XML4Ada95

Zdenko Vrandečić

7. Oktober 2003

Inhaltsverzeichnis

1. Übersicht

"XML: the next big thing"
(Tom R. Halfhill, 1999)

Ziel dieser Diplomarbeit ist es, ein validierendes XML-Paket für Ada 95 zur Verfügung zu stellen. Es ermöglicht mittels der DOM Zugriff auf XML–Dokumente.

In Kapitel 2 werden zunächst die Grundlagen geklärt: was ist XML, welche Ideen stecken dahinter? Auch die Motivation für diese Arbeit wird hier dargelegt.

Kapitel 3 beschreibt die Kriterien für die Auswahl einer zugrunde liegenden Implementierung, beleuchtet dann die möglichen Kandidaten und begründet die getroffene Auswahl. Zwar konzentriert sich dieses Kapitel auf die für die Aufgabe relevanten Kriterien, dennoch kann es jedem eine Hilfe sein, der eine entsprechendes Auswahl zu treffen hat.

In Kapitel 4 wird die Vorgehensweise beschrieben, mit der die in C++ vorliegende Implementierung an Ada 95 gebunden wurde. Die hier beschriebene Techniken können als allgemein gültige Muster zur Einbindung von C++-Modulen in Ada 95-Projekte verwendet werden, auch wenn sie nicht alle möglichen Probleme lösen. Dies ist die zur Zeit umfassendste Beschreibung von Mustern zu diesem Zweck.

Der Entwurf der DOM-Schnittstelle für Ada 95 erforderte einige Entscheidungen. Diese werden in Kapitel 5 begründet und verteidigt.

Motivation dieser Arbeit war es, einen robusteren Parser für die GXL-Dokumente innerhalb des Bauhausprojektes zur Verfügung zu stellen. Wie dieses Ziel mit dem neu erstellten Paket bewältigt wurde, erläutert Kapitel 6.

Kapitel 7 beleuchtet die Möglichkeiten und voraussichtlichen Schwierigkeiten bei einer etwaigen Realisierung der SAX-Schnittstelle für Ada 95.

Schließlich wird in Kapitel 8 die Arbeit noch einmal abschließend betrachtet und die wichtigsten Ergebnisse kurz zusammengefasst. Ein kurzer Ausblick soll eine mögliche weitere Entwicklung auf diesem Gebiet skizzieren.

Im Anhang findet sich ein Glossar der Abkürzungen und technischen Begriffe sowie ein Verzeichnis der verwendeten Literatur, insbesondere der Webadressen. Auf der diese Diplomarbeit begleitenden Website - `http://www.nodix.de/xml4ada95` - finden sich die Links ebenfalls (mit den in dieser Arbeit verwendeten Abkürzungen), so dass ein fehleranfälliges Eintippen von Hand nicht notwendig ist. Dort findet sich auch das XML4Ada95-Paket mitsamt Dokumentation.

Zdenko „Denny" Vrandečić ist Student der Informatik an der Universität Stuttgart. Dieser Text entstand als Diplomarbeit im Sommer 2003 und wurde von Daniel Simon und Stefan Bellon betreut und von Prof. Dr. Erhard Plödereder geprüft.

2. XML, DOM und andere Abkürzungen

„Die Geschichte ist ein Drehbuch von miserabler Qualität."
(Norman Mailer)

In diesem Kapitel wird der Hintergrund für die gesamte Diplomarbeit aufgebaut.

Hier werden die technischen Grundlagen der Arbeit beschrieben: was ist XML eigentlich (Abschnitte 2.1 und 2.2)? Welche relevanten Standards in diesem Bereich sollten bekannt sein (Abschnitte 2.3 und 2.4)? All jenen, die sich bisher nur am Rande damit beschäftigt haben, wird hier ein schneller und dennoch gründlicher Einstieg in XML angeboten. Schließlich wird hier auch die Motivation dargelegt, warum es jetzt zu dieser Arbeit kam, und die ersten, grundlegenden Entscheidungen werden erläutert (Abschnitt 2.5).

2.1. Die Entwicklung von XML

Ab Mitte der 1990er schossen private Homepages wie Pilze aus dem Boden. Heute, nur zehn Jahre nachdem der Startschuss für das World Wide Web 1993 gefallen war, gibt es allein in Deutschland weit über sechs Millionen registrierte Domains ([Den03]). Doch obwohl nahezu alle Homepages in der Sprache **HTML** geschrieben sind, ist den meisten der Hintergrund dieser Sprache nicht bekannt.

Ohne sich das historische und technische Basiswissen zu HTML zu erwerben, erlernten hunderttausende die neue Sprache, und dank immer leistungsfähigerer und intuitiv bedienbarer Editoren wuchs die Zahl der HTML-Autoren noch schneller. Schließlich verwenden heute aberhunderte Millionen Menschen täglich mit HTML aufgebaute Dienste, ohne sich dessen Grundlagen bewusst zu sein. Genau das spricht für eine stabile Technik, genau das erklärt ihre rasante Verbreitung - dass die Nutzer sich keine Gedanken über diese machen müssen.

Kam HTML aus dem nichts? Wie erklärt sich die schnelle Verbreitung? Wurde es von einem genialen Schweizer Forscher am CERN einfach so entworfen, der dann auch gleich die notwendigen Werkzeuge zur Verfügung stellte?

Natürlich nicht. HTML beruhte auf **SGML**, mit dem man bereits damals 30 Jahre Erfahrung gesammelt hatte. Deswegen standen auch von Beginn an ausreichend Werkzeuge und genug Experten mit Erfahrung zur Verfügung, eine ideale Situation für die Förderung einer neuen Technik.

Doch was ist SGML? SGML basiert auf GML, welches in den 1960ern von Charles Goldfarb, Edward Mosher und Raymond Lorie bei IBM entwickelt wurde. GML steht für ***G**eneralized **M**arkup **L**anguage* (*Verallgemeinerte Auszeichnungssprache*; und nicht etwa

für die Anfangsbuchstaben der Nachnamen der Entwickler), das später hinzugefügte S steht schlicht für *Standard*. SGML ist - trotz ihres Namens - eine **Metasprache**. Man kann also keine Texte in SGML schreiben, sondern vielmehr Sprachen definieren, in denen dann Dokumente verfasst werden können. SGML wurde 1986 als ISO Standard 8879 verabschiedet ([Gol86]).

Derart definierte Sprachen werden *applications*, Anwendungen, von SGML genannt, Texte, die dann mit einer solchen Sprache verfasst werden, *documents*, Dokumente. Um eine Anwendung zu entwerfen, muss der Benutzer eine **Document Type Definition** (*Definition der Dokumentenart*), oder kurz **DTD**, verfassen. Hierbei wird die Syntax von möglichen Dokumenten definiert. Ein allgemeiner SGML Parser kann dann mit der entsprechenden DTD ein beliebiges Dokument einlesen und auf seine syntaktische Korrektheit überprüfen.

