblichen Dokumentationsmethoden

Herkömmliche Dokumentationssysteme arbeiten mit vom Benutzer speziell formatierten Kommentarzeilen, die dann automatisiert ausgewertet werden können. Bei diesen Systemen können aber nur bis zu einer – von den jeweiligen Programmen vorgegebenen – Granularitätsstufe die eingesetzten Sprachelemente beschrieben werden. Meist endet dies entweder schon auf Ebene der Prozeduren und Funktionen oder aber bei einzelnen Befehlen.

Bei XAlgo können mit diesem Dokumentationssystem jedoch die Beschreibungstexte auf beliebiger Hierarchiehöhe angegeben werden. So ist es zum Beispiel möglich, den Namen einer lokal angelegten Variable ausführlichst zu dokumentieren. Andererseits können die gleichen Beschreibungselemente auch auf der höchsten Ebene eingesetzt werden, um den generellen Zweck und Ziel eines Algorithmus zu umschreiben.

Kapitel 7

Rendering-Möglichkeiten

Ebenso wie andere XML-Sprachen, können auch XAlgo-Dokumente als Quelle für eine Vielzahl von gewünschten Ziel-Dokumenten genutzt werden. Hier werden nur die beiden naheliegendsten Formen kurz betrachtet, die aber dennoch leider im Rahmen dieser Arbeit nicht umgesetzt werden konnten (was aber auch nicht geplant war).

7.1 Detaillierte Dokumentation

Wie im Kapitel 6 zuvor beschrieben, kann man mit XAlgo auch umfangreiche Dokumentationen zu dem Algorithmus beifügen.

Diese können sehr leicht über ein XSLT-Stylesheet in XHTML, DocBook, XSL-FO oder andere Formatierungssprachen zur gewünschten Ausgabe umgesetzt werden. Diese Zielsprachen wiederum lassen sich mit verbreiteter Software – etwa Internetbrowsern für XHTML – leicht in ansehnliche Dokumente umwandeln (rendern¹).

7.2 Erweitertes Coderendering

Aber auch der algorithmusbezogene Anteil eines XAlgo-Dokumentes kann zum Beispiel dank vorhandener MathML-Rendering-Engines (wie die in den Mozilla-Webbrowser integrierte; siehe Abbildung 7.1 auf der nächsten Seite) zu einer besonders gut lesbaren Ausgabe gebracht werden.

Daher müssen Berechnungsvorschriften nicht mehr mit – für Menschen – schlecht lesbaren, häufig kompliziert kombinierten Berechnungsunterfunktionen angezeigt

¹engl. für wiedergeben

werde. Vielmehr können auf diese Weise Konstrukte wie Wurzeln, Potenzen, Mehrfachklammerungen etc. auch originalgetreu und mathematisch korrekt angezeigt werden.

Auch [Bauer et al. 1984, Seite 7] unterstreicht die Bedeutsamkeit der Darstellung: „Unter praktischen Gesichtspunkten kann das Erscheinungsbild eines Algorithmus nicht vernachlässigt werden. Menschliche Lesbarkeit ist für das Aufstellen und Verwenden eines Programms oder, wie wir es lieber sehen würden, für eine systematische Programmentwicklung, von kardinaler² Bedeutung.“

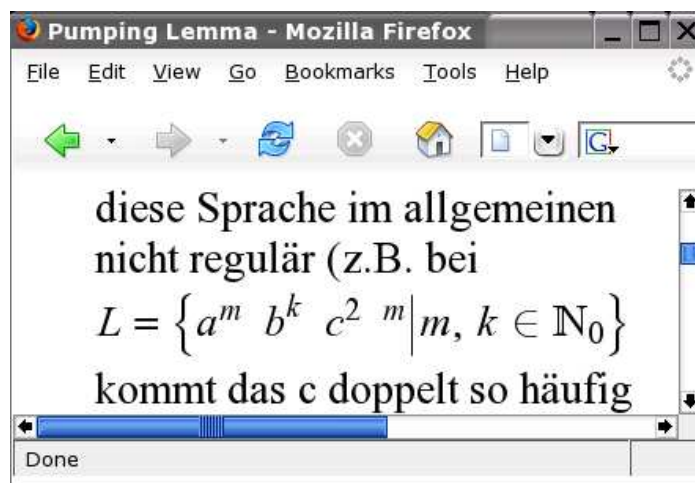


Abbildung 7.1: Darstellung einer in MathML beschriebenen Formel im Mozilla Firefox Webbrowser

²laut Rechtschreibungs-Duden: veraltet für grundlegend; hauptsächlich

Kapitel 8

Verwendungsmöglichkeiten

In diesem Kapitel wird ein Abriss dessen aufgezeigt, was schon jetzt mit XAlgo-Dokumenten möglich ist und was im nächsten Zuge für Verwendungsmöglichkeiten offenstehen.

8.1 Validierung über RELAX NG-Schema

RELAX NG¹ ist eine so genannte Schemasprache für XML [Vlist 2004, Seite xi]. Seit 2001 steht damit, neben den beiden noch gängigeren Alternativen (DTD und „XML Schema“), eine einfache, ausdrucksstarke und gut lesbare Sprache zur Beschreibung von Dokumentenstrukturen wieder anderer XML-Anwendungen zur Verfügung [Vlist 2004, Seite xi]. Gegenüber DTDs hat sie den Vorteil, selbst eine XML-Sprache zu sein. Im Vergleich zu XML Schema ist RELAX NG viel einfacher aufgebaut und somit auch leichter bzw. schneller erlern- und einsetzbar. Außerdem kann RELAX NG praktisch alle überhaupt möglichen XML-Sprachen² beschreiben, wohingegen XML Schema trotz seiner Komplexität nicht alle für XML-Sprachen gebotenen Möglichkeiten ausschöpfen kann [Vlist 2004, Seite xiii]. Daher fiel die Wahl bei dieser Arbeit auf RELAX NG statt der noch gebräuchlicheren Konkurrenz.

8.2 Generierung von Programmgerüsten

Mit der Hilfe von XSLT können aus XML-Dokumenten beliebige andere XML- oder Text-Dateien erzeugt werden. Wie im Praxisteil der Arbeit noch genauer

¹gesprochen wie engl. „relaxing“

²sog. wohlgeformtes XML Version 1.0 samt XML Namensräume

zu sehen sein wird, kann mit Hilfe des so genannten Push-Modells³ eine Art syntaxgesteuerte Übersetzung realisiert werden.

8.3 Kombination mit anderen XML-Techniken

Die Kombinationsmöglichkeiten von XAlgo mit anderen Mitgliedern der XML-Familie sind sehr vielseitig und ermöglichen ein noch nicht absehbares Wirkungsbereich für diese Sprache. Auch die im Kapitel 7 (auf Seite 19) genannten Wiedergabemöglichkeiten zählen schon zu diesem Gebiet.

8.3.1 Signierung von Algorithmen

Dank der Technik zur digitalen Signierung von XML-Dokumenten könnten man sich besser gegen absichtliche und unabsichtliche Veränderungen in den Algorithmen schützen, was sowohl bei der Übertragung – z. B. via Internet – als auch bei der Lagerung interessant ist. Bei der „Unterzeichnung“ werden die XML-Dokumente zunächst einer so genannten Kanonifizierung unterzogen. Dadurch wird das Dokument auf eine eindeutige Darstellung gebracht. In XML können inhaltlich exakt gleiche Dokumente durchaus textlich unterschiedlich dargestellt werden. Zum Beispiel kann man leere Elemente abkürzen oder voll ausschreiben, Attribute innerhalb von Elementen in ihrer Reihenfolge vertauschen⁴ etc. Das so entstandene Dokument (bzw. auch nur ein Teil) wird nun signiert.

Da XAlgo eine allgemeinere Form als konkrete Implementierungen hat, kann man es als Repräsentation einer ganzen Gruppe von möglichen Ausprägungen der verschiedensten Programmiersprachen auffassen. Da dieses Dokument nun dank der Kanonifizierung weiter unempfindlich gegen unbedeutende Änderungen geworden ist, bekommt man eine sehr zuverlässige Authentizitätsprüfung. Schränkt man bei der XML Signatur diese noch weiter auf die besonders relevanten Stellen ein (Teile der Dokumentation könnte man z. B. auslassen), kann man die Aussagekraft einer Signatur sogar noch weiter erhöhen.

8.3.2 Verschlüsselung

XML Encryption ermöglicht es, Dokumente beziehungsweise genau spezifizierte Teile davon zu verschlüsseln. Dies kann man so feingranular steuern, dass beispielsweise lediglich der Chefentwickler mit seinem „privaten Schlüssel“ Zugriff auf die vertraulichsten Stellen bekommt, ein Juniorentwickler hingegen nur die Grobstruktur betrachten kann.

³Eingangsdokument wird über passende Vorlagemuster durchgearbeitet, wobei das Zieldokument aufgebaut wird.

⁴Die Reihenfolge von XML-Attributen innerhalb eines Elementes ist definitionsgemäß nicht von inhaltlicher Bedeutung [W3C Rec. 2004, Punkt 3.1].

Teil II

Praxisteil

Kapitel 9

Entwurf von XAlgo

Am Anfang der praktischen Entwicklung stand der Entwurf der Sprache. Einerseits sollte XAlgo dabei möglichst unabhängig von Programmiersprachen aufgebaut sein, andererseits musste aber dennoch – schon für eine leichte Erlernbarkeit – Sorge dafür getragen werden, dass die verwendeten Konstrukte ähnlich denen von bekannten Programmiersprachen sind.

Es wurden dabei auch viele bisher kaum eingesetzte Techniken zur Qualitätskontrolle und Fehlerprüfung in Betracht gezogen. So zum Beispiel die Möglichkeit, über Vor- und Nachbedingungen zu jeder Anweisung eine semantische Validierung durchführen zu können. Derartige Bemühungen wurden jedoch zugunsten eines möglichst einfachen Aufbaues wieder verworfen.

9.1 Präzisierung des Sprachentwurfs

9.1.1 Das Typsystem

XAlgo hat nur ein extrem rudimentäres Typsystem, da weitere Spezialisierungen eher als implementatorisches Detail angesehen werden, die zudem auch über die Dokumentationselemente bei den jeweiligen Variablen oder Konstanten vermerkt werden könnten. Dies führt dazu, dass nur eine Reihe von Grundtypen und deren Einsatz als Vektoren bzw. Arrays vorgesehen sind. Selbst einfache Aufzählungstypen existieren hier nicht.

Eine Typüberprüfung innerhalb eines XAlgo-Dokumentes findet im augenblicklichen Entwicklungsstand noch nicht statt. Diese Aufgabe muss derzeit noch von dem nachgeschalteten Compiler der Zielsprache geleistet werden.

Einfache Typen

Als mögliche einfache Datentypen sind bei XAlgo im Moment nur die in Tabelle 9.1 vorgesehen. Die Frage nach der Genauigkeit bleibt dabei außer Acht, da es hier nur um den grundlegenden Typ geht und dessen Genauigkeit – zum Beispiel nach den Empfehlungen oder Anweisungen in den Dokumentationselementen – erst in der Zielsprache manuell angepasst werden sollte. Explizite Typumwandlungen zur Laufzeit sind nicht vorgesehen. Da die Datentypen aber weder von dem erstellten RELAX NG-Schema, noch von den Transformationskomponenten geprüft werden, kann zum Teil noch die implizite Typwandlung in den Zielsprachen genutzt werden.

Datentypname	Beschreibung
integer	ganze Zahlen
float	Gleitkommazahlen
boolean	Wahrheitswert
char	einzelnes Textzeichen
string	Zeichenkette

Tabelle 9.1: Einfache Datentypen von XAlgo

Arrays

Arrays dürfen Felder mit beliebig vielen Dimensionen in jeweils beliebiger Größe enthalten. Die Elemente der Felder dürfen nur aus einfachen Typen bestehen. Die einzelnen Dimensionen sind – vom Benutzer – benannt und werden nicht wie in Programmiersprachen meist üblich über deren Reihenfolge bei der Definition, sondern ausschließlich über diese einmal festgelegten Namen angesprochen.

Für den prozeduralen Anweisungsteil steht außerdem eine Funktion zur Ermittlung der Arraygröße zur Verfügung.

9.1.2 Bezeichner

Bezeichner (für Variablen, Konstanten, Klassen, Objekte etc.) sind in keinsten Weise von XAlgo beschränkt (sowohl weder in Länge noch im Zeichenumfang). Groß-/Kleinschreibung wird unterschieden. Es wird aber empfohlen diese Freiheit nicht unnötig zu strapazieren, weil die Übersetzungskomponenten z. B. die Sonderzeichen auf erlaubte Zeichen abbilden müssen und so die Bezeichner in der Zielsprache evtl. unleserlich werden.

9.1.3 Sichtbarkeit und Lebensdauer

Die Sichtbarkeit und Lebensdauer von Variablen, Konstanten, Objekten und Unterfunktionen ist in klassischer Weise von den gängigen Programmiersprachen übernommen. Alle selbstdeklarierten Elemente des Algorithmus existieren bis zum Ende der Hierarchiestufe, in der sie festgelegt wurden. Sichtbar sind diese Elemente auf ihrer eigenen und allen darunter angesiedelten Hierarchieebenen.

9.1.4 Subalgorithmen

Jeder Algorithmus darf beliebig viele Subalgorithmen definieren, die jeweils wieder ganz genauso aufgebaut sein können, wie der Hauptalgorithmus (das Wurzelement) des XAlgo-Dokumentes selbst.

Die Parameter für Subalgorithmen werden stets im „call-by-value“-Verfahren übergeben.

9.1.5 Klassen in XAlgo

XAlgo stellt ein sehr einfaches Klassensystem bereit, mit dem eine simple Objektorientierung realisiert wird. Die Mächtigkeit dieses Systems in XAlgo ist im Moment noch sehr beschränkt und reicht nur zur Modellierung äußerst einfacher Klassen, bildet also noch eine Möglichkeit für zukünftige Erweiterungen.

Klassenelemente

Klassen können beliebige Variablen, Konstanten und Unterfunktionen, hier so genannte Subalgorithmen, als Klassenelemente deklarieren. Klasseninstanzen dürfen jedoch an dieser Stellen nicht vorkommen.

Die Klassenmitglieder sind dabei alle frei und vollständig zugänglich. Ein Mittel zum so genannten Informationhiding existiert nicht.

Vererbung

Jede Klasse kann optional eine Basisklasse angeben, von der sie alle Elemente erbt. Es liegt demnach die Möglichkeit einer einfachen Vererbung vor, wobei eventuell geerbte Subalgorithmen in der neuen Klasse überschrieben werden dürfen. Dies muss allerdings explizit mit einem speziellen XAlgo-Element passieren (`<redefines>`), wobei der Ersatz dieselben Schnittstellen aufweisen muss.

Instanzenbildung

Im Deklarationsteil von Algorithmen ist ein spezielles XML-Element vorgesehen, um neben normalen Variablen und Konstanten auch Klasseninstanzen, also Ob-

jekte erzeugen zu können. Hierbei ist es auch möglich, einem Objekt auch die Instanz einer Kindklasse zuzuweisen, wobei dann auch eventuell überschriebene Subalgorithmen dieser Kindklasse genutzt werden.

9.1.6 Prozedurale Elemente

Für die prozeduralen Anweisungen stehen acht „Grundanweisungen“ bereit.

- Null-Anweisung
- Anweisungssequenz
- Schleife
- bedingte Ausführung
- Zuweisung
- Aufruf von Subalgorithmus
- Return-Anweisung

9.1.7 Elemente mit Nebenwirkungen

Neben den grundlegenden Anweisungen sind in XAlgo noch fünf spezielle Befehle realisiert, mit denen man auf das System zugreifen kann.

- Einlesefunktion (von Standardeingabe)
- Schreiben auf Standardausgabe
- Schreiben auf Fehlerausgabe
- Unix-Systemzeit
- Anhalten der kompletten Verarbeitung

9.1.8 Dokumentationselemente

Alle Dokumentationsumgebungen dürfen auf jeder Ebene, die keinen Textknoten erwartet, beliebig oft vorkommen. Eine Ausnahme hiervon bildet die `<metadocu>`-Umgebung, die je Ebene nur einmal verwendet werden kann. Die Beschreibungselemente innerhalb der Dokumentationselemente haben nur inhaltlichen Bezug, also keinen expliziten Formatierungseinfluss auf eine mögliche spätere Ausgabe.

<metadocu>-Umgebung

Innerhalb dieser Umgebung – also in weiteren Elementen zwischen `<metadocu>` und `</metadocu>` – werden Informationen festgehalten, die allgemeine Auskünfte über das jeweilige Algorithmentelement geben. Hierzu gehören beispielsweise Autor, Erstellungsdatum, Urheberrechts- und Lizenzvermerk etc.

<develdoc>-Umgebung

Hier können Zusatzinformationen zum Entwicklungsprozess festgehalten werden. Beispielsweise eine ToDo-Liste, bekannte Bugs, Nachrichten an Entwicklerkollegen usw.

<explanation>-Umgebung

Dabei handelt es sich um die eigentliche, inhaltlich beschreibende Dokumentation der Elemente im Algorithmus. Die dort eingesetzten Tags sind stark an die XML-Kommentare von C# angelehnt, jedoch für XAlgo angepasst.

<comment>-Umgebung

Diese Umgebung stellt einen einfachen Kommentar dar. Das Element erwartet entweder einen Textknoten oder so genannten mixed content¹. Damit sollten jedoch lediglich Zusatzinformationen abgelegt werden, die in keine der übrigen Umgebungen passen.