HTML ist tatsächlich nichts anderes als eine Anwendung von SGML. Die entsprechende DTD findet sich auf den Seiten des **World Wide Web Consortium**, kurz **W3C** genannt (wenn auch, um der tatsächlichen Entwicklung Tribut zu zollen, hier kurz angemerkt werden soll, dass die meisten *Web-Browser*, Programme, die zur Darstellung von HTML-Dokumenten verwendet werden, nie die Dokumente gegen die HTML-DTD prüften, sondern vielmehr versuchten syntaktische Fehler zu ignorieren, um die Dokumente dennoch darzustellen).

Zwar wurde HTML zur bei weitem meistgenutzten Anwendung von SGML, doch schon zuvor wurde SGML vor allem in großen Projekten zur Dokumentation verwendet. SGML ist extrem mächtig und flexibel, doch auch hier gilt, wie so häufig in der Informatik, aus großer Macht folgt große Komplexität. Und so wurde eben dies zum Stolperstein für SGML auf seinem Weg zur weiteren Verbreitung: Werkzeuge waren teuer und umfangreich, für die meisten möglichen Anwendungsbereiche wurde SGML als übertrieben und zu umständlich betrachtet. Der überraschende Erfolg von HTML förderte schließlich auch das Interesse an SGML, doch bald war klar, dass SGML in dieser Art nur schwerlich weite Verbreitung finden würde.

Als Lösung für dieses Problem wurde schließlich **XML** vorgeschlagen, die **Extensible Markup Language** (*erweiterbare Auszeichnungssprache*). XML ist ein *profile* von SGML, eine stark vereinfachte Untermenge. Vor allem ohnehin selten verwendete Möglichkeiten von SGML wurden dabei entfernt, um die Flexibilität und Mächtigkeit weitestgehend zu erhalten und dennoch die Komplexität zu senken.

XML wurde im Februar 1998 vom W3C als *Recommendation* ([BPSM98]) verabschiedet. Dies entspricht einem Standard bei anderen Institutionen. Wie auch SGML ist XML eine Metasprache und die Definition von anwendbaren Sprachen erfolgt in der DTD, wobei es trotz der Namensgleichheit zwischen SGML-DTDs und XML-DTDs kleine Unterschiede gibt.

Aufgrund des offenen Standardisierungsprozesses des W3C und der Ähnlichkeit mit SGML standen schon bald erste Werkzeuge für die Arbeit mit XML zur Verfügung. Immer mehr Projekte begannen nun, XML einzusetzen. Mit der wachsenden Verbreitung standen auch bessere Werkzeuge zur Verfügung, was wiederum zu einer noch weiteren Verbreitung führte. Bald begann auch die Entwicklung weiterer Standards in diesem Bereich, um die Möglichkeiten der Anwendung noch zu erweitern. Heute hat XML den Status einer *Lingua franca* wenn es um die syntaktische Definition von Auszeichnungsprachen geht.

2.2. Ein einfaches Beispiel

An folgendem Beispiel soll die grundlegende Struktur von XML verdeutlicht werden. Hierbei wurden nur sehr wenige Möglichkeiten von XML verwendet. Zunächst eine entsprechende DTD (geburtstage.dtd):

```
<!ELEMENT Geburtstagsliste (Person*)>
<!ELEMENT Person (Name, Geburtstag, Sternzeichen, Vorlieben?)>
<!ELEMENT Geburtstag (Tag, Monat, Jahr?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Vorlieben (#PCDATA)>
<!ELEMENT Tag (#PCDATA)>
<!ELEMENT Monat (#PCDATA)>
<!ELEMENT Jahr (#PCDATA)>
<!ELEMENT Sternzeichen (#PCDATA)>
```

In dieser DTD werden einfache Elemente deklariert, die verschiedene Unterelemente haben können (nach dem Namen in Klammern gegeben). * nach dem Namen bedeutet, dass das Element beliebig oft vorkommen kann, ? markiert ein optionales Element. #PCDATA wiederum steht für beliebige Daten - den eigentlichen Inhalt (dies ist eine sehr verkürzte Darstellung. Eine genauere Beschreibung von DTDs findet sich in [BPSMM00]).