9.2 Festlegung über RELAX NG-Schema

Sowohl zur Festlegung der XAlgo-Dokumentenstruktur, XAlgo-Element- und Attributnamen, sowie zur späteren Validierung von ganzen XAlgo-Dokumenten, wurde zunächst ein RELAX NG-Schema erstellt.

9.2.1 Besondere Herausforderungen

Als besondere Schwierigkeit stellte sich die Aufgabe der Integration der Dokumentationselemente der Sprache heraus. Diese sind auf allen Ebenen und in allen Elementen erlaubt, bis auf diejenigen, in denen so genannte XML-Textknoten – also Dateninhalte – erwartet werden.

Obwohl RELAX NG sehr leicht zu erlernen und auch anzuwenden ist, gab es doch immer wieder Hürden der oberen Art. Diese speziellen Hindernisse konnten über

¹Text mit eingestreuten XML-Elementen

„benannte Referenzen“ (für die Dokumentationsstruktur), die bei jedem sonstigen XAlgo-Dokument mit eingebracht werden mussten, überwunden werden.

9.2.2 Zwei Versionen: Restriktiv oder offen

Ein ähnliches Problem ergab sich mit der Freigabe von allen „fremden“ XML-Elementen aus anderen Namensräumen. Da das Einbringen von fremden XML-Namensräumen – von dem fest vorgesehenen MathML einmal abgesehen – nicht immer genutzt wird, wurde zunächst eine „strict“-Version geschaffen, die nur die XAlgo-Elemente in ihrer entsprechenden Struktur und MathML-Elemente an den vorgesehenen Stellen erlaubt. Dies hat den Vorteil, dass bei dem Einsatz einer möglichst strengen Validierungsvorschrift mehr Fehler erkannt werden können.

Daneben existiert außerdem noch die „open“-Version, die relativ zu der strict-Version formuliert ist. Konkret bedeutet dies, dass eventuelle Änderungen an XAlgo in der strict-Version sich automatisch auch in der open-Version widerspiegeln. Der einzige inhaltliche Unterschied zwischen den beiden Varianten ist, dass in der offenen Version auch beliebige Elemente anderer Namensräume an allen Stellen erlaubt sind, an denen keine Textknoten erwartet werden. Möchte man also beispielsweise XML-Elemente einer weiteren XML-Formatierungssprache bei der Ausgabe einsetzen, muss man die open-Version des Schemas verwenden. Verzichtet man allerdings auf solche Möglichkeiten, ist es besser, die strict-Fassung einzusetzen.

Kapitel 10

Beschreibung von XAlgo

In diesem Kapitel kommt nun schließlich die ausführliche Beschreibung, wie XAlgo-Dokumente im Detail aufgebaut sind.

10.1 Grundaufbau von XAlgo-Dokumenten

Das Wurzelement (oft auch als Dokument-Element bezeichnet) heißt hier stets `<xalgo>` und befindet sich in dem Namensraum „<http://www.f-seidel.de/xmlns/2004/XAlgo/1.0>“. Die hier erlaubten Nachfolgeelemente sind (in genau dieser Reihenfolge):

- (optional) `<declarations>`
- (optional) `<definitions>`
- (zwingend) `<start>`

Das bedeutet, dass jedes XAlgo-Dokument zumindest eine `<start>`-Umgebung besitzen muss.

Innerhalb von `<declarations>` werden Algorithmennamen, Variablen, Konstanten, Klassen etc. deklariert (siehe Kapitel 10.2 auf der nächsten Seite). Durch die `<definitions>`-Umgebung können weitere Subalgorithmen angelegt werden. Dazu können dort beliebig viele `<subalgo>`-Elemente vorkommen, von denen jedes die gleiche Elementstruktur enthalten kann wie die Wurzel selbst. An dieser Stelle liegt also eine rekursive Definition der Sprache (auch im RELAX NG-Schema) vor. `<start>` beschreibt den prozeduralen Anteil eines Algorithmus (siehe Kapitel 10.3 auf Seite 38).

Ein Dokument sieht also auf der ersten Ebene mindestens wie Listing 10.1 aus und maximal wie Listing 10.2 auf der nächsten Seite.

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <start>
    ...
  </start>
</xalgo>
```

Listing 10.1: Minimale XAlgo-Grundstruktur

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <declarations>
    ...
  </declarations>
  <definitions>
    ...
  </defintions>
  <start>
    ...
  </start>
</xalgo>
```

Listing 10.2: Maximale XAlgo-Grundstruktur

10.2 Elemente für Programmdeklarationen

In der <declarations>-Umgebung könne bzw. müssen die folgenden Elemente verwendet werden (in dieser Reihenfolge):

- (zwingend) <name>
- (optional) <parameters>
- (beliebig oft¹) <classes>
- (beliebig oft) <constants>
- (beliebig oft) <variables>
- (beliebig oft) <objects>
- (optional) <returns>

¹auch gegebenenfalls null mal

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <declarations>
    <name>Hauptalgorithmus</name>
  </declarations>
  <definitions>
    <subalgo>
      <declarations>
        <name>Subalgorithmus1</name>
      </declarations>
      <start>
        ...
      </start>
    </subalgo>
  </definitions>
  <start>
    ...
  </start>
</xalgo>
```

Listing 10.3: XAlgo-Grundstruktur mit einem Subalgorithmus

10.2.1 Namensdeklaration

Das `<name>`-Element, das als einziges innerhalb der Deklarationen zwingend erforderlich ist, erwartet einen Textknoten als Inhalt.

Also zum Beispiel `<name>Quicksort-Algorithmus</name>`.

Der hier festgelegte Name wird – vor allem in Subalgorithmen – für das Referenzieren des Algorithmus herangezogen.

10.2.2 Parameter des Algorithmus

Der optionale `<parameters>`-Knoten wird im Moment von den Compilern nur in Subalgorithmen unterstützt. Es ist jedoch auch denkbar, dass hier selbst der „Hauptalgorithmus“ des Dokumentes die evtl. auf der Kommandozeile angebbaren Parameter spezifizieren kann.

Spezifizierung der einzelnen Parameter

Innerhalb der Umgebung können beliebig viele `<param>`-Elemente vorkommen. Jedes von Ihnen steht für einen beim Aufruf erwarteten Parameter.

Im Inneren können drei Spezifikationen angegeben werden (in dieser Reihenfolge):

- (zwingend) `<name>`
- (zwingend) `<type>`
- (optional) `<defaultvalue>`

Das `<name>`-Element ist auch an dieser Stelle immer nötig, da in XAlgo Parameter immer nur über ihren Namen angesprochen werden (selbst beim Aufruf von Subalgorithmen). Wie schon im vorigen Abschnitt erwartet auch dieses Element direkt einen Textknoten.

Mit `<type>` gibt man den erwarteten Datentyp des Parameters an. Es stehen dabei die in Abschnitt 9.1.1 (ab Seite 24) beschriebenen Typen zur Verfügung. Dies wiederum bedeutet, dass dort entweder direkt ein Textknoten mit der Typbezeichnung stehen kann oder eine Array-Struktur (siehe dazu auch Unterkapitel 10.2.8 auf Seite 36).

Wahlweise kann auch noch `<defaultvalue>` – über einen erwarteten Textknoten mit dem Wert – zur Festlegung einer Standardbelegung genutzt werden.

Eine typische Parameterliste eines Subalgorithmus könnte daher z. B. wie die in Listing 10.4 aussehen.

```
<parameters>
  <param>
    <name>radius</name>
    <type>float</type>
  </param>
  <param>
    <name>pi</name>
    <type>float</type>
    <defaultvalue>3.14159265</defaultvalue>
  </param>
</parameters>
```

Listing 10.4: Beispiel einer typischen Parameterliste

10.2.3 Klassendeklarationen

In den (beliebig vielen) `<classes>`-Tags stehen wiederum beliebig viele `<class>`-Umgebungen, von denen jede eine separate Klasse beschreibt. Mit `<classes>` kann man also zusammengehörige Listen aus Klassen gruppieren.

In den einzelnen `<class>`-Umgebungen kommen wiederum diese Elemente zur Beschreibung vor (ebenfalls in dieser Reihenfolge):

- (zwingend) `<name>`
- (optional) `<baseclass>`
- (optional) `<extensions>`
- (optional) `<redefines>`

Die `<name>`-Umgebung erwartet wieder (und auch künftig an allen auftretenden Stellen) den Namen als direkt enthaltenen Textknoten. `<baseclass>` erwartet unmittelbar den Namen einer Basisklasse, aus der alle Eigenschaften geerbt werden sollen. In `<extensions>` hingegen können in beliebiger Folge und Zahl Variablen (über `<variables>`-Elemente siehe Abschnitt 10.2.5), Konstanten (über `<constants>`-Elemente siehe Abschnitt 10.2.4) und Subalgorithmen (über `<subalgo>`-Elemente siehe Abschnitt 10.1 und Listing 10.3) angelegt werden. Diese werden dann den evtl. ererbten Klassenmitgliedern zugefügt. Mit Hilfe der `<redefines>`-Umgebung kann man ererbte Subalgorithmen überschreiben. Allerdings müssen diese die gleichen Spezifikationen für Parameter und Rückgabewert besitzen. Listing 10.5 zeigt ein Beispiel für die Deklaration einer Klasse.

10.2.4 Konstanten

Über `<constants>`-Umgebungen können Konstanten festgelegt werden. Darin wird zunächst immer über ein `<type>`-Element der Datentyp der folgenden Konstanten festgelegt. Danach folgen beliebig viele `<const>`-Umgebungen, die die einzelnen Konstanten repräsentieren. In diesen müssen stets die beiden Elemente `<name>` und `<value>` (in genau dieser Abfolge) vorkommen. Wie erwartet legt man damit zuerst den Namen und dann den Wert fest (siehe auch Listing 10.6).

10.2.5 Variablen

Variablen werden fast identisch zu den Konstanten beschrieben. Das `<variables>`-Element enthält zunächst immer ein `<type>`-Tag zur Definition des hier verwendeten Datentyps. Darauf folgen beliebige `<var>`-Umgebungen, von denen jede genau eine Variable darstellt. Innerhalb von `<var>` werden wieder die `<name>` und `<value>`-Elemente verwendet, wobei diesmal die Wertangabe freiwillig ist und als Vorbelegung genutzt wird (siehe auch Listing 10.7).

```
<classes>
  <class>
    <name>circle</name>
    <baseclass>point</baseclass>
    <extensions>
      <variables>
        <type>float</type>
        <var>
          <name>radius</name>
        </var>
        <var>
          <name>area</name>
        </var>
      </variables>
    </extensions>
  </class>
</classes>
```

Listing 10.5: Beispiel einer typischen Klassendeklaration

10.2.6 Klasseninstanzen

Instanzen – aus vorher oder in übergeordneten Ebenen deklarierten Klassen – werden in der `<objects>`-Umgebung angelegt. Wie in den vorherigen Deklarationen wird auch hier nun als nächstes der Typ (über `<type>`) festgelegt, wobei sich dieser aber nur auf Klassennamen beziehen darf. Als weiteres kommen nun die `<object>`-Elemente, die jeweils eine Instanz darstellen.

Ein Objekt wird nun weiter über den Namen (`<name>`-Element) und den Instanzentyp mit `<instancetype>` beschrieben. Der genaue Typ des Objektes muss hier deshalb noch einmal angegeben werden, weil man auch Instanzen einer Kindklasse von der vorher angegeben erzeugen kann (siehe auch Listing 10.8).

10.2.7 Rückgabewert

In der Spezifikation des optionalen Rückgabewertes über die `<returns>`-Umgebung werden zumindest das `<type>`-Element für den Rückgabebetyp und optional das `<default>`-Element für eine Standardbelegung des Rückgabewertes erwartet (siehe auch Listing 10.9). Möchte man explizit beschreiben, dass kein Rückgabewert vorgesehen ist, setzt man ausschließlich das `<null />`-Element in die `<returns>`-Umgebung.

```
<constants>
  <type>float</type>
  <const>
    <name>pi</name>
    <value>3.14159265</value>
  </const>
  <const>
    <name>e</name>
    <value>2.718281828</value>
  </const>
</constants>
```

Listing 10.6: Beispiel einer typischen Konstantendeklaration

```
<variables>
  <type>float</type>
  <var>
    <name>ergebnis</name>
  </var>
  <var>
    <name>zwischenergebnis</name>
    <value>0.0</value>
  </var>
</variables>
```

Listing 10.7: Beispiel einer typischen Variablendeklaration

10.2.8 Typbeschreibungen

Datentypen werden in XAlgo ausschließlich über `<type>` und nur in der Deklarationsumgebung (`<declarations>`) benutzt.

Einfache Datentypen

In `<parameters>`, `<classes>`, `<variables>`, `<constants>`, `<returns>` kann man über das `<type>`-Element direkt den Name einer der fünf Datentypen (siehe auch Tabelle 9.1 auf Seite 25) angeben.

```
<objects>
  <type>point</type>
  <object>
    <name>Zentrum</name>
    <instancetype>point</instancetype>
  </object>
  <object>
    <name>Zielgebiet</name>
    <instancetype>circle</instancetype>
  </object>
</objects>
```

Listing 10.8: Beispiel einer typischen Objektdeklaration

```
<returns>
  <type>float</type>
  <default>0.0</default>
</returns>
```

Listing 10.9: Spezifikation eines Rückgabewertes

Arrays

Arrays werden über die `<arraytype>`-Umgebung festgelegt und können überall dort stehen, wo auch einfache Datentypen erwartet werden. Innerhalb dieser gibt es genau ein `<elementtype>`-Element, das den einfachen Datentyp der Arrayelemente enthält, und je benötigter Dimension eine `<dimension>`-Umgebung.

Unterhalb von `<dimension>` stehen beim Anlegen von konkreten Arrays immer zwei Elemente: `<name>` für den Namen und `<length>` für die Größe der Dimension. Lediglich bei der Typbeschreibung für Parameter wird die Größenangabe weggelassen. Ein typisches Beispiel einer Arraydeklaration sieht man in Listing 10.10 auf der nächsten Seite.

Objekte

Bei der Erstellung von Objekten in der `<objects>`-Umgebung beziehen sich die Typangaben (in `<type>`) nur auf vorher selbstdeklarierte Klassennamen (siehe auch Abschnitt 12.3 auf Seite 69).

```
<variables>
  <type>
    <arraytype>
      <elementtype>integer</elementtype>
      <dimension>
        <name>x</name>
        <length>10</length>
      </dimension>
      <dimension>
        <name>y</name>
        <length>5</length>
      </dimension>
    </arraytype>
  </type>
  <var>
    <name>Array1</name>
  </var>
  <var>
    <name>Array2</name>
  </var>
</variables>
```

Listing 10.10: Beispiel einer typischen Arraydeklaration für zwei Arrays mit den zwei Dimensionen x und y der Größe 10 x 5

10.3 Beschreibung der prozeduralen Elemente

Alle im Folgenden beschriebenen prozeduralen Elemente dürfen optional noch ein `name`-Attribut tragen. Dieses hat keinerlei inhaltliche Funktion. Es dient lediglich als Orientierungshilfe bei einer späteren eventuellen Fehlersuche. Die dort eingesetzte Zeichenkette wird vom Übersetzer als Kommentar vor die jeweiligen Anweisungen der Zielsprache gesetzt. Auf diese Weise kann man bei auftretenden Störungen leichter nachvollziehen, aus welchen XAlgo-Anweisungen die betreffenden Befehle hervorgegangen sind.

Prozedurale Anweisungen stehen in XAlgo-Dokumenten immer (und ausschließlich) innerhalb von `<start>`-Umgebungen, wobei sie wie erwartet in der Reihenfolge ihres Auftretens abgearbeitet werden.

10.3.1 Null-Anweisung

Die Null-Anweisung wird über `<null />` (oder ausführlich `<null></null>`) angegeben und muss immer leer sein, darf also keine weiteren Elemente oder Text enthalten. Die Anweisung selbst ist nur eine „Füllanweisung“ und hat keine Wirkung. Sie kann z. B. für noch unfertige (Sub-) Algorithmen verwendet werden, bei denen der prozedurale Teil noch nicht formuliert werden kann. Da jede `<start>`-Umgebung immer (genau) einen Befehl erwartet, kann damit eine „wirkungslose“ Prozedur aufgebaut werden. So würde ein minimales XAlgo-Dokument z. B. wie das aus Listing 10.11 aussehen.

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <start>
    <null />
  </start>
</xalgo>
```

Listing 10.11: Minimales XAlgo-Dokument (Unbenannter Algorithmus ohne jede Wirkung)

10.3.2 Sequence-Struktur

Die `<sequence>`-Umgebung kann überall dort stehen, wo eine prozedurale Anweisung erwartet wird und nimmt in sich wiederum beliebige prozedurale Elemente auf, die der Reihe nach verarbeitet werden (siehe Listing 10.12).