Nun das Beispiel einer XML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Geburtstagsliste SYSTEM "geburtstage.dtd">
<Geburtstagsliste>
  <Person>
    <Name>Erika Musterfrau</Name>
    <Geburtstag>
      <Tag>23</Tag>
      <Monat>Mai</Monat>
    </Geburtstag>
    <Sternzeichen>Stier</Sternzeichen>
    <Vorlieben>Schnelle Autos, Nietzsche, Pralinen</Vorlieben>
    <Wohnort>Vanilleeis</Wohnort>
  </Person>
  <Person>
    <name>Erik Mustermann</name>
    <Geburtstag>
      <Tag>36</Tag>
      <Monat>11</Monat>
      <Jahr>Das Jahr der Mondlandung</Jahr>
    </Geburtstag>
  </Person>
</Geburtstagsliste>
```

Eines der stärksten Argumente für den Gebrauch von XML ist die mögliche syntaktische Überprüfung. Statt diese selber schreiben zu müssen, entwirft man nunmehr für seine Daten eine DTD und das XML-Paket übernimmt den Rest.

Die erste Zeile verrät die XML-Version und die verwendete Zeichenkodierung. In der zweiten Zeile erhält hier der Parser die Information zur entsprechenden DTD. Die DTD kann auch in die XML-Datei integriert sein (wobei externe DTDs üblich sind) oder über eine **URI** lokalisiert werden (**Uniform Resource Identifer**, meistens eine Webadresse).

Ein XML-Parser kann das Dokument auf zweierlei Arten prüfen: ob es **wohlgeformt** (*well-formed*) ist, und ob es **validiert**, also ob es gültig gemäß der DTD ist (*validated*). Wohlgeformtheit bedeutet hierbei, dass es (unter anderem) folgende Punkte erfüllt:

- Wohlgeformte Dokumente haben genau eine „Wurzel“, ein Element, dass alle restlichen Elemente umschließt
- Alle geöffneten Elemente wurden auch wieder geschlossen
- Attribute müssen in Anführungszeichen stehen (im Beispiel sind keine vorhanden)
- alle Elemente müssen wieder geschlossen werden, bevor das sie umfassende Element geschlossen wird

Letzteres bedeutet, dass Auszeichnungen wie *<h> X <f> X X </f> </h>* nicht möglich sind, da *<f>* noch offen war, als das *<f>* umfassende ** geschlossen wurde (für eine genaue Definition von Wohlgeformtheit sei auf [BPSMM00] verwiesen).

Das Beispieldokument etwa wird vom Parser als wohlgeformt erkannt. Es wird jedoch nicht validiert werden: Das Element *<Wohnort>* der ersten Person wurde nicht definiert und der zweiten Person fehlt das als notwendig angegebene Element *<Sternzeichen>*. Weiterhin steht bei der zweiten Person *<name>* statt *<Name>*: XML **unterscheidet zwischen Groß- und Kleinbuchstaben** und wird somit auch hier einen Fehler ausgeben (genauer zwei: sowohl wegen des unbekannten Elements *<name>* als auch wegen des fehlenden Elements *<Name>*).

Nicht alle XML-Parser können Dokumente gegen ihre DTDs prüfen - die es können, heißen **validierende Parser** im Gegensatz zu **nicht-validierenden Parsern**. Die Aufgabenstellung verlangt einen validierenden Parser.

An diesem einfachen Beispiel sehen wir auch einige der **Schwächen von XML**: obwohl der Parser das Dokument validiert und als gültig erkennt, sind die Daten teilweise unsinnig - auch auf einer durchaus syntaktisch erkennbaren Ebene. Zudem werden auch vergleichsweise kleine Datenbestände schnell zu umfangreichen Dokumenten (die sich aber hervorragend komprimieren lassen). Auch wird in keiner Weise verhindert, dass sich jeder seinen eigenen Standard erschafft - zudem mit wesentlich weniger aussagekräftigen Elementnamen (etwa *<g>* statt *<Geburtstag>*, oder auch *<Pusteblume>*).

2.3. Verwandte Standards

Diese und viele andere Probleme wurden schon sehr bald erkannt, und es gab zahlreiche Lösungsansätze. Bei vielen Lösungen ist noch nicht ersichtlich, welche sich letztlich durchsetzen werden, andere stecken noch in der Entwicklung. Die wichtigsten, zumal im Zusammenhang mit der Aufgabenstellung, sollen hier kurz genannt werden.

XML Schema wurde im Mai 2001 verabschiedet ([Fal01], [TBMM01], [BM01]) und hat letztlich zum Ziel, DTDs zu ersetzen, die an zahlreichen Schwächen leiden. Im Gegensatz zu den DTDs verfügen Schemata über Datentypen wie Integer, Buchstaben oder Fließkommazahlen, und können dieselben auch weiter beschränken oder zu komplexeren Datenstrukturen zusammenlegen.

Um die Möglichkeiten am oben genannten Beispiel (Seite 9) zu erläutern: XML Schema hätte erlaubt, den <*Tag*> auf ganzzahlige Werte von 1 bis 31 zu begrenzen, beim <*Monat*> entweder nur die ganzzahlige Werte von 1 bis 12 oder nur die Monatsnamen (Januar, Februar...) oder auch beides zuzulassen, schließlich das <*Jahr*> auch mit ganzzahligen Werten zu definieren, und so weiter.

Dass der <*Wohnort*> mit *Vanilleeis* angegeben ist, hätte jedoch auch XML Schema nicht verhindern können (außer es definiert alle möglichen Wohnorte), doch ist durch die Möglichkeiten von Schema den Programmen, die den Parser benutzen, schon sehr geholfen: Überprüfungen, ob bestimmte Grenzwerte eingehalten sind (etwa beim Datum), ob die Werte überhaupt einen bestimmten Typ haben, können so schon an den Parser delegiert werden, und so die Entwicklung weiter vereinfachen.

DTDs weisen einen weiteren gravierenden Nachteil auf, der das Erlernen und demzufolge auch das Benutzen erschwert: sie verwenden eine ganz eigene Syntax, die separat erworben werden muss. XML Schema definiert seine eigene Syntax mit XML, ist also selbst eine Anwendung von XML - dies vereinfacht den Einstieg für den Benutzer (Hinzu kommt, dass es auch das Schreiben von Werkzeugen einfacher gestaltet, kann doch für das Einlesen des Schemas der XML Parser verwendet werden statt eines eigenen DTD Parsers).

Namespaces in XML wurde im Januar 1999 verabschiedet ([BHL99]). Die Aufgabe dieses Standards ist es, Kollisionen von Namen durch das Einführen von Namensräumen zu verhindern. Da ein einzelnes Dokument auf modular zusammengeschlossenen DTDs und Schemata beruhen kann, kann es durchaus passieren, dass kurze und beliebte Elementnamen wie <*name*> oder <*p*> in zwei verschiedenen Definitionen verwendet werden. Um das zu verhindern, werden Namensräume definiert (ähnlich wie etwa in vielen Programmiersprachen), so dass stets klar ist, welches Element konkret verwendet wird (<*Geburtstagsliste:name*>, <*HTML:p*>, <*physics:p*>).

Um die Gefahr einer babylonischen Sprachverwirrung, welche durch die einfache Möglichkeit zur Definition neuer Sprachen auftritt, zu minimieren, gibt es auch mehrere Ansätze. Die klassische Methode ist die weitere Standardisierung: man definiert einfache, konkrete Sprachen für verschiedenste Aufgaben, legt diese Standards offen und erlaubt (oder fördert) deren Gebrauch.

Dies wird in zahlreichen Bereichen mehr oder minder erfolgreich versucht: **MathML** (für Formeln, Gleichungen und mehr, [IM99]), **SVG** (*Scalable Vector Graphics*, für Vektorgrafiken, [Fer01]), **RSS** (*Really Simple Syndication*, für Informationen im Internet, die im

Pushverfahren verbreitet werden, so genannte Newsfeeds, [Win02]), **XHTML** (eine Weiterentwicklung von HTML auf XML-Basis, [Pem00]) und natürlich **GXL** (*Graph Exchange Language*, eine Graphenbeschreibungssprache, [HWS00]) - das sind nur wenige Beispiele für Anwendungen von XML.

Die andere Methode ist es, statt auf gemeinsamen Standards aufzubauen, eine einfache Möglichkeit zur Umwandlung von Dokumenten einer Anwendung in eine andere zu verwenden. **XSL** (*Extensible Stylesheet Language*) und ihr verwandte Techniken erlauben nicht nur die Transformation in eine mögliche Ausgabeform (etwa für den Bildschirm, als klassisches HTML oder PDF) sondern auch die Transformation eines Dokuments in ein anderes. Solange zwei Dokumente die gleiche Informationsmenge in einer lediglich anderen Syntax beinhalten ist verlustfreies Transformieren möglich. Statt also gemeinsame Standards zu verwenden, bietet man günstige, automatische Übersetzer an. Weitere Informationen finden sich hierzu in [ABC+01].

Weitere Standards befassen sich mit XML und Sicherheit (XML Encryption [ER02], XML Signature [ERS02], XML Key Management [HB03]); XML Query, einer Abfragesprache für XML (ähnlich SQL für relative Datenbanken, [FRS+03]) und viele mehr. Zwei der wichtigsten Standards zur Arbeit mit XML werden im folgenden Abschnitt vorgestellt.

2.4. Die Schnittstellen - Arbeiten mit XML

Die ersten XML-Pakete konnten XML-Dateien einlesen und schreiben, die besseren auch validieren. Doch wie der Benutzer des Pakets auf die Daten der XML-Datei zugreifen konnte, war Sache des einzelnen Pakets, und so entstanden zahlreiche Schnittstellen zum Arbeiten mit XML. Schon bald begann man auch hier mit der Standardisierung, wobei zwei Schnittstellen als besonders vielversprechend hervortraten: **SAX** und **DOM**.

SAX - **Simple API for XML** - entstand ohne großer Komiteearbeit auf der *xml-dev*-Mailingliste und wurde schnell von zahlreichen Firmen und Personen implementiert. Der führende Entwickler war David Megginson, heute wird die Schnittstelle von David Brownell gepflegt ([Bro]).

Der besondere Vorteil von SAX ist seine Geschwindigkeit und die Möglichkeit, das Lesen sehr großer XML-Dateien einfacher zu implementieren. Der Benutzer hängt dem Parser entsprechende *callbacks* ein, die, sobald bestimmte Ereignisse auftreten (ein Element wird geöffnet, man begegnet einer *ProcessingInstruction* etc.) aufgerufen werden. Dann liest der Parser das Dokument seriell durch, und jedes Mal wenn ein Ereignis auftritt, für welches ein *callback* registriert ist, wird dieses aufgerufen. Der Benutzer ist selber dafür verantwortlich, sich den Kontext von Elementen zu merken, er kann dem SAX-Parser keine Fragen über den Inhalt des Dokuments stellen, insbesondere nicht vor- oder rückwärts blicken.

Weitere Informationen zur SAX-Schnittstelle finden sich in Kapitel 7.

Das zweite, wichtige Interface bei der Arbeit mit XML ist DOM, das **Document Object Model**. Dabei wird das Dokument als ein Baum betrachtet. Hierfür ein Beispiel mitsamt der daraus aufgebauten DOM.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Person SYSTEM "person.dtd">
<Person>
  <Name>Erika</Name>
  <Geburtstag>23. Mai</Geburtstag>
  <Wohnort>Mainz</Wohnort>
</Person>
```