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <start>
    <sequence>
      <assign> ... </assign>
      <loop> ... </loop>
      <return>0</return>
    </sequence>
  </start>
</xalgo>
```

Listing 10.12: Beispiel einer Sequence-Struktur

10.3.3 Schleifen

Mittels `<loop>` können auf sehr universelle Art Schleifen beschrieben werden. Dazu werden zunächst in der eingebetteten `<conditions>`-Struktur die genauen Wiederholungsregeln für die Schleife festgelegt. In der darauf folgenden `<body>`-Umgebung wird der Schleifenrumpf beschrieben.

Die Schleifenbedingungen

Innerhalb `<conditions>` darf zunächst optional das `<loop-init-statement>`-Element verwendet werden. In diesem kann eine Anweisung (oder via `<sequence>` natürlich auch mehrere) eingefügt werden, die ein einziges mal beim Betreten der Schleife ausgeführt wird. Dies kann beispielsweise zum Initialisieren von Zählervariablen genutzt werden.

Darauf folgt die `<breakcondition>`-Umgebung, die als Mindestangabe immer in den Schleifenbedingungen stehen muss. In ihr wird ein boolescher Ausdruck erwartet, wie er später in Abschnitt 10.4.2 (auf Seite 49) beschrieben wird. Ist der Ausdruck erfüllt bzw. ergibt seine Prüfung den Wert „true“, wird die Verarbeitung des Schleifenrumpfes an der angegebenen Stelle abgebrochen.

Optional kann nun noch ein `<loop-update-statement>`-Element folgen, welches die in ihm enthaltene Anweisung nach jedem Durchlaufende des Rumpfes ausführt. Dies kann z. B. genutzt werden, um eine Variable zu inkrementieren.

Der Schleifenrumpf

Die `<body>`-Umgebung erwartet eine Folge von Anweisungen, wie sie auch in `<sequence>` stehen könnten. Des Weiteren stehen hier drei zusätzliche Befehle zur Verfügung, mit denen der Schleifenablauf kontrolliert wird (siehe Tabelle 10.1 auf der nächsten Seite). Nutzt man diese nicht, hat man – unabhängig von der gewählten Schleifenbedingung – eine Endlosschleife geschaffen. Außerdem bemerkenswert ist, dass der Schleifenrumpf der einzige Fall in XAlgo ist, wo direkt – ohne `<sequence>` – mehrere Anweisungen stehen dürfen. Da man davon ausgehen kann, dass in einer Schleife mindestens eine Anweisung wiederholt ausgeführt werden soll und man stets noch mindestens eine der Schleifenkontrollbefehle benötigt, wird hier sozusagen implizit eine Sequenz-Struktur angenommen. In Listing 10.13 ist ein Beispiel eines Schleifenkonstrukts zu finden.

10.3.4 Bedingte Ausführung

Über die `<choice>`-Struktur kann eine bedingte Ausführung gesteuert werden. Hierfür benutzt man als erstes Unterelement `<select>`, worin ein beliebiger Ausdruck zur Auswertung stehen darf.

```

<loop>
  <conditions>
    <breakcondition>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply><gt /><ci>ergebnis</ci><ci>31</ci></apply>
      </math>
    </breakcondition>
  </conditions>
  <body> <checkloopcondition /> ... </body>
</loop>

```

Listing 10.13: Beispiel einer Schleifen-Struktur

Anweisung	Beschreibung
<checkloopcondition />	An dieser Stelle wird die Schleifenbedingung geprüft und bei Ihrer Erfüllung die Schleifenverarbeitung abgebrochen. Setzt man die Anweisung an den Anfang des Rumpfes erhält man eine „while not do“-Schleife. An das Ende gesetzt entspräche sie einer „repeat until“-Schleife. Die Anweisung darf aber auch zwischen den übrigen vorkommen. Nach [Pratt et al. 1997, Seite 361] („do-while-do“) eine Lösung, um goto-Anweisungen noch besser vermeiden zu können.
<continueloop />	Überspringen des restlichen Schleifenrumpfes
<breakloop />	Vorzeitig verlassen der gesamten Schleife

Tabelle 10.1: Anweisungen zur Kontrolle des Schleifenablaufs

Danach folgen beliebig viele <case>-Blöcke. In diesen wiederum kann zunächst eine optionale <condition>-Umgebung vorkommen, die auch einen Ausdruck enthält. Dabei kommt nur der erste der <case>-Blöcke zur Ausführung, dessen ausgewertete <condition> mit dem Ergebnis aus <select> übereinstimmt. Fehlt diese Ausführungsbedingung in einem Block, wird der boolesche Ausdruck „true“ als Standardwert angenommen, da sich auf diese Weise ein „if ... then ... else ...“-ähnliches Konstrukt sehr einfach erzeugen lässt. Die in jedem Block zwingende <then>-Struktur enthält den zur jeweiligen Ausführung gewollten Befehl.

Nach den <case>-Umgebungen kann optional noch ein <othercase>-Element folgen, dessen <then> eine Anweisung enthält, die nur ausgeführt wird, wenn keiner der

vorherigen Blöcke gegriffen hat. Ein `<condition>` innerhalb von `<othercase>` ist unnötig und auch nicht erlaubt.

In Listing 10.14 sieht man ein sehr einfaches Beispiel einer „if ... then ... else ...“-Entsprechung. Listing 10.15 auf der nächsten Seite zeigt ein allgemeineres Beispiel der `<choice>`-Umgebung.

```
<choice>
  <select>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <eq />
        <ci>Radius</ci>
        <cn>0</cn>
      </apply>
    </math>
  </select>
  <case>
    <then>
      <output>
        <text>Punkt</text>
      </output>
    </then>
  </case>
  <othercase>
    <then>
      <output>
        <text>Kein Punkt</text>
      </output>
    </then>
  </othercase>
</choice>
```

Listing 10.14: Beispiel einer bedingten Ausführung, die einem „if ... then ... else ...“-Konstrukt entspricht: Wenn `Radius=0`, dann schreibe 'Punkt', ansonsten schreibe 'Kein Punkt'.

```
<choice>
  <select>benutzername</select>
  <case>
    <condition><text>Meier</text></condition>
    <then>
      <output>
        <text>Hallo, Herr Meier.</text>
      </output>
    </then>
  </case>
  <case>
    <condition><text>Knuth</text></condition>
    <then>
      <output>
        <text>Es ist mir eine Ehre.</text>
      </output>
    </then>
  </case>
  <othercase>
    <then>
      <output>
        <text>Ich kenne Sie nicht!</text>
      </output>
    </then>
  </othercase>
</choice>
```

Listing 10.15: Beispiel einer bedingten Ausführung über `<choice>` (Anmerkung: `<sequence>` ist hier unnötig, da jeweils nur der eine Befehl `<output>` genutzt wird.)

10.3.5 Zuweisungen

Mit Hilfe des `<assign>`-Elements kann man Zuweisungen realisieren. Es werden dabei immer die beiden Kindelemente `<source>` und `<target>` (in genau dieser Folge) erwartet. Die Quelle wird dabei mit einem frei wählbaren Ausdruck angegeben, während das Ziel ein vorher deklariertes Bezeichner sein muss (siehe auch Listing 10.16).

```
<assign>
  <source>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <plus />
        <ci>alterKmStand</ci>
        <ci>gefahreneKm</ci>
      </apply>
    </math>
  </source>
  <target>KmStand</target>
</assign>
```

Listing 10.16: Beispiel einer Zuweisung

10.3.6 Subalgorithmen-Aufruf

Subalgorithmen werden mit `<callsub>` aufgerufen. Darin muss stets ein `<name>`-Element den Namen des aufzurufenden Subalgorithmus bestimmen. Erwartet dieser noch Parameter, so sind diese nach dem Namen innerhalb von `<parameters>` anzugeben.

`<parameters>` erwartet hierzu entsprechende `<param>`-Blöcke, die wiederum über `<name>` und `<value>` die Aktualparameter des Aufrufes festlegen. In welcher Reihenfolge die `<param>`-Umgebungen beschrieben werden ist nicht relevant, da die Parameter nur über ihre Namen angesprochen werden.

Soll ein eventuell vorhandener Rückgabewert der aufgerufenen Programmeinheit aufgefangen und in einer Variablen entsprechenden Typs gespeichert werden, so kann man optional noch mit dem `<returnvaluetarget>` den Namen einer passenden Variablen angeben (siehe auch Listing 10.17 auf der nächsten Seite).

Nach Beendigung des Subalgorithmus und dem Ablegen des Rückgabewertes wird der Programmpfad nach dem Subalgorithmen-Aufruf weiter fortgesetzt.

```
<callsub>
  <name>Flaechenberechnung</name>
  <parameters>
    <param>
      <name>Laenge</name>
      <value>12.7</value>
    </param>
    <param>
      <name>Breite</name>
      <value>3.5</value>
    </param>
  </parameters>
  <returnvaluetarget>Grundflaeche</returnvaluetarget>
</callsub>
```

Listing 10.17: Beispiel eines Subalgorithmenaufwurfes

10.3.7 Rückkehranweisung

Mit `<return>` kann man einen Algorithmus beenden. Der eventuell im Element stehende Ausdruck wird ausgewertet und an den Aufrufer zurückgegeben. Ist die Rückkehranweisung leer, aber es wurde vorher ein Ausdruck als Standardwert für die Rückkehr angegeben, wird dieser nun ausgewertet und an den Aufrufer zurückgegeben. War dies jedoch nicht der Fall, wird der Algorithmus ohne Rückgabewert einfach beendet.

```
...
<start>
  <sequence>
    ...
    <return>Ergebniswert</return>
  </sequence>
</start>
...
```

Listing 10.18: Beispiel einer Rückkehranweisung

10.3.8 Systemanweisungen

Da mit XAlgo Algorithmen möglichst einfach beschrieben werden sollen, aber andererseits nicht so einfach Programmbibliotheken, z. B. für die Ein- und Ausgabe, eingebunden werden können, stellt die Sprache diese fünf rudimentären Schnittstellen zum Laufzeitsystem bereit.

Die `<input>`-Anweisung

Das `<input>`-Element erwartet nur den Namen eines Elements, in dem ein von der Tastatur bzw. Standardeingabe eingelesenes Datum abgelegt werden kann. Optional kennt das Element auch das `file`-Attribut, das aber im Augenblick von den Compilern nicht unterstützt wird. Darin darf entweder „stdin“ für die Standardeingabe oder eine beliebige URI als Datenquelle stehen. Ein Beispiel ist in Listing 10.19 zu finden.

```
<input file="stdin">ErfragterKontostand</input>
```

Listing 10.19: Beispiel einer Eingabeanweisung

Die `<output>`-Anweisung

In dem `<output>`-Element kann ein Ausdruck eigener Wahl stehen, dessen Ergebnis auf die Standardausgabe des Systems ausgegeben wird. Auch hier sieht XAlgo eigentlich den Einsatz des `file`-Attributs vor, das hier entweder „stdout“ oder eine beliebige URI als Ausgabeziel enthalten darf (jedoch noch ununterstützt). In Listing 10.20 ist die Ausgabeanweisung für ein Hallo-Welt-Programm dargestellt.

```
<output file="stdout"><text>Hallo Welt.</text</output>
```

Listing 10.20: Beispiel einer Ausgabeanweisung

Die `<errorout>`-Anweisung

Auch das `<errorout>`-Element wird zur Ausgabe benutzt (siehe Listing 10.21 auf der nächsten Seite). Allerdings sollten hiermit nur Fehlerausgaben angewiesen werden, da die Standardfehlerausgabe verwendet wird. Ansonsten unterscheidet es sich von `<output>` nur dadurch, dass die Standardoption für das `file`-Attribut „stderr“ ist.

```
<errorout><text>Fehler: Wert ungueltig!</text></errorout>
```

Listing 10.21: Beispiel einer Fehlerausgabe

Der <unixtime />-Ausdruck

Das leere <unixtime />-Element steht in Ausdrücken für die jetzige Zeit in der so genannten Unixtime². Dies könnte beispielsweise für Zeitmessungen herangezogen werden. Ein Beispiel zum Ablegen der Zeit in einer (integer) Variablen ist in Listing 10.22 zu sehen.

```
<assign>
  <source>unixtime /</source>
  <target>Programmbeginn</target>
</assign>
```

Listing 10.22: Beispiel einer Zeitermittlung

Die <halt />-Anweisung

Mit der <halt />-Anweisung ist es aus jeder Ebene möglich, die gesamte Algorithmenverarbeitung sofort zu stoppen. Zur Erinnerung: mit <return> wird nur der gerade ablaufende (Sub-) Algorithmus beendet und gegebenenfalls zum Aufrufer zurückgekehrt. Listing 10.23 zeigt ein kleines Beispiel zu dem Befehl.

```
<sequence>
  ...
  <choice>
    ...
    <othercase>
      <then>halt /</then>
    </othercase>
  </choice>
</sequence>
```

Listing 10.23: Beispiel einer <halt />-Anweisung

²Sekunden seit 0 Uhr UTC am 01.01.1970

10.4 Ausdrücke

Ausdrücke werden an sehr vielen Stellen in XAlgo-Dokumenten benötigt. Im Folgenden werden die verschiedenen möglichen Ausdrucksformen erläutert.

10.4.1 Mathematische Ausdrücke

Ausdrücke mathematischer Art werden in XAlgo-Dateien immer über MathML³-Umgebungen beschrieben. Eine vollständige Beschreibung dieser XML-Sprache würde leider den Rahmen dieser Arbeit sprengen. Aus diesem Grunde ist hier nur eine kurze, eher praktisch orientierte Einführung zu finden.

Das Hauptelement der MathML-Struktur ist das `<math>`-Element, das im Namensraum „`http://www.w3.org/1998/Math/MathML`“ liegen muss. Unterhalb davon befindet sich eine `<apply>`-Umgebung, die die eigentliche mathematische Operation aufnimmt. Außerdem hat sie eine „Klammer“-Wirkung und kann somit zur Prioritätsregelung eingesetzt werden.

Der nun folgende Operator gibt an, wie die folgenden Elemente zu verarbeiten sind. Die in Tabelle 10.2 erläuterten mathematischen MathML-Operatoren stellen die wohl gängigsten (und vom Compiler derzeit unterstützen) dar.

XML-Element	Operator-Beschreibung
<code><plus /></code>	Additions-Operator: Alle folgenden Werte werden addiert.
<code><minus /></code>	Subtraktions-Operator: Erwartet nur zwei Werte, wobei der zweite vom ersten subtrahiert wird.
<code><times /></code>	Multiplikations-Operator: Alle folgenden Werte werden miteinander multipliziert.
<code><div /></code>	Divisions-Operator: Erwartet nur zwei Werte, wobei der erste durch den zweiten dividiert wird.
<code><mod /></code>	Modulo-Operator: Erwartet nur zwei Werte, wobei der Divisionsrest der beiden ermittelt wird.

Tabelle 10.2: Wichtige bzw. unterstützte mathematischen MathML-Operatoren aus [Mangano 2003, Seite 136]

Die Werte werden in einer `<ci>`-Umgebung angegeben, wenn es sich um ein per

³siehe Abkürzungsverzeichnis

Bezeichner angesprochenes Datum handelt, oder in einer `<cn>`-Umgebung, wenn Zahlen direkt (also als literale Konstanten) angegeben werden. Ein typisches Beispiel eines mathematischen Ausdrucks für XAlgo sieht man in Listing 10.24.

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <apply>
    <plus />
    <cn>32</cn>
    <ci>x</ci>
  </apply>
</math>
```

Listing 10.24: Beispiel des mathematischen Ausdrucks $(32 + x)$

10.4.2 boolesche Ausdrücke

Auch boolesche (Wahrheits-) Ausdrücke werden in MathML formuliert (siehe Listing 10.25). Hierbei kommen ebenfalls die `<math>`- und `<apply>`-Umgebungen zum Einsatz, aber mit Operatoren zur Erzeugung oder Verknüpfung von Wahrheitswerten (aus Tabelle 10.3 auf der nächsten Seite).

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <apply>
    <lt />
    <ci>x</ci>
    <ci>y</ci>
  </apply>
</math>
```

Listing 10.25: Beispiel des booleschen Ausdrucks $(x < y)$

10.4.3 Pfade zu Klassenelementen

Um in Programmiersprachen die Elemente eines Objektes anzusprechen gibt es in aller Regel spezielle Zeichenfolgen wie z.B. den Punkt oder „->“. Da XAlgo jedoch XML als Syntaxgrundlage verwendet, gibt es hier natürlich eine passende Elemente-Struktur. Über die `<elementpath>`-Umgebung kann man das gewünschte Klassenmitglied ansprechen. Im ersten Unterelement `<name>` gibt man nun den Bezeichner, also den Namen der gewünschten Klasseninstanz, an. Danach folgt

XML-Element	Operator-Beschreibung
<code><eq /></code>	Gleichheits-Operator: wahr, wenn zwei gleiche Werte
<code><neq /></code>	Ungleichheits-Operator: wahr, wenn zwei Werte unterschiedlich
<code><leq /></code>	Kleiner-Gleich-Operator: wahr, wenn erster Wert kleiner oder gleich dem zweiten
<code><lt /></code>	Kleiner-Operator: wahr, wenn erster Wert echt kleiner als zweiter
<code><geq /></code>	Größer-Gleich-Operator: wahr, wenn erster Wert größer oder gleich dem zweiten
<code><gt /></code>	Größer-Operator: wahr, wenn erster Wert echt größer als zweiter
<code><and /></code>	Und-Operator: wahr, sobald alle Teilausdrücke wahr sind
<code><or /></code>	Oder-Operator: wahr, sobald ein Teilausdruck wahr ist

Tabelle 10.3: Unterstützte Vergleichs- und Verknüpfungsoperatoren aus MathML

(optional⁴) die `<sub>`-Umgebung, die in klassischen Sprachen dem Punkt entspricht. Innerhalb von `<sub>` folgt nun wieder ein `<name>`-Element, der schließlich den Namen des Klassenmitglieds erwartet. In Listing 10.26 sieht man ein typisches Beispiel.

```

<elementpath>
  <name>kreisinstanz</name>
  <sub>
    <name>radius</name>
  </sub>
</elementpath>

```

Listing 10.26: Beispiel eines Elementpfades

⁴Weil `<elementpath>` in `<sub>` rekursiv definiert ist; zur Schachtelung, falls künftig innerhalb von Klassen auch Objekte deklariert sein können

10.4.4 Text

Für die Belegung oder den Vergleich von Strings oder zur Ausgabe möchte man auch Zeichenketten möglichst direkt angeben können. Dies ist mit der `<text>`-Umgebung möglich, die direkt eine Zeichenkette als Inhalt erwartet (siehe Listing 10.27).