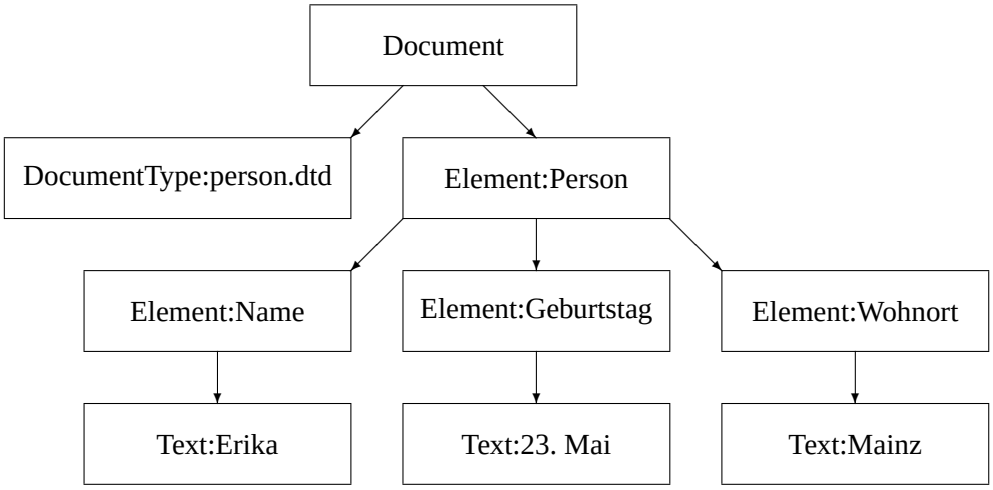

Abbildung 2.1.: DOM von Beispiel

Dem Benutzer bietet die DOM wahlfreien Zugriff auf die Daten des XML-Dokuments. Als Ergebnis des Einlesens eines Dokuments erhält er einen *Document*-Knoten, von welchem aus er beliebig auf den Inhalt des Baumes zugreifen oder ihn verändern kann. Natürlich braucht das bei den gewöhnlichen Implementierungen wesentlich mehr Platz im Speicher als übliche SAX-Implementierungen, doch dafür erhält man eine komfortable Möglichkeit, das XML-Dokument zu verwenden. Dies ist besonders dann interessant, wenn die Strukturen des XML-Dokuments nicht die internen Datenstrukturen des verwendenden Programms widerspiegeln. In diesem Fall können dennoch in einer der Logik der internen Datenstrukturen folgenden Reihenfolge die Daten aus dem XML-Dokument gewonnen werden, statt sich an die durch das Dokument vorgegebene Struktur halten zu müssen, wie es mit der SAX der Fall wäre.

DOM ist modular aufgebaut. Sie besteht bislang (2003) aus drei *Leveln* (*Ebenen*), die wiederum in verschiedene Module aufgeteilt sind.

DOM Level 1 stellt alle zur Navigation, zum Erweitern, Löschen und Modifizieren relevanten Schnittstellen zur Verfügung. Sie wurde bereits im Oktober 1998 von der W3C verabschiedet ([ABC+98]).

DOM Level 2 Core ist insbesondere um die Möglichkeiten von *Namespaces in XML*, Namensräumen, erweitert ([HHW+00]). Level 2 verfügt neben der Core noch folgende

Module: **Views** (Schnittstelle zur dynamischen Aktualisierung der Repräsentation eines Dokuments, [HC00]), **Events** (generisches Ereignismodell, [Pix00]), **Style** (Schnittstelle zur Unterstützung von *StyleSheets* und *CSS*, [AWH00]), **Traversal and Range** (generisches Iterieren und Erstellen von bestimmten Untermengen, [ACK+00]), und **HTML** (Erweiterung der HTML-Fähigkeiten, [SHH03]). Diese Arbeit bietet von diesen Modulen nur für Level 2 Core ein Binding an.

Von den Modulen des **DOM Level 3** ist noch keines vom W3C als *Recommendation* verabschiedet worden. Voraussichtlich wird Level 3 folgende Module beinhalten: **Core** (Basis der restlichen Module dieses Levels, unterstützt benutzerdefinierte Objekte, relative URIs und mehr, [HHW+03]), **Load and Save** (generische Methoden zum Laden und Speichern von XML-Dokumenten, [SH03]), **Validation** (bietet eine bessere Zusammenarbeit mit DTDs oder Schemata, um leichter zu validierende Dokumente zu erstellen, [CKR03]), **Events** (Erweiterung des Ereignismodells aus Level 2, [HP03]), und **XPath** (Einbinden der XPath-Möglichkeiten ([CD99]) in die DOM, [Whi03]).

Aufgrund der nicht vorliegenden Standardisierung und der nur experimentellen und unvollständigen Umsetzung in der zugrunde liegenden Implementierung wurde *Level 3 Core* und *Level 3 Load and Save* nur teilweise umgesetzt, und das auch nur insofern sie für die Erfüllung der Aufgabe relevant sind.

Weitere Level der DOM sind zu erwarten, da in den bereits vorliegenden Modulen häufig noch auf „später zu standardisierende Methoden“ verwiesen wird. Die Erweiterung um weitere Module und Level wird für XML4Ada95 dann, sobald sie standardisiert und implementiert sind, ohne größere Schwierigkeiten möglich sein.

Eine nicht vorhandene Standardisierung bedeutet natürlich nicht, dass die darin beschriebenen Möglichkeiten nicht bereits zur Verfügung stehen. Viele XML-Pakete boten seit jeher Fähigkeiten an, die später noch standardisiert wurden.