```
<assign>
  <source>
    <text>Hans Meier</text>
  </source>
  <target>Personenname</target>
</assign>
```

Listing 10.27: Beispiel eines Text-Ausdrucks in einer Zuweisung

10.4.5 Arrayelemente

Elemente von Arrays werden in Ausdrücken über die `<array>`-Umgebung angesprochen (siehe Listing 10.28). In `<name>` wird der Bezeichner des Arrays angegeben. Darauf folgt je vorhandener Dimension eine `<dimension>`-Struktur. Diese wiederum müssen (in dieser Abfolge) den Namen (auch über `<name>`) und den anzusprechenden Indexwert (in `<index>`) enthalten. Der Index wird dabei immer von 0 (bis zu (*Dimensionslänge* - 1)) gezählt.

```
<array>
  <name>demoarray</name>
  <dimension>
    <name>x</name>
    <index>3</index>
  </dimension>
  <dimension>
    <name>y</name>
    <index>5</index>
  </dimension>
</array>
```

Listing 10.28: Beispiel für das Ansprechen von Arrayelementen

10.4.6 Arrayliterale

Die direkte Festlegung eines vollständigen Arrays einschließlich Inhalt in einem einzigen Ausdruck ist zwar versuchsweise im RELAX NG-Schema vorgesehen (über die `<arrayliteral>`-Umgebung), gehört aber nicht zum „offiziellen“ Sprachumfang von XAlgo.

Für die Realisierung eines solchen Konstrukts müssten gegebenenfalls aufwendige Matrixtransformationen vorgenommen werden – besonders bei großen vieldimensionalen Feldern. Da die Dimensionen nur über ihre Namen angesprochen werden und im Vorhinein nicht bekannt ist, in welcher Folge sie vom Benutzer angegeben werden, muss das vollständig belegte Array im Compiler erzeugt werden und entsprechend der Ausrichtung des Zielarrays umgeformt werden. Allerdings ist es auch mit den bisher vorgestellten Methoden möglich, ein Array mit Daten zu füllen.

Zwar wären Arrayliterale etwas komfortabler zur Erstellung größerer Arrays, jedoch wurde XAlgo auch nicht zur direkten Verarbeitung von großen Datenmengen entworfen, sondern vielmehr zur möglichst allgemeinen Formulierung von Algorithmen. Hierzu ist es jedoch in aller Regel nicht nötig, große vorbelegte Felder zu beschreiben.

10.4.7 Sonstige

Arraygröße

Zur Bestimmung der Größe eines Arrays bzw. der einzelnen Dimensionen kann man die `<arraylength>`-Struktur verwenden. Als erstes erwartet sie ein `<name>`-Element mit dem Namen der Arrayvariablen. Danach muss das `<dimension>`-Element folgen, über dessen eigenes `<name>`-Element man den Namen der Dimension angibt, für die man die Größe erfahren möchte. Es kann in einer `<arraylength>`-Umgebung immer nur eine Dimensionsgröße ermittelt werden. Listing 10.29 verdeutlicht die Verwendung an einem Beispiel.

Native Ausdrücke

Mit dem `<directnative>`-Element existiert eine „Hintertür“, über die man Ausdrücke in herkömmlicher Form angeben kann. Der Inhalt wird dabei ohne irgendeine Anpassung direkt an die Zielsprache übergeben. Da der Einsatz eines solchen Mittels aber die Idee der programmiersprachlichen Unabhängigkeit hintergeht, wird von dem Einsatz dringend abgeraten. Das Element kann jedoch hilfreiche Dienste während der Algorithmusentwicklung leisten (z. B. zur Fehlersuche in MathML-Ausdrücken). Listing 10.30 zeigt ein Beispiel zur Verwendung.

```
<assign>
  <source>
    <arraylength>
      <name>Vektor1</name>
      <dimension>
        <name>x</name>
      </dimension>
    </arraylength>
  </source>
  <target>Vektorgroesse</target>
</assign>
```

Listing 10.29: Beispiel der Größenbestimmung eines Arrays in einer Zuweisung

```
<assign>
  <source>
    <directnative>10*(3*x+y)</directnative>
  </source>
  <target>Ergebnisvariable</target>
</assign>
```

Listing 10.30: Beispiel für nativen Ausdruck in einer Zuweisung

10.5 Dokumentationselemente

Wie schon in Kapitel 9.1.8 beschrieben, dürfen die Dokumentationselemente von XAlgo überall dort eingestreut werden, wo keine Textknoten erwartet werden. Abschließend finden sie in Listing 10.32 (auf Seite 57) ein kurzes Beispiel, das die Verwendung einiger hier erklärter Elemente verdeutlichen soll.

10.5.1 Metainformationen

Eine Reihe von Metainformationen zu den Elementen einer Ebene kann man mit der `<metadocu>`-Umgebung angeben. In ihr können die Elemente in den folgenden Abschnitten (in arbiträrer Abfolge) vorkommen.

Autorinformationen

Die `<author>`-Struktur darf mehrmals (also einmal je Autor) vorkommen und enthält mindestens das Element `<personname>`, das den vollständigen Namen dieses Autors enthalten soll. Nach Bedarf kann man noch die beiden Elemente

<birthday> und <homecountry> zur eindeutigeren Spezifikation des Autors heranziehen. Dabei erwartet <birthday> ein Datum und <homecountry> einen Ländercode nach RFC 1766 (also z. B. „de“).

```
<metadocu>
  <author>
    <personname>Frank Seidel</personname>
    <homecountry>de</homecountry>
  </author>
</metadocu>
```

Listing 10.31: Beispiel für Dokumentation von Autoren

Erstellungsdatum

Das Erstellungsdatum kann man über das <creationdate>-Element festlegen. Dieses erwartet entweder ein Datum oder ein Datum mit Uhrzeit (nach ISO 8601).

Copyright-Informationen

Informationen zum Copyright kann man über das <copyright>-Element angeben, das einen beliebigen Text erwartet.

Lizenzinformationen

Die <licence>-Umgebung zur Spezifizierung der vorliegenden Lizenz kennt die drei optionalen Elemente <licencetype> für den Lizenztyp (Open-Source etc.), <name> für den Namen der Lizenz (z. B. GPL) und <fulltext> für den vollständigen Wortlaut der Lizenz, die alle beliebigen Text enthalten dürfen.

Versionsinformationen

In <version> können in freiem Text Versionsinformationen angegeben werden.

Compilierinformationen

Hilfreiche Informationen zur Übersetzung des Abschnittes kann man (auch in beliebigem Text) in dem <build>-Element unterbringen.

Zweck

Über die <purpose>-Umgebung kann man Zweck bzw. das Ziel frei beschreiben.

Anmerkungen

Mit dem `<annotation>`-Element kann man sonstige Anmerkungen angeben, die in keine der sonstigen Elemente passen.

10.5.2 Entwicklungs-Dokumentation

Mit diesen Zusatzinformationen zum Entwicklungsprozess soll die Zusammenarbeit erleichtert und das Führen einer separaten Entwicklungsdokumentation abgenommen werden. Alle hier beschriebenen Elemente können (jeweils optional) innerhalb der `<develdoc>`-Umgebungen eingesetzt werden. Dabei darf jedoch jede dieser Umgebungen nur einmal benutzt werden. Für mehrere gleiche Einträge sollten mehrere `<develdoc>`-Umgebungen verwendet werden.

Noch zu erledigendes

In `<todo>` wird (in beliebiger Textform) vermerkt, was an dem jeweiligen Bereich künftig noch zu erledigen ist, bevor er als abgeschlossen gelten darf.

Datum des Entwicklungsstandes

Über `<date>` kann vermerkt werden, von welchem Stand die Eintragung der umgebenden `<develdoc>`-Umgebung sind. Als Format hierfür wird wie bei `<creationdate>` ISO 8601 erwartet.

Verfasser

Innerhalb des `<from>`-Elements kann man festhalten, wer diese Eintragungen zum Entwicklungsstand vorgenommen hat.

Zielpersonen

In `<to>` kann man direkt und frei beschreiben, an wen sich diese Informationen richten.

Nachrichtenaustausch

Mit dem `<message>`-Element kann man zusätzliche kurze Nachrichten an die Leser oder die mit `<to>` festgelegte Zielgruppe oder -Person richten.

Fehlerbeschreibung

Über `<errordescription>` sollen Fehlerbeschreibungen gesammelt bzw. dokumentiert werden.

Erledigte Probleme

In `<fixed>` ist möglich, bereits erledigte Probleme und Fehler festzuhalten.

Anmerkungen

Alle übrigen Informationen zum Entwicklungsprozess, die nicht in die vorherigen Umgebungen passen, können in dem `<annotation>`-Element beschrieben werden.

10.5.3 Erklärungskomponente

Die `<explanation>`-Struktur enthält die eigentliche Dokumentation zu den Sprach-elementen von XAlgo.

Zusammenfassung

Über `<summary>` kann man eine kurze Zusammenfassung zu dem beschriebenen Element festhalten. Beim späteren Rendering könnte dies z. B. für eine Übersicht genutzt werden.

Details

In `<details>` kann in freiem Text ausführlich die Funktion, Intention etc. der Sprachelemente beschrieben werden.

Beispiele

Das `<example>`-Element darf in einer Erklärungsumgebung auch mehrfach vorkommen und enthält jeweils Beispiele zu dem beschriebenen Sprachkonstrukt.

Siehe auch . . .

Auch `<seealso>` kann öfters benutzt werden, um auf andere Sprachkonstrukte zu verweisen, die mit diesem in Zusammenhang stehen.

Links

Mit `<link>` kann man schließlich über eine URI einen Link spezifizieren, der weiterführende Informationen enthält. Natürlich kann auch dieses Element beliebig oft in einer Erklärungskomponenten vorkommen.

10.5.4 Einfache Kommentare

Zwar hat XML eine eigene Kommentarform (`<!-- ... -->`), doch könnte mit diesen nicht unterschieden werden, ob sich der Kommentar auf z. B. technische Verarbeitungsdetails des Dokumentes oder den algorithmischen Inhalt bezieht. Daher ist von XAlgo noch das `<comment>`-Element vorgesehen, in dem ein beliebiger Kommentartext stehen kann. Zwar sollte man zur Dokumentation besser die vorher vorgestellten Umgebungen heranziehen, doch ist diese Kommentarform immer noch sinnvoller, als sonstige Informationen mit den XML-eigenen Kommentaren zu versehen, die aber natürlich grundsätzlich weiterhin erlaubt sind.

```
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <metadocu>
    <author><personname>Frank Seidel</personname></author>
  </metadocu>
  <declarations>
    <name>Beispielprogramm</name>
    <variables>
      <develdoc>
        <from>Frank Seidel</from>
        <to>Leser der Diplomarbeit</to>
        <message>Nachrichtentext ...</message>
      </develdoc>
      <type>integer</type>
      <var>
        <explanation>
          <summary>Beispielvariable</summary>
        </explanation>
        <name>Integervariable</name>
      </var>
    </variables>
  </declarations>
  <start>
    <explanation>
      <summary>Prozeduraler Teil des Hauptalgorithmus</summary>
    </explanation>
    <sequence> ... </sequence>
  </start>
</xalgo>
```

Listing 10.32: Beispiel für die Anwendung der Dokumentationselemente

Kapitel 11

Umsetzung der Transformationskomponenten

Der Erstellung der beiden Transformationskomponenten kam im praktischen Teil der Arbeit das Hauptaugenmerk zu. Da beide Komponenten Dokumente von einer Quellsprache – hier XAlgo – in eine andere Zielsprache –in diesem Fall Java bzw. C++ – übersetzen, kann man in Zusammenhang mit einem XSLT-Prozessor durchaus von Compilern sprechen.

11.1 XSLT als Transformationsprache

Mit XSLT steht eine universelle (touringmächtige) Transformationsprache in XML bereit. Daher lag es Nahe, auch für die XAlgo-Übersetzung auf diese Technik zurückzugreifen, zumal dies auch nochmals die Mächtigkeit der XML-Familie demonstriert.

Die erzeugten XSLT-Stylesheets kommen dabei im Wesentlichen ohne besondere Erweiterungen und mit dem Funktionsumfang von Version 1.0 aus. Lediglich für den Java-Compiler musste an einer Stelle ein Element der Version 1.2 genutzt werden. Als XSLT-Prozessor kam bei der Entwicklung hauptsächlich das freie Programm „Saxon“¹ in Version 6.5.3 zum Einsatz.

Trotz der vielen Möglichkeiten, die einem XSLT bietet, ist es doch deutlich geworden, dass die Sprache für derartige Übersetzungen nicht gut geeignet ist. So können beispielsweise einmal belegte „Variablen“ nicht mehr verändert werden, weswegen man eigentlich gar nicht von Variablen sprechen dürfte. Auch ist man in der Formulierung von Schleifen sehr beschränkt, da diese in XSLT nur Knotenmengen durchlaufen können.

¹siehe auch „<http://saxon.sourceforge.net/>“ oder Anhang B auf Seite 94

11.2 Besondere Problemstellungen

Bei der Erstellung der Transformationskomponenten haben sich einige besondere Schwierigkeiten herauskristallisiert, die hier eine besondere Erwähnung finden sollen.

11.2.1 Namensumsetzung für Bezeichner

Der Freiraum für vom Programmierer festlegbare Bezeichner ist in den meisten Programmiersprachen sowohl in der Länge als auch im Zeichenvorrat stark beschränkt. XAlgo kennt derartige Einschränkungen hingegen nicht. Um zu verhindern, dass die hier festgelegten Bezeichner in der Zielsprache zu Problemen führen, ist es nötig, eine Abbildung durchzuführen.

Zunächst wurde hierfür eine Abbildung auf einen Namen mit dem Präfix „xalgo_“ und einer errechneten alphanumerischen Fortsetzung vorgesehen, die den Bezeichner eindeutig machte (über die XPath-Funktion² „generate-id()“). Auf diese Weise wäre die Umformung zwar immer eindeutig und korrekt gewesen, jedoch sind die so erzeugten Bezeichner in der Zielsprache für den menschlichen Betrachter eher ungeeignet.

Deshalb wurde die Namensumformung schließlich noch so abgeändert, dass bei üblichen Bezeichnern diese komplett erhalten bleiben bzw. die Umformungen so wenig wie möglich an der Lesbarkeit zehren. Umlaute werden z. B. in die Kombinationen „ae“, „oe“ und „ue“ übersetzt. Nicht in der Transformationstabelle enthaltene Sonderzeichen werden durch einen Unterstrich („_“) ersetzt. Mit dieser Technik ist eine perfekte und eindeutige Abbildung streng genommen nicht mehr möglich, doch erscheinen hier die Vorteile der besseren Lesbarkeit und einer möglichst einfache Transformationsregel doch bedeutender.

11.2.2 MathML-Prozessor

Für die Umsetzung der MathML-Ausdrücke in die übliche Infix-Notation der Zielsprachen wurde ein MathML-Prozessor benötigt. Eine sehr einfache Version hiervon wurde in [Mangano 2003, Seite 136f] gefunden. Nach ein paar Erweiterung – hauptsächlich in der Operatorenumsetzung – stand auf diese Weise ein einfacher MathML-Prozessor zur Verfügung, der dazu auch noch in XSLT vorlag und somit elegant in die bestehenden Transformationskomponenten eingebunden werden konnte.

²siehe „<http://www.w3.org/TR/xpath>“

11.2.3 Abbildung von Namen auf Reihenfolgen

Sowohl die Parameter bei Unterprogrammaufrufen als auch die Dimensionen in Arrays werden in XAlgo nur über Ihre Namen referenziert. Da dies in den Zielsprachen nur über ihre Reihenfolge bei der Deklaration geschieht, muss auch hier eine entsprechende Umsetzung stattfinden. Natürlich liegt es hier nahe, eine Umordnung über eine alphabetische Sortierung vorzunehmen.

Genau dies geschieht letztlich auch, doch lagen hier viele Probleme im Detail. So kann man beispielsweise schon nicht immer darauf zählen, dass alle Elemente auch beim Aufruf für die Sortierung parat stehen, wie etwa bei der Arraygrößenabfrage, die nur eine Dimension entgegennimmt.

Aber vor allem kann man durch die Eigenheiten in einer XSLT-Schleife auf einer sortierten Knotenmenge nicht ohne weiteres feststellen, ob man sich im letzten Durchlauf befindet (z. B. um einen Delimiter in einer Aufzählung zu unterdrücken). Die „last()“-Funktion bezieht sich nämlich stets auf die unsortierte Knotenmenge, wodurch einiger zusätzlicher Aufwand entsteht. Hier wurde dieses Problem z. B. über ein benanntes Template gelöst, das vor dem Schleifeneintritt die gewünschte Knotenmenge fertigsortiert bereitstellt, die in einer Variable erfasst wird. Auf diesem Teilbaum kann dann schließlich die Schleife ablaufen, so dass über die XPath-Funktion „last()“ doch der letzte Schleifendurchlauf ermittelt werden kann.

11.3 Stylesheet für Java

Das XSLT-Stylesheet für die Übersetzung in Java-Quellcode wurde zuerst fertiggestellt. Einige Eigenschaften von Java kamen der Transformation sehr zugute. Etwa der dort vorhandene Garbagecollector oder auch die gute Unterstützung von mehrdimensionalen Arrays vereinfachten – im Vergleich zu C++ – die Erstellung erheblich.

11.3.1 Besondere Entwicklungshürden

Aber natürlich gab es hier auch besondere Probleme, die bewältigt werden mussten.

Die erste große Hürde wartete gleich zu Beginn der Entwicklung. In Java werden Klassen in aller Regel nur in Dateien akzeptiert, die genauso benannt sind wie sie selbst. In XSLT 1.0 existiert allerdings kein Element, mit dem man beeinflussen kann, wie die Ausgabedatei benannt wird. Vielfach ist es sogar einfach üblich, dass das Transformationsergebnis schlicht auf die Standardausgabe erfolgt. Aus diesem Grund kam man nicht umhin, an dieser Stelle das <document>-Element mit dem href-Attribut aus XSLT 1.2 einzusetzen, das von Saxon auch einwandfrei unterstützt wird.

11.3.2 Erreichtes Ergebnis

Die Transformationskomponente konnte auf Anhieb mit einer vollständigen Unterstützung von XAlgo erstellt werden. Nur die Typüberprüfung und einige unbedeutende Elemente bzw. deren Attribute werden noch nicht komplett umgesetzt. So wird z. B. bei den Umgebungen für die Ein-/Ausgabe das `file`-Attribut bisher nicht ausgewertet und die Parameter der Kommandozeile können noch nicht verwendet werden (siehe auch Kapitel 14.2 auf Seite 81).

11.4 Stylesheet für C++

Für die Entwicklung des Übersetzers für C++ wurde das Stylesheet für Java als Grundlage herangezogen. Neben den Veränderungen für die offensichtlichen Unterschiede zu Java mussten einige tiefer gehende Änderungen und Erweiterungen vorgenommen werden.

11.4.1 Besondere Entwicklungshürden

Da C++ keinen Garbagecollector vorsieht, musste durch den Übersetzer zusätzlich dafür Sorge getragen werden, dass alle im Speicher explizit angelegten Objekte wieder gelöscht werden.

Leider ist auch die Behandlung von mehrdimensionalen Arrays lange nicht so umfangreich wie in Java. Es existiert beispielsweise keine Möglichkeit, die Größe eines Arrays zur Laufzeit zu erfragen. Um das Problem elegant zu lösen, wurde von der STL³-Klasse „vector“ Gebrauch gemacht. Hierbei werden Instanzen automatisch entsprechend den angelegten Arrays geschachtelt und angesprochen.

11.4.2 Erreichtes Ergebnis

Schließlich war es noch möglich, auch für die Transformationskomponente in C++ eine nahezu vollständige Unterstützung für XAlgo zu erzielen. Die beiden Compiler sind also in ihrem Entwicklungsstand vollkommen gleichwertig.

³Standard Template Library von C++

Kapitel 12

Beispielalgorithmen

Anhand einiger etwas ausführlicheren Beispiele soll die Verwendung von XAlgo noch weiter verdeutlicht werden. Dabei wurde jedoch größtenteils auf die Benutzung der Dokumentationselemente verzichtet, da sonst die Dokumente sehr groß wären und dadurch in gedruckter Form noch unübersichtlicher werden würden.

12.1 Zähler

Das hier vorgestellte, sehr einfache Programm ist eine Zählschleife samt Ausgabe.