2.5. Warum das Ganze? - GXL und Bauhaus

Im Rahmen des Projektes Bauhaus ([EKP+99]) an der Universität Stuttgart wird schon seit jeher mit Graphen gearbeitet. **GXL** (**Graph Exchange Language**, *Graphenaustauschsprache*) wurde entwickelt, um Graphen einfach zwischen verschiedenen Werkzeugen austauschen zu können ([HWS00]). Der Code zum Austausch von Graphen in GXL des Projektes Bauhaus ist in Ada 95 geschrieben - doch ist die Unterstützung von XML in Ada 95 noch sehr mangelhaft.

Zwar existiert ein einfacher XML-Parser aus dem GtkAda-Paket ([BBCS]), der auch bislang verwendet wurde, doch ist dieser, unter anderen Mängeln, nicht in der Lage, zu validieren. Das wohl vielversprechendste Projekt in dieser Richtung ist das unter der GMGPL lizenzierte **XmlAda** von Emmanuel Briot ([Bri]). Doch leider ist auch dieses nicht gegen DTDs oder gar Schemata validierend, und hat, laut Briot und nach eigener Inspektion, eine nur unbefriedigende DOM-Unterstützung (siehe dazu auch die Kapitel 3 und 7).

Erster und wichtigster Schritt dieser Arbeit ist somit die Bereitstellung einer DOM-Schnittstelle für Ada 95. Darauf aufbauend wird ein Lese- und Schreibmodul für GXL-Dokumente innerhalb des Bauhaus-Projektes erstellt (Kapitel 6).

Eine entscheidende Frage ist hierbei, ob eine native Implementierung der DOM angeboten werden soll oder ein Binding an eine andere Programmiersprache. Eine native Implementierung ist deutlich zu bevorzugen: sie ist einfacher in bestehende Ada 95-Projekte zu integrieren und man kann in ihr alle Möglichkeiten der Sprache nutzen. Es ist zwar davon auszugehen, dass eine eventuelle Weiterentwicklung von Ada 95 standardmäßig XML-Fähigkeiten mitbringen wird ([Taf02]), doch ist es keine Option bis dahin zu warten.

So vorteilhaft eine native Implementierung auch wäre, muss sie sich doch entscheidenden Gegenargumenten beugen:

- die Erstellung eines nativen XML-Pakets für Ada 95 ist eine zu umfangreiche Aufgabe für eine einzelne Diplomarbeit (die starke Modularisierung der XML-Standards und der bereits zur Verfügung stehende Kern von XmlAda jedoch kann zu einer Bearbeitung dieser Aufgabe im universitären Umfeld genutzt werden, siehe dazu auch Kapitel 8)

- eine Implementierung in C oder C++ weist eine große *Community* und ein hohe Zahl von Nutzern auf; dadurch können zahlreiche Fehler im Programm aufgedeckt und korrigiert werden. Ob eine solche Aktivität für Ada 95 aufkommen würde, ist leider fraglich

Deshalb bietet diese Arbeit ein DOM-Binding in Ada 95 an. Dies ist nur eine vorübergehende, dafür aber sofort nutzbare Lösung, bis eine entsprechende native Implementierung vorliegt.

3. Auf der Suche nach dem Richtigen

Neo: "But if you already know, how can I make a choice?"
The Oracle: "Because you didn't come here
to make a choice, you've already made it.
You're here to try to understand why you made it."
(Matrix Reloaded)

Wie im letzten Kapitel dargelegt, gilt es, ein Binding für ein XML-Paket in Ada 95 zu erstellen. Zunächst müssen wir uns für eine zugrunde legende Implementierung entscheiden - doch die Auswahl scheint auf den ersten Blick gigantisch. Seit der Standardisierung von XML 1998 sind viele Dutzend Projekte an den Start getreten.

Welche Implementierung soll gewählt werden, und warum? Im Folgenden werden zunächst die für dieses Projekt relevanten Voraussetzungen und Einschränkungen festgehalten. Mit Hilfe dieser Liste kann die anfänglich so umfangreich erscheinende Zahl möglicher Implementierungen schnell auf ein überschaubares Maß reduziert werden, aus der abschließend ein Kandidat ausgewählt wird.

3.1. Voraussetzungen

Schon aus der Aufgabenstellung heraus ist klar, dass nur **validierende XML-Parser**, welche die **DOM-Schnittstelle** zur Verfügung stellen, in Frage kommen. Die Aufgabenstellung spezifiziert nicht, ob der Parser gegen *DTD* oder *XML Schema* validieren muss. Der ideale Kandidat validiert beides. Da das Paket aber insbesondere für die Benutzung mit *GXL* gedacht ist, sollte es auf alle Fälle gegen eine vorhandene *DTD* validieren können, da für die *GXL* eine solche existiert ([HWS00]).

Weiterhin wird in der Aufgabenstellung die **Performanz** angesprochen. Da damit zu rechnen ist, dass es durch die Sprachanbindung an Ada 95 möglicherweise zu einem Performanzverlust kommen wird, ist bei dem Kandidaten umso mehr auf brauchbare Performanz zu achten. **Robustheit** und **Vollständigkeit** sind weitere, bei weitem nicht triviale Bedingungen aus der Aufgabenstellung.

Des Weiteren ergeben sich folgende technische Einschränkungen: der Kandidat sollte eine Schnittstelle idealerweise in **C**, **COBOL** oder **Fortran** anbieten. Für diese drei Sprachen existieren in dem Paket *Interfaces* standardisierte Möglichkeiten, um sie in Ada 95 einzubinden ([ARM95]). Durch die Technik in Kapitel 4 werden zudem Kandidaten in **C++** ermöglicht, jedoch möglicherweise unter weiteren Performanzeinbußen. Kandidaten in Sprachen wie Java und Python werden nicht berücksichtigt.

Der Kandidat sollte zudem auf möglichst vielen **Plattformen** laufen, mindestens jedoch auf den Zielplattformen des Bauhaus-Projekts. Weitere Plattformen wären wünschenswert, um die größere Verbreitung von XML4Ada95 zu ermöglichen, sind jedoch nicht Bedingung.

Schließlich gibt es noch **rechtliche Einschränkungen**. Da der Parser aus der Motivation heraus geschrieben wurde, dem Bauhaus-Projekt einen XML-Parser zur Verfügung zu stellen, sollte er rechtlich in das Bauhaus-Projekt einbindbar sein. Das bedeutet, dass weder Parser möglich sind, die Lizenzgebühren für Benutzung und Einbindung verlangen, noch dass Kandidaten, die unter der GPL ([Sta91]) veröffentlicht wurden, verwendet werden können.

Durch diesen ersten Sieb werden tatsächlich nur eine kleine Anzahl Kandidaten übrig bleiben.

3.2. Eine erste Auswahl

Hatte ich Anfangs tatsächlich eine möglichst vollständige Auflistung der verfügbaren XML-Parser vor, verwarf ich diese Idee sehr schnell wieder. In [Gar] werden über vier Dutzend Parser und zahlreiche weitere Pakete aufgeführt, und auch diese Liste ist nicht vollständig.