```
<?xml version="1.0" ?>
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <declarations>
    <name>countertest</name>
5    <constants>
      <type>integer</type>
      <const>
        <name>limit</name>
        <value>
10      <math xmlns="http://www.w3.org/1998/Math/MathML">
          <apply><cn>10</cn></apply>
          </math>
        </value>
      </const>
15    </constants>
    <variables>
      <type>integer</type>
      <var><name>counter</name></var>
```

```

    </variables>
20 </declarations>
    <start>
        <loop name="Hauptschleife">
            <conditions>
                <loop-init-statement>
25         <assign>
                <source><directnative>0</directnative></source>
                <target>counter</target>
            </assign>
        </loop-init-statement>
30     <breakcondition>
            <math xmlns="http://www.w3.org/1998/Math/MathML">
                <apply><gt /><ci>counter</ci><ci>limit</ci></apply>
            </math>
        </breakcondition>
35     <loop-update-statement>
            <assign>
                <source>
40         <math xmlns="http://www.w3.org/1998/Math/MathML">
                <apply><plus /><cn>1</cn><ci>counter</ci>
                </apply>
            </math>
            </source>
                <target>counter</target>
            </assign>
45     </loop-update-statement>
        </conditions>
        <body>
            <output><text>Zaehler ist </text></output>
            <output>counter</output>
50     <checkloopcondition />
        </body>
    </loop>
</start>
</xalgo>

```

Listing 12.1: Beispielalgorithmus mit Zählschleife

Das `<xalgo>`-Wurzelement erstreckt sich von den Zeile 2 bis 54, fasst also den gesamten Algorithmus ein. Eine Variable, eine Konstante und der Name des Al-

gorithmus werden im Deklarationsteil in den Zeilen 3 - 20 festgelegt. Der Einsatz einer explizit deklarierten Konstante wäre eigentlich nicht nötig, stellt hier aber die zu Variablen sehr ähnliche Deklaration gut heraus. Weil in diesem speziellen Fall nur so wenige Variablen und Konstanten benötigt werden, kommt die Gruppierungswirkung von `<constants>` und `<variables>` leider nicht zur Geltung.

Der prozedurale Teil liegt vollständig in dem einen `<start>`-Element (Zeile 21 bis 53), enthält nur eine Schleifenanweisung und braucht daher auch keine `<sequence>`-Umgebung. Das `name`-Attribut der Schleife hat keinerlei inhaltliche Bedeutung, es dient bei einer Übersetzung lediglich als Orientierungshilfe, wo dieses Konstrukt im Ergebnis (Listing 12.2 auf der nächsten Seite) wiederzufinden ist.

Der Schleifenrumpf in den Zeilen 47 bis 51 ist vergleichsweise übersichtlich. Er enthält lediglich zwei Ausgabeanweisungen für den Text und die Zählvariable, sowie die Festlegung, wo die Schleifenabbruchsbedingung geprüft werden soll. Der Schleifenkopf in der `<conditions>`-Umgebung (Zeile 23 bis 46) beinhaltet ein `<loop-init-statement>`-Element zur Initialisierung des Zählers mit dem Wert 0, ein `<breakcondition>`-Element für die Abbruchbedingung in MathML und schließlich das `<loop-update-statement>`-Element für die Fortzählung der Variable „counter“. Die beiden Anweisungsstrukturen innerhalb des Kopfes enthalten jeweils eine `<assign>`-Anweisung, wobei erstere nur ein einziges mal zu Beginn der Schleife – vor allen Anweisungen des Schleifenrumpfes – zur Ausführung kommt. Die `<loop-update-statement>`-Anweisung kommt hingegen stets nach allen Anweisungen im Rumpf zur Ausführung, sofern die Schleife nicht vorher verlassen wurde. Nach allen Umformungen (über Stylesheet und nachgeschaltetem Beautifier) erhält man das Listing 12.2 auf der nächsten Seite als Ergebnis.

Betrachtet man alleine schon den Umfang des Schleifenkopfes für eine derart einfache Zählschleife, wird deutlich, dass XAlgo-Dokumente stark dazu neigen, besonders umfangreich zu werden. Dies liegt aber auch in dem Einsatz von XML im Allgemeinen und MathML im Speziellen für die meisten Ausdrücke begründet. Der Einsatz von speziellen XML-Editoren oder -Entwicklungsumgebungen ist also sehr zu empfehlen. Bei dieser Arbeit kam der freie jEdit-Editor¹ zum Einsatz, für den zahlreiche und sehr hilfreiche XML-Erweiterungen bereitstehen.

¹siehe „<http://www.jedit.org>“

```

import java.util.*;
import java.io.*;

public class countertest {
    public static void main(String[] args) {
        final int limit= 10;
        int counter;
        { //XALGO LOOP NAME: Hauptschleife
            // XALGO ASSIGN
            counter = 0;
            do {
                System.out.println( "Zaehler_List_" );
                System.out.println( counter );
                if (counter > limit) break;
                // XALGO ASSIGN
                counter = 1 + counter;
            } while (true) ;
        }
    }
}

```

Listing 12.2: Aus XAlgo-Listing 12.1 generierter Javaquellcode

12.2 Quicksort

Da eine Verwirklichung des Quicksortalgorithmus in XAlgo selbst ohne eingebaute Dokumentation äußerst umfangreich ist, wurden im Listing 12.3 für das Sprachverständnis unbedeutendere Stellen herausgenommen. Den vollständigen Code findet man zudem auch auf dem Datenträger in Anhang B auf Seite 94. Im Folgenden soll zudem nicht die Funktionsweise von Quicksort erklärt, sondern lediglich der mögliche Einsatz der XAlgo-Sprachelemente aufgezeigt werden.

```

<?xml version="1.0" ?>
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0"
        xmlns:math="http://www.w3.org/1998/Math/MathML" >
4  <declarations>
        <name>quicksortdemo</name>
        <variables name="demoarrays">
            <type>
                <arraytype>

```

```

9           <elementtype>integer</elementtype>
           <dimension>
             <name>x</name>
             <length>8</length>
           </dimension>
14        </arraytype>
        </type>
        <var><name>unsorted</name></var>
        <var><name>sorted</name></var>
    </variables>
19    ...
</declarations>
<definitions>
  <subalgo>
    <declarations>
24      <name>sort</name>
      <parameters>
        <param>
          <name>array</name>
          <type>
29            <arraytype>
              <elementtype>integer</elementtype>
              <dimension>
                <name>x</name>
              </dimension>
34            </arraytype>
          </type>
        </param>
        <param>
          <name>lo</name>
39          <type>integer</type>
        </param>
        <param>
          <name>hi</name>
          <type>integer</type>
44        </param>
      </parameters>
      ...
    <returns>
      <type>

```

```

49         <arraytype>
           <elementtype>integer</elementtype>
           <dimension><name>x</name></dimension>
         </arraytype>
       </type>
54     </returns>
  </declarations>
  <start>
    <assign name="get_pivot_element">
      <source>
59         <array>
           <name>array</name>
           <dimension>
             <name>x</name>
             <index>
64                 <math xmlns="http://www.w3.org/1998/Math/MathML">
                   <apply>
                     <div />
                     <apply>
                       <plus />
69                       <ci>lo</ci>
                       <ci>hi</ci>
                     </apply>
                     <cn>2</cn>
                   </apply>
74                 </math>
             </index>
           </dimension>
         </array>
       </source>
79     <target>pivot</target>
    </assign>
    ...
    <return>array</return>
  </start>
84 </subalgo>
</definitions>
<start>
  <sequence>
    <explanation><summary>fill array</summary></explanation>

```

```
89      ...
      <callsub name="sort_the_unsorted_array">
          <name>sort</name>
          <parameters>
            <param>
14      <name>array</name>
            <value>unsorted</value>
            </param>
            ...
          </parameters>
99      <returnvaluetarget>sorted</returnvaluetarget>
      </callsub>
      <loop>
          <explanation><summary>prints sorted array</summary>
          </explanation>
104     ...
      </loop>
    </sequence>
  </start>
</xalgo>
```

Listing 12.3: Beispielalgorithmus für Quicksort

Die Zeilen 4 bis 20 zeigen einen Teil der Deklarationen des Hauptalgorithmus. Der Name wurde auf „quicksort“ festgelegt. Besonders interessant ist dort die Spezifikation des Arraytypes (in den Zeilen 8 bis 14) und dessen einfache Nutzung für die zwei Arrayvariablen „unsorted“ und „sorted“ (Zeile 16 und 17). Der Datentyp der Arrayelemente ist dabei über „integer“ (in `<elementtype>`) auf ganze Zahlen festgelegt. Das Array hat nur eine Dimension mit dem Namen „x“ und einer Elementanzahl von acht.

In der `<definitions>`-Umgebung findet man einen Subalgorithmus mit dem Namen „sort“ (Zeile 22 bis 84). Dieser nimmt beim Aufruf die drei Parameter „array“, „lo“ und „hi“ entgegen. Der „array“-Parameter nimmt alle Arrays mit einer Dimension namens „x“ und dem Elementtyp „integer“ (zur Sortierung) entgegen. Sowohl „lo“ als auch „hi“ akzeptieren Ganzzahlen für die Index-Unter- und Obergrenze des zu sortierenden Bereichs. Für die Werterückgabe ist in den Zeilen 47 bis 54 der gleiche Arraytyp angegeben worden, wie für den „array“-Parameter. Der prozedurale Teil des Subalgorithmus (von Zeile 56 bis 83) erledigt (rekursiv über sog. „divide-and-conquer“) die eigentliche Sortierung. Besondere Aufmerksamkeit verdient dort zu Beginn der MathML-Ausdruck zur Gewinnung des Pivot-Elements. In der `<assign>`-Anweisung (von Zeile 57 bis 80) wird der Wert eines Arrayelements, aus dem bei Aufruf erhaltenem Array, an die Integervariable

„pivot“ (Zeile 79) zugewiesen. Der Index wird dabei über einen mathematischen Ausdruck in MathML gebildet, der diesmal zwei `<apply>`-Umgebungen enthält. Zunächst wird von ihnen die innere ausgewertet und mit deren Ergebnis die äußere berechnet. Schließlich wird in Zeile 82 die Rückkehr mit dem sortierten Array als Rückgabewert ausgelöst.

Der prozedurale Teil des Hauptalgorithmus beginnt in Zeile 86. Dort wird in einer Sequenz zunächst das Array „unsorted“ gefüllt (in Listing nur durch Dokumentationselement angedeutet). Mit `<callsub>` wird dann die Sortierfunktion aufgerufen und an sie das unsortierte Feld übergeben (Zeile 90 bis 100). Dabei wird in Zeile 99 der Rückgabewert aufgefangen und im Array „sorted“ abgelegt. Die abschließende Schleife (in den Zeilen 101 bis 105) enthält lediglich Anweisungen zur Ausgabe des sortierten Arrays und enthält im Vergleich zum Beispiel aus Abschnitt 12.1 nichts neues.

12.3 Klassendemo

In Listing 12.4 findet man ein komplettes Beispiel zur Verwendung der objektorientierten Sprachelemente von XAlgo.

```
<?xml version="1.0" ?>
<xalgo xmlns="http://www.f-seidel.de/xmlns/2004/XAlgo/1.0">
  <declarations>
    <name>classdemo1</name>
5    <classes>
      <class>
        <name>vehicle</name>
        <extensions>
          <variables><type>integer</type>
10          <var><name>weight</name></var>
          <var><name>maxspeed</name></var>
          </variables>
          <subalgo>
            <declarations>
15            <name>start</name>
            <start><null/></start>
            </declarations>
          </subalgo>
          </extensions>
20        </class>
        <class>
          <name>car</name>
```

```

        <baseclass>vehicle</baseclass>
        <extensions>
25         <variables><type>integer</type>
            <var><name>power</name></var>
        </variables>
        </extensions>
    </class>
30 </classes>
    <objects>
        <type>vehicle</type>
        <object>
            <name>traktor</name>
35         <instancetype>vehicle</instancetype>
        </object>
        <object>
            <name>ferrari</name>
            <instancetype>car</instancetype>
40         </object>
    </objects>
</declarations>
<start>
    <sequence>
45     <callsub>
        <name>
            <elementpath><name>ferrari</name>
                <sub><name>start</name></sub>
            </elementpath>
50         </name>
        </callsub>
    </sequence>
</start>
</xalgo>

```

Listing 12.4: Beispielalgorithmus zur Demonstration von Klassen

In diesem Beispiel nimmt die Deklaration der Klassen und Objekte die meisten Quellcodezeilen in Anspruch (Zeile 3 bis 42). Die vierte Zeile legt mit `<name>` den Namen des Algorithmus auf „classdemo1“ fest. Darauf folgt die Spezifizierung der Klassenstrukturen in den Zeilen 5 bis 30. Die erste Klasse (in den Zeilen 6 bis 20) erhält den Namen „vehicle“ (in Zeile 7) und enthält die zwei Integervariablen „weight“ und „maxspeed“, sowie den Subalgorithmus mit dem Namen „start“,

der einen funktionslosen prozeduralen Teil aufweist.

Die zweite Klasse „car“ (Zeile 22) erbt von der vorherigen alle Elemente und Subalgorithmen (Zeile 23). Außerdem wird noch die Integervariable „power“ zu den Klassenelementen hinzugefügt (in den Zeilen 24 bis 28).

In der <objects>-Umgebung (von Zeile 31 bis 40) werden zwei Klasseninstanzen spezifiziert. Der Grundtyp ist dabei für beide die Basisklasse „vehicle“. Das erste Objekt (Zeile 33 bis 36) „traktor“ erhält dabei wirklich auch eine Instanz von „vehicle“. Das zweite Objekt „ferrari“ hingegen wird mit der Kindklasse „car“ initialisiert.

Die <start>-Umgebung des Hauptalgorithmus zeigt lediglich den Aufruf des Subalgorithmus „start“, den das Objekt „ferrari“ über die Klasse „vehicle“ geerbt hat. Die Referenzierung des Subalgorithmus muss dabei über die <elementpath>-Umgebung erfolgen (Zeile 47 bis 49).