Dennoch ist aus verschiedenen Gründen eine erste Auswahl recht einfach zu treffen: Vollständigkeit bezüglich der DOM-Spezifikation lässt die meisten Projekte wegfallen, da diese zwar XML parsen können, häufig aber nur eigene Schnittstellen zum Auslesen der Daten anbieten. Auch inkompatible Lizenzen sind (meistens, vgl 3.2.5) auf den ersten Blick zu erkennen, ebenso wie unpassende Programmiersprachen und Plattformen. Nach diesem ersten Sieb bleiben - ohne Anspruch auf Vollständigkeit - folgende Kandidaten übrig.

Die Anmerkungen beziehen sich auf den Stand Mitte September 2003.

3.2.1. Arabica

Quelle: [Hig]
Sprache: C++
Lizenz: BSD-Lizenz [BSD98]
Validierend: Nein
DOM: Ja
SAX: Ja
Plattformen: Alle, auf denen C++ läuft, ansonsten abhängig von der zu Grunde liegenden Implementierung

Arabica ist ein in C++ geschriebener Paket, welches einen standardisierten Zugriff auf SAX und DOM für C++ anbieten will. Da das W3C lediglich standardisierte Schnittstellen für Java und ECMAScript festgelegt hat, variiert die genaue DOM-Schnittstelle unter C++ mit den verschiedenen Implementierungen teilweise erheblich. Jez Higgins' Ziel ist es, durch eine standardisierte C++-Schnittstelle auch unter C++ das Austauschen der tatsächlich verwendeten Implementierung zu ermöglichen, ohne dass dabei der Quelltext des Projekts verändert werden muss.

Wiewohl dies ein hehres und wünschenswertes Ziel ist, so deuten die Ergebnisse von XML Benchmark [Chi03] darauf hin, dass Arabica noch Schwierigkeiten mit der DOM-Schnittstelle hat und dass die SAX-Schnittstelle etwa den eXpat (3.2.2) um den Faktor 10 verlangsamt.

Arabica kann auf MSXML ([Micb]), eXpat (3.2.2), libxml (3.2.3) und Xerces (3.2.4) arbeiten. Die anfangs angedachte Möglichkeit, die Ada 95-Schnittstelle über Arabica anzubinden, und so konkret verschiedenen Implementierungen ausprobieren zu können, wurde ob der Geschwindigkeitsprobleme verworfen. Außerdem bietet Arabica keine Schnittstelle zur Validierung an.

3.2.2. eXpat

Quelle: [CDP]
Sprache: C
Lizenz: MIT-Lizenz [MIT]
Validierend: Nein
DOM: Nicht von Haus aus, aber mit *Arabica* (siehe 3.2.1
SAX: Nicht von Haus aus, aber mit *Arabica* (siehe 3.2.1
Plattformen: Laut [Chi03] Windows, Linux QNX, LynxOS, BSD, Solaris, MacOS X, HP-UX, IRIX, AIX, OS/2, OpenUnix, OpenVMS; laut eigenen Angaben gar plattformunabhängig

Der 1998 von James Clark begonnene eXpat (XML Parser Toolkit) ist einer der in der Open Source-Gemeinschaft beliebtesten XML-Parser überhaupt. Inzwischen wird eXpat von einem eigenem Team weitergeführt, unter Anleitung von Clark Cooper, Fred Drake und Paul Prescod. Der Parser erfreut sich seiner Beliebtheit vor allem ob seiner Größe, Geschwindigkeit und der Einfachheit, mit der Sprachanbindungen geschrieben werden können. Solche bestehen vor allem für Skriptsprachen wie Perl, PHP, Python, Ruby oder Tcl/Tk.

Trotz dieser offensichtlichen Vorteile gerade für Anbindungen ist der Parser aus einer näheren Betrachtung leider auszuschließen. Weder kann er validieren, noch beherrscht er die Standardschnittstellen DOM und SAX. Wenn auch letzteres mit Hilfe von Arabica gelöst werden kann (auf Kosten der einfachen Anbindbarkeit, wohlgemerkt, denn Arabica ist in C++ verfasst), so gibt es doch kein einfaches Validierungsmodul.

CenterPoint XML ([CPX]) erweitert eXpat ebenfalls um SAX- und DOM-Fähigkeiten, allerdings auch in C++.

3.2.3. libxml

Quelle: [Vei]
Sprache: C
Lizenz: MIT-Lizenz [MIT]
Validierend: gegen DTDs
DOM: mit gdome2 oder Arabica
SAX: mit Arabica

Plattformen: Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC OS, OS/2, VMS, QNX, MVS

Daniel Veillard wartet diesen extrem schnellen und portablen Parser. Ursprünglich für das Gnome-Projekt geschrieben, wird er nunmehr auch von vielen anderen Projekten verwendet.

Zusammen mit dem von Paolo Casarini unter der LGPL veröffentlichten Modul gdome2 ([Cas]) ist der Zugriff auf XML-Dokumente über eine DOM-Schnittstelle möglich.

Da die DOM offenbar unter Berücksichtigung des objektorientierten Paradigmas entworfen wurde, musste bei der Entwicklung von gdome2 einiges angepasst werden, um eine Übersetzung in das imperative Programmierparadigma von C zu erlauben ([CP01]). Dies führte insbesondere zu einer Menge Schreibarbeit bei den ableitenden Schnittstellen: Schnittstellen, wie *Text* oder *Comment* werden extrem umfangreich, indem sie von mächtigen Schnittstellen wie *Node* erben. All diese Methodenaufrufe müssen nun explizit angeboten werden, da ja eine Schnittstellenvererbung wie in C++ oder Ada 95 nicht möglich ist.

Auch bei einer Anbindung an Ada 95 hätte das dazu geführt, dass die objektorientierten Möglichkeiten der Programmiersprache nicht ausgenutzt werden könnten. Dafür hätte man aber eine standardisierte, einfache Anbindung, da gdome2 wie auch libxml vollständig in C geschrieben sind.

3.2.4. Xerces

Quelle: [Apa]
Sprache: C++
Lizenz: Apache-Lizenz [Apa99]
Validierend: DTDs und Schemata
DOM: Ja
SAX: Ja
Plattformen: Linux, HP-UX, AIX, Windows, Solaris, OS/390, AS/400, FreeBSD, SGI IRIX, Macintosh, OS/2, PTX, UnixWare und mehr

Xerces ist der XML-Parser der Apache Group. Er wird sowohl in C++ als auch in Java gewartet und setzt die Empfehlungen des W3C unter den Open Source-Produkten mit am schnellsten um. Des Weiteren ist Xerces sehr umfangreich dokumentiert, bietet Validierungen sowohl gegen DTDs als auch gegen XML Schema an, bietet Schnittstellen in DOM und SAX an und wird auf einer Unmenge von Plattformen gewartet.

Nachteilig ist jedoch, dass das Paket vollständig in C++ geschrieben ist, wozu in Ada 95 keine standardisierte Schnittstelle vorhanden ist.

IBMs *XML4C* ([IBM99]) basiert auf dem Xerces Paket und ist noch durch IBMs Unicode-Pakete erweitert. Entgegen des Namens ist auch dieses kein C- sondern ein C++-Paket.