```
import java.util.*;
import java.io.*;

public class classdemo1 {
    public static void main(String[] args) {
        // XALGO CLASSES (Name: trafficobjects)
        public class vehicle{
            int weight;
            int maxspeed;
            static public void start() { }
        }
        public class car extends vehicle{
            int power;
        }
        vehicle traktor = new vehicle();
        vehicle ferrari = new car();
        { //XALGO.SEQUENCE NAME:
            // XALGO CALLSUB: here a subalgorithm is called
            ferrari.start ( );
        }
    }
}
```

Listing 12.5: Aus XAlgo-Listing 12.4 generierter Javaquellcode

```
using namespace std;
#include <vector>
#include <iostream>
#include <string>
#include <ctime>

// XALGO CLASSES (Name: trafficobjects)
class vehicle{
public:
    int weight;
    int maxspeed;
    void start() {

    }
};
class car : public vehicle{
public:
    int power;
};
int main(int argc, char* argv[]) {
    vehicle* traktor = new vehicle();
    vehicle* ferrari = new car();
    { //XALGO.SEQUENCE NAME:
        // XALGO CALLSUB: here a subalgorithm is called
        ferrari->start ( );
    }
    delete traktor;
    delete ferrari;
    return 0;
}
```

Listing 12.6: Aus XAlgo-Listing 12.4 generierter C++-Quellcode

Kapitel 13

Benutzung von Schema und Stylesheet

In diesem Kapitel wird erklärt, wie die im Rahmen dieser Arbeit erstellten Schemas und Transformationskomponenten benutzt werden können.

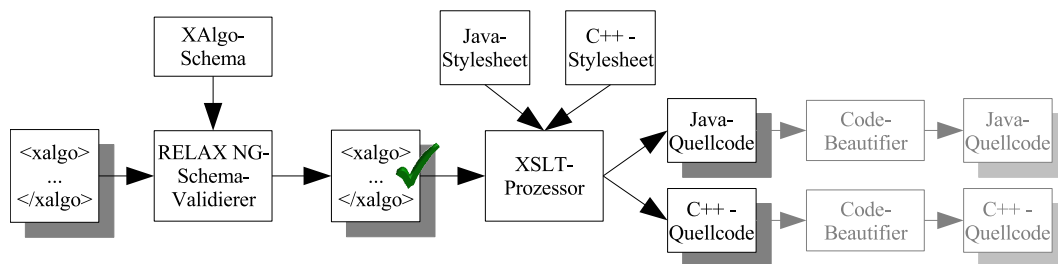


Abbildung 13.1: Erweitertes Verarbeitungsschema für XAlgo-Dokumente

13.1 RELAX NG-Schemas für XAlgo

Die beiden Versionen des XAlgo RELAX NG-Schemas (siehe Kapitel 9.2.2 auf Seite 29) liegen in den Dateien „XAlgo/Schemas/xalgo-strict.rng“¹ und „XAlgo/Schemas/xalgo-open.rng“ auf der CD aus Anhang B. Sie werden zusammen mit einem RELAX NG-fähigen Schemavalidierer gebraucht, um XAlgo-Dokumente auf ihre strukturelle Korrektheit hin zu überprüfen.

Für RELAX NG sind z. B. über das Internet zahlreiche Validierer und Validierungsgeneratoren erhältlich. Auf der RELAX NG-Website² kann man eine Über-

¹relativ zum Wurzelverzeichnis des Datenträgers

²auf „<http://relaxng.org>“ bzw. „<http://relaxng.org/#software>“

sicht der bekanntesten Softwareprojekte zu dieser Schemasprache ansehen. Im Rahmen dieser Arbeit wurden die beiden Schemavalidierer „Jing“³ und „MSV“⁴ eingesetzt. Mit Tools wie „Bali“⁵ wäre es auch möglich, aus dem Schema einen eigenständigen Validator zu generieren, doch verliert man so wieder ein Stück Flexibilität, weshalb hier kein Gebrauch von solchen Validierungsgeneratoren gemacht wurde. Im praktischen Einsatz sind Jing und MSV bezüglich RELAX NG nach den im Rahmen der Arbeit gemachten Erfahrungen etwa gleich gut geeignet, allerdings bietet MSV mehr Konfigurationsmöglichkeiten und auch (optional) ausführlichere Fehlermeldungen. Beiden gemein ist jedoch die schlechte Erkennung der Fehlerursache. Häufig werden Fehler nur an Stellen gemeldet, die scheinbar mit dem eigentlichen Problem nichts zu tun haben.

13.1.1 Jing

Jing liegt in der verwendeten Version vom 19.06.2003 dem Datenträger dieser Arbeit bei. Um den Validierer verwenden zu können, muss auf dem System eine Java-Laufzeitumgebung installiert sein. Java Version 1.4 wird dabei empfohlen, allerdings soll es selbst auf Umgebungen mit Version 1.1 noch (eingeschränkt) laufen. Da bei Jing schon ein XML-Parser mitgeliefert wird (auf Apache Xerces basierend), braucht man für die Validierung keine weiteren Softwarepakete installieren.

Entpackt man das ZIP-Archiv, werden u. a. eine Reihe von Verzeichnissen angelegt. Im „bin“-Verzeichnis findet man die Datei „jing.jar“, die man auf einer Kommandozeileingabe nun mit

```
java -jar <Jingverzeichnis>/bin/jing.jar <Schemadatei> <Dokument>
```

aufrufen kann (auf Windows-Systemen muss dabei natürlich statt „/“ stets der Verzeichnisdelimiter „\“ verwendet werden).

Liegen beispielsweise ein XAlgo-Dokument „algorithmus.xml“ und das Schema „xalgo-strict.rng“ im aktuellen Arbeitsverzeichnis, so sieht eine normale Validierung mit Jing z. B. wie folgt aus:

```
java -jar bin/jing.jar xalgo-strict.rng algorithmus.xml
```

Werden vom Programm nach dem Aufruf keine Meldungen ausgegeben, so ist das Dokument korrekt bzw. „gültig“ nach dem angegebenen Schema strukturiert.

³Java-basierter Open-Source Schemavalidator von James Clark für RELAX NG, Schema-
tron, XML Schema u.a.; siehe „<http://www.thaiopensource.com/relaxng/jing.html>“

⁴Sun Multi-Schema XML Validator (auch Java-basiert) für RELAX NG, DTD, TREX und
XML Schema; siehe „<http://www.sun.com/software/xml/developers/multischema/>“

⁵siehe „<http://www.kohsuke.org/relaxng/bali/doc/>“

Zwischen der Angabe der Schemadatei und „jing.jar“ können eine Reihe von optionalen Parametern genutzt werden, die in Tabelle 13.1 kurz erläutert sind.

Parameter	Beschreibung
-c	Schema verwendet spezielle kompakte Syntax (kein XML mehr)
-e enc	Verwendet die Codierung enc zum Lesen des Schemas
-f	„ <i>feasibly valid</i> “: Testet, ob Dokument gültig wäre, wenn alle Elemente und Attribute nur optional wären.
-i	Deaktiviert die Überprüfung von ID, IDREF und IDREFS
-t	Gibt die gebrauchte Zeit mit aus

Tabelle 13.1: Optionale Kommandozeilenparameter für Jing

13.1.2 MSV

MSV liegt der Arbeit in der Version 20030225 bei. Auch für diesen Validator wird eine Java-Laufzeitumgebung benötigt (Versionsabhängigkeiten werden bei der Software nicht erwähnt; auf dem Entwicklungssystem für diese Arbeit kam Version 1.4.2 zum Einsatz, die einwandfrei funktionierte). Auch MSV enthält – wie Jing – einen XML-Parser, ist also ebenfalls nicht von anderer installierter Software abhängig (von Java abgesehen).

Durch die Dekomprimierung des ZIP-Archivs „msv.20030225.zip“ ist im entstandenen „msv-20030225“-Verzeichnis die Datei „msv.jar“ zu finden. Diese ist auf gleiche Weise zu benutzen wie bei Jing:

```
java -jar msv.jar <Schemadatei> <XML-Dokument>
```

Das Beispielkommando von Jing würde bei MSV so lauten:

```
java -jar msv.jar xalgo-strict.rng algorithmus.xml
```

War das Dokument fehlerfrei gibt MSV die folgenden Zeilen aus:

```
start parsing a grammar.
validating algorithmus.xml
the document is valid.
```

Auch MSV kennt einige Optionen, die man nach „msv.jar“ angeben kann (siehe Tabelle 13.2 auf der nächsten Seite). Ruft man MSV ohne Angabe von Dateien auf, so kommt auch eine kurze Übersicht mit englischen Kurzerklärungen.

Parameter	Beschreibung
-warning	Zeigt auch Warnungen an
-standalone	Verhindert die Verwendung externer Ressourcen
-strict	Auch das Schema wird einer Prüfung unterzogen
-dump	Gibt Schema als Grammatik aus (keine Validierung)
-debug	Informationen zur Fehlersuche werden mit ausgegeben
-maxerror	Erzeugt maximal viele Fehler (auch evtl. falsche)
-catalog catfile	Verwendet Katalogdatei catfile zur Auflösung externer Entities
-version	Gibt nur die Versionsnummer aus
-locale lang	Setzt die Nachrichtensprache auf lang (z. B. „de“, „en“ etc.)
-xerces	verwendet den Parser Xerces-J
-crimson	verwendet den Parser Crimson
-oraclev2	verwendet den Parser OracleV2

Tabelle 13.2: Optionale Kommandozeilenparameter für MSV

13.2 Transformationskomponenten

Für die Transformationskomponenten wurde „Saxon“ als freien XSLT-Prozessor verwendet. Auch hier befindet sich die im Rahmen der Arbeit verwendete Version auf dem beigegeführten Datenträger. Entpackt man die ZIP-Datei, erhält man in dem Extraktionsverzeichnis die Datei „saxon.jar“, die alleine lauffähig ist. Legt man diese in den Klassensuchpfad⁶ der Laufzeitumgebung, kann man über den Befehl

```
java com.icl.saxon.StyleSheet
Saxon aufrufen.
```

Dabei erwartet das Programm nach den optionalen Parametern zuerst den Dateinamen des Quelldokumentes und danach die Datei mit dem XSLT-Stylesheet. Wenn man über die Programmparameter oder im Stylesheet nichts anderes angibt, wird das Ergebnis der Transformation auf die Standardausgabe geschrieben. Es ist also ratsam, entweder eine Ausgabeumleitung oder die Optionen von Saxon entsprechend zu nutzen. In der Tabelle 13.3 auf der nächsten Seite sind

⁶unter Linux z. B. mit dem Befehl „export CLASSPATH=/tmp/saxon.jar:\$CLASSPATH“, wenn die Datei „saxon.jar“ im Verzeichnis „/tmp/“ liegt

die wichtigsten Programmoptionen aufgelistet. Die Transformationskomponenten zur Übersetzung der XAlgo-Dokumente in Java bzw. C++ liegen auf dem Datenträger im Verzeichnis „XAlgo/Stylesheets/“. Zu beachten ist auch, dass sich beide Stylesheets des gleichen MathML-Prozessors aus der Datei „mathmlprocessor.xml“ bedienen. Dieses Dokument muss sich bei der Anwendung einer der Transformationskomponenten immer im gleichen Verzeichnis wie diese befinden.

Parameter	Beschreibung
-o ausgabedatei	Statt auf die Standardausgabe wird das Transformationsergebnis in die Datei ausgabedatei geschrieben.
-u	Dateiname sind URL-Adressen
-t	Version und Zeitverbrauch werden angezeigt.
-a	Verwendet das im Quelldokument angegebene Stylesheet

Tabelle 13.3: Optionale Kommandozeilenparameter für Saxon

13.2.1 XAlgo nach Java

Das XSLT-Stylesheet für die Transformation von XAlgo-Dokumenten in Java heißt „xalgo2java.xsl“. Dieses kann wie jedes gewöhnliche Stylesheet auf eine XAlgo-XML-Datei angewendet werden. Allerdings gilt hier zu beachten, dass beim Aufruf von Saxon keine Ausgabedatei angegeben werden sollte, da diese durch die Transformationskomponente ermittelt und genutzt wird. In Java müssen Klassen in .java-Dateien abgelegt sein, die den gleichen Namen tragen wie die jeweilige Klasse. Ein typischer Aufruf könnte z. B. so aussehen:

```
java com.icl.saxon.StyleSheet algorithmus.xml xalgo2java.xsl
```

13.2.2 XAlgo nach C++

Die Verwendung des Stylesheets für C++ passiert ganz analog. Die Datei mit dem XSLT-Stylesheet heißt in diesem Falle „xalgo2c++.xsl“. Da bei C++ keine zwingenden Bestimmungen für den Dateinamen existieren, kann und muss der Benutzer hier selbst die Ausgabedatei angeben. Dies kann beispielsweise so passieren:

```
java com.icl.saxon.StyleSheet -o out.cpp algorithmus.xml xalgo2c++.xsl
```

13.3 Beautifier

Der mit den Stylesheets erzeugte Code ist zwar eigentlich schon fertig einsetzbar und auch gut lesbar, es fehlt ihm aber noch die übliche Einrückung der Programmzeilen, zum einfacheren Erkennen zusammengehöriger Strukturen. Ein Beispiel hierfür ist das Listing 13.1.

```
import java.util.*;
import java.io.*;

public class countertest {
public static void main(String [] args) {
final int limit= 10;
int counter;
{ //XALGO LOOP NAME: Hauptschleife
// XALGO ASSIGN
counter = 0;
do {
System.out.println( " Zaehler_list_" );
System.out.println( counter );
if (counter > limit) break;
// XALGO ASSIGN
counter = 1 + counter;
} while (true) ;
}
}
}
```

Listing 13.1: Generierter Javaquellcode ohne nachgeschalteten Beautifier

Zur besseren Übersichtlichkeit des Quellcodes wurde in der vorliegende Arbeit der freie Code-Beautifier AStyle⁷ eingesetzt, der C, C++, C# und Java-Quellcode neu formatiert. Das Programm ist ebenfalls mit auf der CD in Anhang B enthalten. Die Windows-Variante enthält bereits eine ausführbare EXE-Datei. Für Linux⁸ muss man hingegen zunächst das beigefügte Makefile benutzen (über den Aufruf „make“ im entpackten Verzeichnis), um ein ausführbares Programm zu erhalten.

Um Javaquellcode zu verschönern, ruft man das Programm mit der Option „-j“

⁷für „Artistic Style“; siehe „<http://astyle.sourceforge.net>“

⁸siehe „<http://www.linux.org>“

und danach mit dem Dateinamen auf (z. B. `astyle -j countertest.java`). Das Programm sichert die Veränderungen wieder in dieselbe Datei zurück, legt aber sicherheitshalber eine neue Datei mit der zusätzlichen Endung „.orig“ an, die die ursprüngliche Fassung enthält. Für C++ benutzt man hingegen die Option „-c“ (z. B. `astyle -c out.cpp`).

13.4 Beispiel

Abschließend wird an einem vollständigen Beispiel noch einmal eine mögliche komplette Abfolge der Bearbeitungsschritte gezeigt. Es soll dabei das Beispielprogramm aus Listing 12.1 (auf Seite 62)⁹ in ein funktionsfähiges Javaprogramm umgesetzt werden. Die verwendeten Kommandos und Verzeichnisnamen beziehen sich exemplarisch auf eine Linux-Umgebung, können aber mit marginalen Änderungen auch auf Windows-System angewendet werden.

Dafür gehen wir davon aus, dass Jing zum Einsatz kommt und dessen bin-Verzeichnis unter `/tmp/jing/bin` liegt. Auch Saxon sollte installiert und über `java com.icl.saxon.StyleSheet` ansprechbar sein. Die ausführbare Datei von AStyle kopiert man am besten in ein Verzeichnis, das im Suchpfad für ausführbare Programme liegt (unter Linux z. B. unter `/usr/local/bin/`). Hat man die Stylesheets „xalgo2java.xsl“ und „xalgo2c++.xsl“, den MathML-Prozessor aus „mathmlprocessor.xsl“, die zwei Schemaversionen „xalgo-strict.rng“ und „xalgo-open.rng“, sowie das XAlgo-Dokument (hier z. B. „counter.xml“) im Verzeichnis `/tmp/xalgo/` abgelegt, sind die Vorbereitungen fast abgeschlossen. Um die Befehle etwas abkürzen zu können ist es vorteilhaft, wenn man mit „`cd /tmp/xalgo`“¹⁰ das Arbeitsverzeichnis wechselt.

13.4.1 Validierung des XAlgo-Dokumentes

Zuerst wird das XAlgo-Dokument „counter.xml“ auf seine (strukturelle) Korrektheit mit Jing und dem strict-Schema überprüft. Das open-Schema ist hier nicht nötig, da außer MathML keine fremden XML-Sprachen in der Datei verwendet werden.

```
java -jar /tmp/jing/bin/jing.jar xalgo-strict.rng counter.xml
```

Werden keine Fehlermeldungen ausgegeben, gilt das Dokument als wohlgeformt und gültig.

⁹die Beispielprogramme finden sich auch auf dem beigefügten Datenträger

¹⁰unter Windows müssen die Pfade jeweils natürlich entsprechend angepasst werden

13.4.2 Übersetzung in Java

War die Überprüfung des XAlgo-Dokuments erfolgreich, kann es nun mit Hilfe von Saxon und dem Stylesheet in „xalgo2java.xml“ transformiert werden.

```
java com.icl.saxon.StyleSheet counter.xml xalgo2java.xml
```

Durch dieses Kommando wurde die Datei „countertest.java“ erzeugt. Der Inhalt müsste exakt wie in Listing 13.1 (auf Seite 78) aussehen.

13.4.3 Optionale Umformatierung

Je nach der weiteren Verwendung bietet es sich evtl. an, den erzeugten Quellcode über AStyle etwas übersichtlicher zu formatieren. Dieser Schritt ist aber zur Erzeugung eines lauffähigen Javaprogrammes nicht notwendig.

```
astyle -j countertest.java
```

Dieser Aufruf hat nun den Inhalt der Datei so verändert, dass er wie in Listing 12.2 (auf Seite 65) anzutreffen ist.

13.4.4 Compilierung mit javac

Für den letzten Schritt benötigt man schließlich auch noch eine installierte Java-Entwicklungsumgebung samt dem Javacompiler `javac`.