3.2.5. Oracle XDK

Quelle: [Oraa]
Sprache: C++ (mit einem C-Wrapper)
Lizenz: Oracle Technology Network license [Orab]
Validierend: DTDs und Schemata
DOM: Ja
SAX: Ja
Plattformen: Linux, HP-UX, Windows, Solaris

Obwohl Oracles *XDK* (*XML Developers Kit*) ein ansonsten recht vielversprechendes Paket anbietet (alles, was auch Xerces kann, dazu noch weitergehende DOM-Unterstützung und bereits von Haus aus einen C-Wrapper, den man für die Anbindung an Ada 95 ohnehin erstellen muss), disqualifiziert es sich durch seine restriktive Lizenz.

Durch die Lizenz wäre es notwendig, jedes das Paket benutzende Programm mit einer ebensolchen Lizenz auszustatten. Diese wiederum verbietet, aufgrund von Bestimmung des US-Handelsministeriums, den Export nach Kuba, Iran, Sudan, Irak, Libyen und Syrien und setzt weitere Einschränkungen ([Orab]).

Hinzu kommt, dass die Benutzung des Pakets zwecks Benchmarking oder zur Analyse der Software verboten wird. Des Weiteren ist das Herabladen der Softwarepakets aus dem Internet, ja selbst die Einsicht in die Lizenz von Oracle, erst nach einer Registrierung auf der Website möglich. Schließlich ist das Paket insbesondere auf die Zusammenarbeit mit Oracles kommerziellen Datenbankprodukten abgestimmt.

3.3. Spieglein, Spieglein an der Wand... - Die Entscheidung

Betrachtet man sich die Kandidaten aus dem vorhergehenden Kapitel, bleiben zwei Favoriten übrig, *Xerces* und *libxml*. *eXpat* fällt aufgrund seiner mangelnden Validierungsfähigkeit aus, *Arabica* wegen den Performanzeinbußen und *XDK* insbesondere wegen der Lizenz und seiner technischen Anpassung an Oracle. Sowohl *Xerces* wie auch *libxml* erfüllen hingegen alle in Kapitel 3.1 genannten Bedingungen.

Beide Kandidaten würden wahrscheinlich zu erfolgreichen Ada 95-Bindings führen, und für beide gibt es starke Argumente. Ich will versuchen, sie hier zusammenzufassen.

Für die Kombination *libxml*/*gdome2* spricht:

- die hervorragende Geschwindigkeit. Es ist laut ([Chi03]) das schnellste XML-Paket zur Zeit.
- es ist in C geschrieben, wofür es eine standardisierte Schnittstellen in Ada 95 gibt
- es ist aktuell und wird gewartet, zudem ist *libxml* häufig Bestandteil von Linux-Standarddistributionen, was zu einer großen Zahl von Nutzern führt
- die hohe Zahl von unterstützen Plattformen
- es ist Open Source, *libxml* ist unter der sehr liberalen MIT License veröffentlicht

Dagegen spricht jedoch:

- die Übertragung der DOM in C verursacht ein starkes Anwachsen der Zahl der Methoden in vererbenden Schnittstellen. Gerade bei der Erweiterung der DOM durch weitere Module und *Level* kann dies zu Mehrarbeit bei der Wartung führen.

- das Paket bietet keine Validierung gegen XML Schema

- *gdome2* ist unter der LGPL lizensiert, welche die Weiterverwendung des Codes in eigenen Projekten einschränkt (nicht jedoch des Pakets)

Für *Xerces* spricht:

- es ist aktuell und wird gewartet, zudem ist es, als Teil des Apache-Projekts, ein Paket mit einer hohen Zahl von Nutzern

- die sehr hohe Zahl von unterstützen Plattformen

- es ist Open Source und unter der sehr liberalen Apache License veröffentlicht

- Validiert sowohl gegen XML Schema als auch gegen DTDs

- es bietet bereits eine experimentelle Implementierung von DOM Level 3 Core und DOM Level 3 Load and Save (vgl. Kapitel 2.4). Die Hoffnung, dass bis zum Ende der Arbeit das W3C die beiden Module als Standards verabschieden und Xerces eine endgültige Implementierung anbieten würde, so dass es noch Bestandteil des Bindings werden könnte, erfüllte sich allerdings nicht

- ich habe bereits Erfahrungen mit Xerces sammeln können und verwende das Paket schon seit Monaten

Gegen *Xerces* spricht schließlich:

- es ist in C++ geschrieben, nicht in C - Ada 95 bietet für C++ keine standardisierte Schnittstelle an

- das Paket ist vergleichsweise umfangreich

Nach dieser Auflistung fiel die Entscheidung auf *Xerces*. Insbesondere die bereits vorhandene Erfahrung mit dem Paket und die ansonsten hervorragend erfüllten Voraussetzungen sprachen dafür. Die Probleme, die dadurch entstanden, dass Xerces in C++ geschrieben ist, wurden zu einer steten Herausforderung bei der Implementierung des Bindings. Sie sind in Kapitel 4 genauer dargestellt - ebenso wie deren Lösung.

3.4. A#

Es sollte noch auf eine weitere Möglichkeit hingewiesen werden, von Ada aus auf XML-Pakete zuzugreifen. Ein Projekt, dass von der US Air Force Academy gestartet wurde, zielt auf die Integration von Ada in die *.Net*-Plattform ([CSH02]).

Die *.Net*-Plattform bietet eine gemeinsame Laufzeitumgebung für Programme und Programmteile, die in verschiedenen Programmiersprachen geschrieben und dennoch miteinander interagieren können ([Mica]). Das geht soweit, dass nicht nur Schnittstellen, die in anderen Sprachen verfasst wurden, benutzt werden können, sondern es ist sogar möglich, eine Klasse, die in Sprache A geschrieben ist, in Sprache B zu beerben und zu erweitern, um sie dann in Sprache C zu instantiieren.

Da die *.Net*-Plattform von Haus aus mit einem umfangreichen XML-Paket ausgestattet ist, könnte man vom in *.Net* integriertem Ada aus auf dieses XML-Paket zugreifen. Das Binding steckt dann bereits in der zugrunde liegenden Plattform.

Diese Lösung ist leider für die vorliegende Aufgabenstellung nicht anwendbar. Die *.Net*-Plattform, obwohl theoretisch plattformunabhängig, steht zur Zeit nur aktuellen Windowsversionen zur Verfügung. Implementierungen ([Xim], [Dot]) für weitere Betriebssysteme haben derzeit nur experimentellen Status. Des Weiteren ist unklar, wie weit native Ada 95-Projekte auf die *.Net*-Plattform portierbar sind, und ob dann noch eine für das Projekt ausreichende Performanz und Stabilität sichergestellt ist.