```
javac countertest.java
```

Dieser Befehl erzeugt die Datei „countertest.class“, die nun einfach mit dem Befehl

```
java countertest
```

zur Ausführung gebracht werden kann. Dieser Aufruf ergibt nun schließlich die gewünschten Ausgaben (Zählung von 0 bis 11):

```
Zaehler ist  
0  
Zaehler ist  
1  
...  
Zaehler ist  
11
```

Kapitel 14

Offene Ergänzungen

Hier sollen noch mögliche Ergänzungen und Erweiterungen aufgeführt werden, die im Rahmen dieser Arbeit nicht mehr realisiert werden konnten.

14.1 Schemas

Das Schema ist vollständig und spiegelt den gesamten entworfenen Sprachumfang und dessen Struktur wieder. Sogar eine mögliche Erweiterung mit den Arrayliteralen (`<arrayliteral>`) ist bereits eingearbeitet. Bei der Erstellung wurden viele Regeln aus [Vlist 2004] herangezogen, um die Schematas möglichst gut erweiterbar zu halten und dennoch möglichst exakte Prüfungen zu ermöglichen (daher auch die zwei aufeinander aufbauenden Versionen).

14.2 Transformationskomponenten

Da sich die beiden XSLT-Stylesheets zur Transformation der XAlgo-Dateien in Java bzw. C++ tatsächlich auf gleichem Entwicklungsniveau befinden, gelten die hier erwähnten Punkte für beide Fassungen in gleichem Maße.

14.2.1 Typüberprüfung

Das Einbringen einer vollständigen Typüberprüfung wäre im Rahmen der Arbeit (auch zeitlich) nicht zu verwirklichen gewesen. Aus diesem Grund fällt dieser Punkt unter die Rubrik „*future work*“.

14.2.2 Vollständige MathML-Unterstützung

Augenblicklich enthält der verwendete MathML-Prozessor nur eine sehr rudimentäre Unterstützung des Content-Modells von MathML. Diese dürfte für einfache Aufgaben reichen und ist zudem in der Datei „mathmlprocessor.xml“¹ einfach und zentral erweiterbar.

14.2.3 Kommandozeilenparameter

Eine Möglichkeit im Hauptalgorithmus auf die Aufrufparameter zu reagieren (über die <parameters>-Umgebungen in den Deklarationen) wäre zwar wünschenswert, spielt aber angesichts des Entwurfszieles eher eine untergeordnete Rolle.

14.2.4 file-Attribut bei Ein- und Ausgaben

Zwar kann man ohne die Unterstützung des file-Attributs keine Ein- und Ausgaben mit Dateien vornehmen, doch ist es zumindest möglich, dieses Vorhaben mit den Dokumentationselementen zu hinterlegen und dann einfach in den erzeugten Programmiersprachencode per Hand einzuarbeiten.

14.2.5 Code-Säuberung

Die Entwicklung der XSLT-Stylesheet war eine äußerst anspruchsvolle Aufgabe, die bis in die letzten Tage dieser Arbeit noch Anpassungen, Ergänzungen und Fehlerkorrekturen verlangte. Aus diesem Grunde sehen die Dateien für Außenstehende evtl. sehr unübersichtlich und unaufgeräumt aus. Nach den durchgeführten Tests und Versuchen mit den Stylesheets scheinen sie aber in einem gut funktionierenden und stabilen Zustand zu sein.

¹auf dem Datenträger im Verzeichnis „XAlgo/Stylesheets/“

Kapitel 15

Schlusswort

In dieser Arbeit habe ich den Entwurf, die Verwendung und die theoretischen Bezüge der neuen XML-Sprache XAlgo aufgezeigt. Wie geplant habe ich dabei ein Validierungsschema und zwei Transformationskomponenten für die Übersetzung in die Programmiersprachen Java und C++ erstellt. Alle hierfür verwendeten Techniken basieren ebenfalls auf XML und stehen jedem gewillten Benutzer zur freien Verwendung z. B. über das Internet zur Verfügung.

Auch wenn zu XAlgo noch eine Reihe von Ergänzungen möglich und sinnvoll sind, so konnte ich doch alle vereinbarten Ziele erfüllen.

Als Ergebnis steht nun eine vollständig einsatzfähige bzw. benutzbare XML-Sprache zur Formulierung programmiersprachenunabhängiger Algorithmen bereit. Der so gewonnene Ansatz ist meiner Meinung nach durchaus vielversprechend und könnte z. B. für Algorithmen-Archive, Lehrbücher oder auch zum Austausch von Algorithmen zwischen unterschiedlichen Firmen und Behörden gute Dienste leisten.

XAlgo zeigt, dass der Entwurf einer derart portablen Sprache durch die Verwendung von XML möglich und sogar innerhalb einer Diplomarbeit praktikabel durchführbar ist. Wünschenswert wäre es noch, wenn sich weitere Fachleute oder Experten finden, die dieses Thema oder eventuell sogar diese spezielle Arbeit weiterentwickeln. Dadurch stünde für einen breiten öffentlichen Einsatz von XAlgo mehr Unterstützung zur Verfügung. Zur unabhängigen Ergänzung der vorliegenden Arbeit habe ich aus diesem Grund die Internetdomäne „www.xalgo.org“ registrieren lassen. Dort ist eine offene Kommunikationsplattform zur Weiterentwicklung von XAlgo im Entstehen.

Anhang

Anhang A

Sprachreferenz von XAlgo

A.1 Grundelemente

Element	Beschreibung
<code><xalgo></code>	Wurzelement von XAlgo-Dokumenten im Namensraum „ http://www.f-seidel.de/xmlns/2004/XAlgo/1.0 “
<code><declarations></code>	(Optional) Erstes Unterelement von <code><xalgo></code> und <code><subalgo></code> ; Behälter für alle deklarativen Elemente
<code><definitions></code>	(Optional) Zweites Unterelement von <code><xalgo></code> und <code><subalgo></code> ; Behälter für alle Subalgorithmen
<code><start></code>	(Zwingend) Letztes Unterelement von <code><xalgo></code> und <code><subalgo></code> ; enthält prozedurales Element
<code><subalgo></code>	(Beliebig oft) Unterelement von <code><definitions></code> ; Container für Subalgorithmen; eigene Unterstruktur wie die von Wurzelement <code><xalgo></code>

Tabelle A.1: Grundelemente von XAlgo

A.2 Deklarative Elemente

Element	Beschreibung
<name>	Zwingend direkt als erstes Unterelement von <declarations> als Name für Algorithmus oder Subalgorithmus; außerdem universelles Element zur Bezeichnung von Variablen, Konstanten, Klassen, Objekten, Klassenelementen, Arraydimensionen und Parametern
<type>	Zwingendes Element zur Datentypspezifikation innerhalb von Variablen, Konstanten, Klassenelementen, Parametern, Rückgabewert und Objekten
<parameters>	(Optional) Zweites Unterelement von <declarations>; Spezifikation der erwarteten Parameter
<param>	Beliebig oft innerhalb von <parameters>; repräsentiert einen Parameterwert und enthält Namen, Type und optional <defaultvalue>
<defaultvalue>	Optional in <param> zur direkten Spezifikation eines Standardwertes, falls Parameter bei Aufruf weggelassen wird
<classes>	(Beliebig oft) Container für Klassendeklarationen in <class>-Umgebungen
<class>	(Beliebig oft) Innerhalb des <classes>-Containers zur Spezifikation einer Klasse; enthält (zwingend) Name, <baseclass>, <extensions> und <redefines>
<baseclass>	(Optional) Innerhalb einer Klasse; gibt Elternklasse an, von der geerbt werden soll
<extensions>	(Optional) Container für beliebig viele Variablen, Konstanten und Subalgorithmen
<redefines>	(Optional) Darf Subalgorithmen enthalten, die geerbte mit exakt gleichen Schnittstellen überschreibt.

Tabelle A.2: Deklarative Elemente von XAlgo Teil 1

Element	Beschreibung
<constants>	Enthält zuerst (zwingend) einen Datentyp; danach beliebig viele <const>-Elemente
<const>	In <constants>-Umgebung; repräsentiert eine Konstante; enthält immer Namen und <value> mit direkter Belegung
<value>	Enthält Vorbelegung von Konstanten und Variablen
<variables>	Enthält zuerst (zwingend) einen Datentyp; danach beliebig viele <var>-Elemente
<var>	In <variables>-Umgebung; repräsentiert eine Variable; enthält immer einen Namen und optional <value>
<objects>	Spezifikation von Klasseninstanzen; enthält zuerst den Typ, der allerdings Klassennamen enthalten muss; danach folgen beliebig viele <object>-Umgebungen.
<object>	Repräsentiert ein Objekt; enthält immer Namen und <instancetype> (mit Klassenname) zur Festlegung der erzeugten Instanz.
<returns>	Legt Informationen zu optionalem Rückgabewert fest; enthält entweder nur <null /> oder Typ und optional <default>
<null />	Innerhalb von <returns>; legt explizit fest, dass keine Rückgabe eines Wertes vorgesehen ist
<default>	Innerhalb von <returns>; enthält Ausdruck, der zurückgegeben wird, wenn im Algorithmus nichts angegeben wird

Tabelle A.3: Deklarative Elemente von XAlgo Teil 2

A.3 Prozedurale Elemente

Element	Beschreibung
<null />	Anweisung ohne Funktion; z. B. für leere prozedurale Teile
<sequence>	Container-Element zur Erfassung beliebig vieler Anweisungen
<loop>	Schleifenkonstrukt; enthält <conditions> und <body>
<conditions>	Schleifenkopf; enthält (optional) <loop-init-statement>, <breakcondition> und (optional) <loop-update-statement>
<loop-init-statement>	Enthält Anweisung zur Initialisierung der Schleife
<breakcondition>	Schleifenabbruchsbedingung; enthält MathML-Ausdruck
<loop-update-statement>	Enthält Anweisung, die nach jedem Ende des Schleifenrumpfes ausgeführt wird
<body>	Schleifenrumpf; Enthält beliebig viele Anweisungen, gegebenenfalls <breakloop />, <continueloop /> und <checkloopcondition />
<breakloop />	Vorzeitiges Beenden der umgebenden Schleife
<continueloop />	Überspringen des restlichen Schleifenrumpfes
<checkloopcondition />	Überprüfung der Schleifenabbruchbedingung und gegebenenfalls Abbruch der Schleife
<choice>	Umgebung zur Realisierung einer bedingten Ausführung; enthält eine <select>-Umgebung, beliebig viele <case>-Blöcke und optional eine <othercase>-Struktur

Tabelle A.4: Prozedurale Elemente von XAlgo Teil 1

Element	Beschreibung
<select>	Innerhalb von <choice>; enthält auszuwertenden Ausdruck für Vergleiche
<case>	Blöcke innerhalb von <choice>; enthält <condition> und <then>
<condition>	Innerhalb von <case>; enthält auszuwertenden Ausdruck für Vergleich mit dem Ergebnis aus <select>. Bei positivem Ergebnis wird das folgende <then>-Element ausgeführt
<then>	Innerhalb von <case>; Anweisung, die ausgeführt wird, wenn der <case>-Block zutrifft.
<othercase>	Innerhalb von <choice>; wie <case>-Block, aber ohne <condition>; Inneres <then> wird ausgeführt, wenn keiner der vorigen <case>-Strukturen getroffen hat.
<assign>	Anweisung zur Zuweisung eines Ausdrucks an Variable, Klasselement oder Arrayelement; enthält immer <source> und <target>
<source>	Enthält den auszuwertenden Ausdruck für eine Zuweisung
<target>	Enthält Bezeichner einer Variablen, Arrayelement oder Klasselement
<callsub>	Anweisung zum Aufrufen eines Subalgorithmus; enthält Namen, Parameter (mit <value> in <param>-Umgebungen) und optional <returnvaluetarget>.
<returnvaluetarget>	Innerhalb <callsub>; enthält Bezeichner für Element, das den Rückgabewert eines Subalgorithmus aufnehmen soll
<return>	Anweisung zur Rückkehr aus dem gerade ausgeführten Haupt- oder Subalgorithmus; enthält Ausdruck, dessen Ergebnis von Aufrufen entgegengenommen werden kann (mit <returnvaluetarget>)

Tabelle A.5: Prozedurale Elemente von XAlgo Teil 2

Element	Beschreibung
<output>	Ausgabeeanweisung; erwartet Ausdruck, dessen Ergebnis auf Standardausgabe ausgegeben wird. Optionales Attribut <code>file</code> wird noch nicht unterstützt.
<input>	Eingabeeanweisung; erwartet Bezeichner, der Ergebnis der Eingabe aufnehmen kann. Optionales Attribut <code>file</code> wird noch nicht unterstützt.
<errorout>	(Fehler-) Ausgabeeanweisung; erwartet Ausdruck, dessen Ergebnis auf Fehlerausgabe ausgegeben wird. Optionales Attribut <code>file</code> wird noch nicht unterstützt.
<halt />	Abbruchanweisung; bricht Algorithmenverarbeitung sofort komplett ab

Tabelle A.6: Prozedurale Elemente von XAlgo Teil 3

A.4 Elemente für Ausdrücke

Element	Beschreibung
<unixtime />	Steht für seit dem 01.01.1970 (0:00 Uhr UTC) vergangenen Sekunden
<math>	Mathematischer Ausdruck (jeglicher Art) in MathML; in Namensraum „http://www.w3.org/1998/Math/MathML“
<elementpath>	Pfad zu Elementen in Klasseninstanzen; erwartet zuerst Namen des Objekts, dann kommt ein <sub>-Element
<sub>	In <elementpath>; enthält den Namen (in <name>) des Klasselements
<text>	Enthält direkt eine beliebige Zeichenkette; z. B. für Strings Textausgaben
<array>	Ausdruck zur Referenzierung eines bestimmten Arrayelements; enthält zuerst den Namen und je vorhandener Dimension eine <dimension>-Umgebung
<dimension>	In <array>; enthält zuerst den Namen der Dimension und dann <index>
<index>	In <dimension>; enthält direkt den Indexwert der jeweiligen Dimension als Zahlenwert oder MathML-Ausdruck; Die Indexzählung beginnt immer bei 0.
<arrayliteral>	Ausdruck für vollständig (samt Inhalt) beschriebenes Array; experimentell; noch nicht unterstützt
<arraylength>	Ausdruck zur Größenbestimmung einer Arraydimension; enthält zuerst den Namen der Arrayvariablen, danach ein <dimension>-Element, das nur den Namen der Dimension, aber keinen Indexwert enthält
<directnative>	Ausdrücke in Textform, die direkt und ohne Prüfung in die Zielsprache übernommen werden; wird nur zur Fehlersuche empfohlen

Tabelle A.7: Elemente für Ausdrücke in XAlgo

A.5 Dokumentationselemente

Element	Beschreibung
<author>	(Beliebig oft) Enthält Informationen über den Autor; enthält (mindestens) <personname>, (optional) <birthday> und (optional) <homecountry>.
<personname>	In <author>; vollständiger Name des Autors
<birthday>	In <author>; Geburtstag des Autors
<homecountry>	In <author>; Herkunftsland des Autors in RFC 1766 (z. B. „de“, „en“ etc.)
<creationdate>	Legt Erstellungsdatum fest; erwartet Datum oder Datum mit Uhrzeit nach ISO 8601
<licence>	Beschreibt die genutzte Lizenz; enthält (jeweils optional) <licencetype>, <name> mit Lizenzname und <fulltext>
<licencetype>	In <licence>; Typ der Lizenz (z. B. Open-Source etc.)
<fulltext>	In <licence>; gesamter Text der Lizenzbestimmung
<version>	Versionsinformationen
<build>	Zusatzinformationen für Übersetzung
<purpose>	Zweck bzw. Ziel des beschriebenen Bereichs
<annotation>	Sonstige Anmerkung (auch in freier Textform)

Tabelle A.8: Dokumentationselemente in XAlgo-<metadocu>-Umgebung

Element	Beschreibung
<todo>	Text zur Beschreibung der noch zu erledigenden Aufgaben
<date>	Datum des Eintrages (nach ISO 8601)
<from>	Verfasser des Eintrages
<to>	Zielpersonen des Eintrages
<message>	Zusatznachricht für Zielpersonen
<errordescription>	Beschreibung noch vorhandener Fehler
<fixed>	Beschreibung bereits reparierter Fehler
<annotation>	Sonstige Anmerkungen (auch in freier Textform)

Tabelle A.9: Dokumentationselemente in XAlgo-<develdoc>-Umgebung

Element	Beschreibung
<summary>	Kurze Zusammenfassung für beschriebenen Bereich
<details>	Ausführliche Beschreibung
<example>	(Beliebig oft) Codebeispiele zur Verdeutlichung
<seealso>	(Beliebig oft) Verweise auf andere Sprach- oder Algorithmenkonstrukte
<link>	(Beliebig oft) Spezifikation eines Links (mit einer URI), der weiterführende Informationen enthält

Tabelle A.10: Dokumentationselemente in XAlgo-<explanation>-Umgebung

Element	Beschreibung
<comment>	Darf überall, wo kein Text erwartet wird, vorkommen und enthält in freiem Text einen Kommentar oder Zusatzinformationen, die in keine der vorigen Elemente passt.
<!--... -->	Gewöhnlicher XML-Kommentar; Sollte nur sehr begrenzt eingesetzt werden.

Tabelle A.11: Sonstige Dokumentationselemente

Anhang B

Datenträger

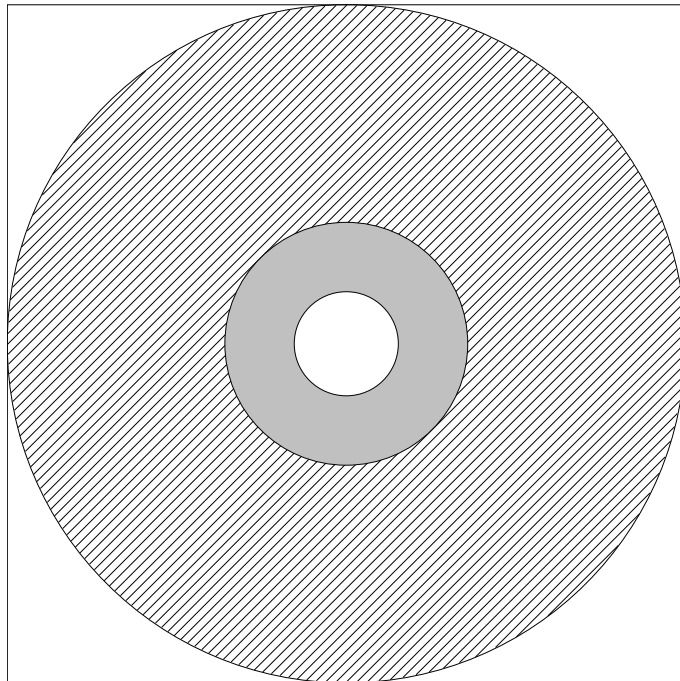
B.1 Beschreibung

Auf der CD, die auf der nächsten Seite beigelegt ist, befindet sich:

- die vorliegende Arbeit in verschiedenen Dateiformaten
- das Diplomarbeitposter
- die erstellten RELAX NG-Schemas
- die programmierten XSLT-Transformationskomponenten
- mehrere XAlgo-Beispielprogramme
- die RELAX NG-Validatoren Jing und MSV
- der XSLT-Prozessor Saxon
- der Quellcode-Beautifiler AStyle
- mehrere lokale Kopien wichtiger Internetquellen

Der Datenträger wurde mit äußerster Sorgfalt erstellt. Sollte es dennoch zu Leseschwierigkeiten kommen, kann der vollständige Inhalt über die unveröffentlichte Internetadresse „<http://www.it-seidel.de/xalgo/>“ heruntergeladen werden.

B.2 Medium



Abbildungsverzeichnis

3.1	Verarbeitungsschema für XAlgo-Dokumente	8
4.1	Phasen eines Compilers	9
4.2	abstrakter Syntaxbaum einer Zuweisung	10
5.1	XAlgo-Dokument in Firefox	15
5.2	XAlgo-Dokument in Microsoft Internet Explorer	16
7.1	MathML in Mozilla Firefox	20
13.1	Erweitertes Verarbeitungsschema für XAlgo-Dokumente	73

Tabellenverzeichnis

9.1 Einfache Datentypen von XAlgo	25
10.1 Schleifenablaufbehle	41
10.2 Unterstützte mathematische MathML-Operatoren	48
10.3 Unterstützte Vergleichs- und Verknüpfungsoperatoren aus MathML	50
13.1 Optionale Kommandozeilenparameter für Jing	75
13.2 Optionale Kommandozeilenparameter für MSV	76
13.3 Optionale Kommandozeilenparameter für Saxon	77
A.1 Grundelemente von XAlgo	85
A.2 Deklarative Elemente von XAlgo Teil 1	86
A.3 Deklarative Elemente von XAlgo Teil 2	87
A.4 Prozedurale Elemente von XAlgo Teil 1	88
A.5 Prozedurale Elemente von XAlgo Teil 2	89
A.6 Prozedurale Elemente von XAlgo Teil 3	90
A.7 Elemente für Ausdrücke in XAlgo	91
A.8 Dokumentationselemente in XAlgo-<metadocu>-Umgebung	92
A.9 Dokumentationselemente in XAlgo-<develdoc>-Umgebung	93
A.10 Dokumentationselemente in XAlgo-<explanation>-Umgebung	93
A.11 Sonstige Dokumentationselemente	93

Listings

4.1 XAlgo-Segment einer Zuweisung	10
10.1 Minimale XAlgo-Grundstruktur	31
10.2 Maximale XAlgo-Grundstruktur	31
10.3 XAlgo-Grundstruktur mit einem Subalgorithmus	32
10.4 Beispiel einer typischen Parameterliste	33
10.5 Beispiel einer typischen Klassendeklaration	35
10.6 Beispiel einer typischen Konstantendeklaration	36
10.7 Beispiel einer typischen Variablendeklaration	36
10.8 Beispiel einer typischen Objektdeklaration	37
10.9 Spezifikation eines Rückgabewertes	37
10.10 Beispiel einer typischen Arraydeklaration	38
10.11 Minimales XAlgo-Dokument	39
10.12 Beispiel einer Sequence-Struktur	39
10.13 Beispiel einer Schleife	41
10.14 Beispiel einer If-Entsprechung	42
10.15 Beispiel einer bedingten Ausführung	43
10.16 Beispiel einer Zuweisung	44
10.17 Beispiel eines Subalgorithmen-Aufrufes	45
10.18 Beispiel einer Rückkehranweisung	45
10.19 Beispiel einer Eingabeweisung	46
10.20 Beispiel einer Ausgabeweisung	46
10.21 Beispiel einer Fehlerausgabe	47
10.22 Beispiel einer Zeitermittlung	47
10.23 Beispiel einer <halt />-Anweisung	47
10.24 Beispiel eines mathematischen Ausdrucks	49
10.25 Beispiel eines booleschen Ausdrucks	49
10.26 Beispiel eines Elementpfades	50
10.27 Beispiel eines Text-Ausdrucks	51
10.28 Beispiel für Arrayelementen	51
10.29 Beispiel zur Arraygrößenbestimmung	53
10.30 Beispiel für nativen Ausdruck	53
10.31 Beispiel für Autorendokumentation	54
10.32 Anwendung der Dokumentations-elemente	57

12.1 Zähler-Beispielalgorithmus	62
12.2 Javaquellcode für Zähler-Beispiel	65
12.3 Quicksort-Beispielalgorithmus	65
12.4 Klassen-Beispielalgorithmus	69
12.5 Javaquellcode für Klassendemo-Beispiel	71
12.6 C++-Quellcode für Klassendemo-Beispiel	72
13.1 Javaquellcode ohne Beautifier	78

Abkürzungsverzeichnis

DOM	Document Object Model; definiert API für Baumzugriffe auf XML-Dokumente [Schiedermeier 2004 , Seite 17]
DTD	Document Type Definition; ältere SGML-konforme Definition von Dokumenttypen [Schiedermeier 2004 , Seite 69]
MathML	Mathematical Markup Language; XML-Sprache zur Beschreibung von mathematischen Strukturen und Inhalten; http://www.w3.org/TR/MathML2/
SGML	Standard Generalized Markup Language; Vorbild für XML und seit 1986 ISO-Norm [W3C XML 2003 , Punkt 6]
URI	Uniform Resource Identifiers; kompakte Zeichenkette zur Identifizierung von abstrakten und physikalischen Ressourcen (frei nach RFC 2396; http://www.ietf.org/rfc/rfc2396.txt)
W3C	World Wide Web Consortium (http://www.w3.org bzw. auf deutsch http://www.w3c.de) entwickelt Internet-Technologieempfehlungen (sog. „ <i>W3C Recommendations</i> “)
XML	Extensible Markup Language (frei übersetzt: erweiterbare Auszeichnungssprache); auch manchmal scherzhaft für <i>Excellent Marketing Language</i> [Vlist 2004 , Seite 3]
XSLT	Extensible Style Language for Transformations; XML-Sprache zur Transformation von XML-Dokumenten

Literaturverzeichnis

- [Aho et al. 1999] A. V. Aho, R. Sethi, J. D. Ullmann: *Compilerbau Teil 1* (1999); ISBN: 3-486-25294-1
- [Aho et al. 1999b] A. V. Aho, R. Sethi, J. D. Ullmann: *Compilerbau Teil 2* (1999); ISBN: 3-486-25266-6
- [Bauer et al. 1984] F. L. Bauer, H. Wössner: *Algorithmische Sprache und Programmentwicklung* (1984); ISBN: 3-540-12962-6
- [Dittmer 1996] Ingo Dittmer: *Konstruktion guter Algorithmen* (1996); ISBN: 3-519-02990-1
- [Ghezzi et al. 1989] C. Ghezzi, M. Jazayeri: *Konzepte der Programmiersprachen* (1989); Begriffliche Grundlagen, Analyse und Bewertung; ISBN: 3-486-20619-2
- [Güting 1992] Ralf Hartmut Güting: *Datenstrukturen und Algorithmen* (1992); ISBN: 3-519-02121-8
- [Knuth 1983] Donald E. Knuth: *Literate Programming* (Sept. 1983); aus „THE COMPUTER JOURNAL“; <http://www.literateprogramming.com/knuthweb.pdf>
- [Mangano 2003] Sal Mangano: *XSLT Cookbook* (2003); ISBN: 0-596-00372-2
- [Pingel 2003] Steffen Pingel: *Abstrakte Syntaxbäume* (2003); http://www.iste.uni-stuttgart.de/ps/Lehre/S_Graphen/pingel.pdf

- [Pratt et al. 1997] Terrence Pratt, Marvin Zelkowitz: *Programmiersprachen* (1997); Design und Implementierung; ISBN: 3-8272-9547-5
- [Ray 2001] Erik T. Ray: *Einführung in XML* (2001); ISBN: 3-89721-286-2
- [Sander et al. 1995] P. Sander, W. Stucky, R. Herschel: *Automaten Sprachen Berechenbarkeit* (1995); ISBN: 3-519-12937-X
- [Schiedermeier 1996] Prof. Dr. J. Schiedermeier: *Vorlesung „Programmieren I“* (1996); Kapitel 5.4: Strukturierte Programmierung; <http://www.cs.fhm.edu/~schieder/programmieren-1-ws96-97/struktpgm.html>
- [Schiedermeier 2004] Prof. Dr. Christian Schiedermeier: *XML Skriptum* (WS 2003/2004); http://www.informatik.fh-nuernberg.de/professors/schiedermeier/WS_2003_2004/XML/Vorlesung/xml-12.pdf
- [Schöning 2001] Uwe Schöning: *Algorithmik* (2001); ISBN: 3-8274-1092-4
- [uni-protokolle.de] uni-protokolle.de (o.V.): *Pseudocode* (o.J./Stand 21.08.2004); <http://www.uni-protokolle.de/Lexikon/Pseudocode.html>
- [Vlist 2004] Eric van der Vlist: *RELAX NG* (2004); A Simpler Schema Language for XML; ISBN: 0-596-00421-4
- [W3C Rec. 2004] W3C: *Extensible Markup Language (XML) 1.0 (Third Edition)* (2004); <http://www.w3.org/TR/2004/REC-xml-20040204/>
- [W3C XML 2003] W3C: *XML in 10 points* (2003); <http://www.w3c.de/Misc/XML-in-10-points.html>
- [Withopf 1999] Matthias Withopf für c't: *Duales System Delphi* (1999); Abschnitt „Fehlertolerant“, 6. Absatz; <http://www.heise.de/ct/95/10/028/>
- [ZapThink 2002] Zap Think LLC: *KEY XML SPECIFICATIONS AND STANDARDS* (2002); <http://www.oasis-open.org/committees/download.php/173/xml%20standards.pdf>

Index

A

abstrakte Syntaxbäume, 9, 10
Aktualparameter, 44
<and />, 50
<annotation>, 55, 56, 92, 93
<apply>, 48, 49, 69
<array>, 51, 91
<arraylength>, 52, 91
<arrayliteral>, 52, 81, 91
Arrays, 25, 37, 51, 52
 Dimensionen, 25, 37, 52
 Literale, 52
<arraytype>, 37
<assign>, 44, 64, 68, 89
AST, *siehe* abstrakte Syntaxbäume
AStyle, 78, 80, 94
Attribut
 file, 46, 61, 82, 90
 href, 60
 name, 38, 64
<author>, 53, 92

B

<baseclass>, 34, 86
benannte Referenzen, 29
BIOML, 7
<birthday>, 54, 92
<body>, 40, 88
<breakcondition>, 40, 64, 88
<breakloop />, 41, 88
<build>, 54, 92

C

C++, 1, 7, 13, 61, 77, 78, 81
call-by-value, 26
<callsub>, 44, 69, 89

<case>, 41, 88, 89
<checkloopcondition />, 41, 88
<choice>, 40, 42, 43, 88, 89
<ci>, 48
<class>, 34, 86
<classes>, 31, 34, 36, 86
<cn>, 49
Coderendering, 19
<comment>, 28, 57, 93
Compilerphasen, 9–10
<condition>, 41, 42, 89
<conditions>, 40, 64, 88
<const>, 34, 87
<constants>, 31, 34, 36, 64, 87
<continueloop />, 41, 88
<copyright>, 54
<creationdate>, 54, 55, 92
CSC, 17

D

<date>, 55, 93
Datentypen, 25, 36
<declarations>, 30, 31, 36, 85, 86
<default>, 35, 87
<defaultvalue>, 33, 86
<definitions>, 30, 68, 85
<details>, 56, 93
<develdoc>, 28, 55, 93
<dimension>, 37, 51, 52, 91
<directnative>, 52, 91
<div />, 48
do-while-do, 41
<document>, 60
Dokumentation, 17, 19
DOM, 100
DOM-Bäume, 10, 13

Doxygen, 17
DTD, 21, 100

E

Elemente

- prozedurale, 27
- zur Dokumentation, 27
- <elementpath>, 49, 50, 71, 91
- <elementtype>, 37, 68
- <eq />, 50
- <errordescription>, 55
- <errordescriptions>, 93
- <errorout>, 46, 90
- <example>, 56, 93
- <explanation>, 28, 56, 93
- <extensions>, 34, 86

F

Felder, *siehe* Arrays

- <fixed>, 56, 93
- <from>, 55, 93
- <fulltext>, 54, 92

G

Garbagecollector, 60

- <geq />, 50

goto-Anweisungen, 41

- <gt />, 50

H

- <halt />, 47, 90
- <homecountry>, 54, 92

I

IML, 11

- <index>, 51, 91
- <input>, 46, 90
- <instancetype>, 35, 87

Internet Explorer, 15

ISO

- 8601, 54, 55

J

Java, 1, 2, 7, 13, 60, 60, 74, 75, 77–79, 81

Javadoc, 17
jEdit, 64
Jing, 74, 94

K

Klassen, 26

- Elemente, 26
- Instanzen, 26
- Vererbung, 26

Knuth, 17

L

Lebensdauer, 26

- <length>, 37
- <leq />, 50
- <licence>, 54, 92
- <licencetype>, 54, 92
- <link>, 56, 93

Linux, 76, 78, 79

Literate Programming, 17

- <loop>, 40, 88
- <loop-init-statement>, 40, 64, 88
- <loop-update-statement>, 40, 64, 88
- <lt />, 50

M

- <math>, 48, 49, 91

MathML, 6, 48, 59, 64, 77, 79, 100

- <message>, 55, 93
- <metadocu>, 28, 53, 92

Microsoft, 15

- <minus />, 48
- <mod />, 48

Mozilla, 15

MSV, 74, 75, 94

N

- <name>, 31–35, 37, 44, 49–52, 54, 70, 86, 91, 92

Nebenwirkungen, 27

- <neq />, 50
- <null />, 35, 39, 87, 88

O

- <object>, 35, 87

<objects>, 31, 35, 37, 71, 87
<or />, 50
<othercase>, 41, 42, 88, 89
<output>, 43, 46, 90

P

<param>, 33, 44, 86, 89
<parameters>, 31, 32, 36, 44, 82, 86
Pascal, 2, 5
Perl, 7
<personname>, 53, 92
<plus />, 48
Pseudocode, 2, 4, 7, 12
<purpose>, 54, 92

R

<redefines>, 26, 34, 86
RELAX NG, 11, 12, 14, 21, 25, 28, 30, 52, 73, 94
Rendering, 19–20
<return>, 45, 47, 89
<returns>, 31, 35, 36, 87
<returnvaluetarget>, 44, 89
RFC
 1766, 54, 92
 2396, 100

S

Saxon, 58, 60, 76, 77, 79, 80, 94
<seealso>, 56, 93
<select>, 40, 41, 88, 89
<sequence>, 39, 40, 43, 64, 88
SGML, 100
Sichtbarkeit, 26
<source>, 44, 89
<start>, 30, 38, 39, 64, 71, 85
STL, 61
Stylesheets, 58
<sub>, 50, 91
<subalgo>, 30, 34, 85
Subalgorithmen, 26
<summary>, 56, 93

T

<target>, 44, 89

<text>, 51, 91
<then>, 41, 89
<times />, 48
<to>, 55, 93
<todo>, 55, 93
touringmächtig, 58
Typüberprüfung, 24, 61, 81
<type>, 33–37, 86

U

<unixtime />, 47, 91
URI, 46, 56, 93, 100

V

<value>, 34, 44, 87, 89
<var>, 34, 87
<variables>, 31, 34, 36, 64, 87
Vektoren, *siehe* Arrays
<version>, 54, 92

W

W3C, 100
Wirth, N., 5

X

XAlgo, 7–8
 Entwurf, 24
 Schema, 28
 open, 29, 73
 strict, 29, 73
 Versionen, 29
<xalgo>, 63, 85
xalgo2java.xsl, 77
XML, 4, 100
XML Schema, 21
XML Signaturen, 22
XPath, 59
 generate-id(), 59
 last(), 60
XPath , 60
XSLT, 5, 58, 100