4. C++, Lady Ada zu Diensten

“There are two kinds of researchers: those that have implemented something and those that have not. The latter will tell you that there are 142 ways of doing things and that there isn’t consensus on which is best. The former will simply tell you that 141 of them don’t work.“
(David Cheriton)

Ada 95 ist standardmäßig schon reichlich kommunikativ, wie es sich für eine Dame aus gutem Hause gehört: das Referenzhandbuch ([ARM95]) sieht ein Schnittstellenpaket vor (*Interfaces*), mit dem sich leicht auf Code zugreifen lässt, der in C, Cobol oder Fortran geschrieben wurde. Über Pragmas sind solche Funktionen einbindbar.

```
procedure CFunction(X : in Interfaces.C.Int);
pragma Import(C, cfunction);
```

Damit kann auf folgende Funktion in C zurückgegriffen werden:

```
void cfunction(int X) { ... }
```

Dabei werden beide Quelltextdateien einzeln kompiliert, mit den jeweiligen Spracheinstellungen. Die Objektdatei, die aus dem C-Quelltext generiert wurde, wird dem Linker mitgegeben. Natürlich müssen die Werkzeuge dahingehend kompatibel sein.

Eine Schwierigkeit ist hierbei, dass C im Gegensatz zu Ada 95 zwischen Groß- und Kleinbuchstaben unterscheidet. Die Lösung dieses Problems ist jedoch abhängig vom jeweils verwendeten Linker. Das *pragma External_Name_Casing* kann dies bei *gnat* und *gcc* einstellen, wobei laut [gcc] der Compiler so voreingestellt ist, dass stets Kleinbuchstaben verwendet werden.

Die Namen der übergebenen Parameter spielen übrigens keine Rolle.

Durch diese Möglichkeiten kann Ada 95 eine gewaltige Menge bereits geschriebenen Codes beerben und weiterhin nutzen.

Das von uns ausgewählte Paket, *Xerces*, ist jedoch in einer „portierbaren Untermenge von C++“ ([Apa]) geschrieben. Die weite Verbreitung von C++ und die zahlreichen in C++ geschriebenen Pakete machen es des Weiteren ohnehin wünschenswert, dass Ada auch auf C++ zugreifen kann.

GNAT ([grm]) hat eine integrierte Unterstützung zum Aufruf von Funktionen in C++, in dem es für das Import-Pragma ebenfalls die Konvention CPP für C++ erlaubt. Doch

dieses Pragma ist nicht Teil des Ada 95-Standards und soll deswegen hier nicht verwendet werden.

Wie also können wir C++ dazu überreden, sich mit Ada 95 zu verständigen? Indem wir C++ als C maskieren! - dieses nämlich wiederum versteht Ada 95. Wir legen eine eigene C++-Datei an, in welcher die benötigten Funktionen zugreifbar gemacht werden. Das sieht prinzipiell wie folgt aus:

cpp1.cpp enthält die tatsächliche Implementierung:

```
void cppfunction(int i) { ... }
```

cppwrap.cpp maskiert nur die eigentliche Funktion:

```
#include "cpp1.cpp"
interface "C" void call_cppfunction(int i) {
  cppfunction(i);
}
```

Natürlich hätte man dies auch in einem Schritt machen können, wie im folgenden Beispiel, fakec.cpp:

```
interface "C" void cppfunction(int i) { ... }
```

Doch in diesem Fall müsste man den bestehenden Code der aufzurufenden Bibliothek ändern und diese Änderung mit jedem Update dieser Bibliothek nachholen. So lange jedoch die Schnittstelle stabil bleibt, bedarf die erste Methode keiner weiteren Wartung bei Erscheinen einer neuen Version.

Am Beginn der Diplomarbeit lag Xerces in der Version 2.2.0 vor, doch während der Arbeit erschien die Version 2.3.0. Wie geplant konnte die neue Version ohne jegliche Änderung am Binding verwendet werden, da die Änderungen an Xerces nicht die Schnittstellen betraf.

Im folgenden sollen verschiedene Detailfragen geklärt werden, die beim Erstellen von Schnittstellen von C++ für Ada 95 auftreten können.

4.1. Objektorientierung

Ada 95 und C++ beherrschen das objektorientierte Paradigma. Doch sie müssen über den Umweg C miteinander kommunizieren - und C hat von Objekten keine Ahnung.

Das erfordert folgendes Muster zum Aufrufen von Methoden.

cppclass.cpp enthält die Klasse:

```
class CPPClass {
public:
  void memberfunction(int i) { ... }
  int member;
};
```

cppclasswrap.cpp maskiert die Klasse mit einer prozeduralen Schnittstelle:

```
#include "cppclass.cpp"
interface "C"
void call_cppclass_memberfunction(CPPClass* this, int i) {
  this->memberfunction(i);
}

interface "C"
void get_cppclass_member(CPPClass* this, int& i) {
  i = this->member;
}

interface "C"
void set_cppclass_member(CPPClass* this, int i) {
  this->member = i;
}
```

adacalling.ads bietet diese Schnittstelle für Ada an:

```
use System;
package CPP_Class is
  type CPP_Class is private;
  procedure Memberfunction(Self : in CPP_Class; i : in Integer);
  function Get_Member(Self : in CPP_Class) return Integer;
  procedure Set_Member(Self : in CPP_Class; i : in Integer);

private
  subtype C_Pointer is System.Address;
  C_Null : constant C_Pointer := System.Null_Address;

  type CPP_Class is record
    This : C_Pointer := C_Null;
  end record;
end CPP_Class;
```

adacalling.adb schließlich implementiert das Binding:

```
package body CPP_Class is
  procedure Call_Memberfunction(this : in C_Pointer;
                                i : in Interfaces.C.Int);
  pragma Import(C, call_cppclass_memberfunction);

  procedure Memberfunction(Self : in CPP_Class; i : in Integer) is
  begin
    Call_Memberfunction(Self.This, Interfaces.C.Int(i));
  end Memberfunction;

  procedure Call_Get_Member(this : in C_Pointer;
                            i : out Interfaces.C.Int);
  pragma Import(C, get_cppclass_member);

  function Get_Member(Self : in CPP_Class) return Integer is
    Result : Interfaces.C.Int;
  begin
    Call_Get_Member(Self.This, Result);
    return Integer(Result);
  end Get_Member;

  procedure Call_Set_Member(this : in C_Pointer;
                            i : in Interfaces.C.Int);
  pragma Import(C, set_cppclass_member);

  procedure Set_Member(Self : in CPP_Class; i : in Integer) is
  begin
    Call_Set_Member(Self.This, i);
  end Set_Member;
end CPP_Class;
```

Das Paket CPP_Class kann jetzt genau wie jedes andere Ada-Paket verwendet werden.

Membervariablen der C++-Klasse werden in diesem Muster zudem stets über Get- und Set-Methoden verändert.

4.2. Objektkonstruktion

In C++ funktionieren Konstruktoren anders als in Ada 95. Deswegen brauchen wir ein weiteres Muster, um das Konzept von C++ zu simulieren - doch es folgt den Prinzipien des Musters aus dem vorhergehenden Abschnitt.

cppclass.cpp enthält die Klasse:

```
class CPPClass {
public:
  CPPClass() { ... }
};
```

cppclasswrap.cpp maskiert die Klasse mit einer prozeduralen Schnittstelle:

```
#include "cppclass.cpp"
interface "C" void create_cppclass(CPPClass*& this) {
  this = new CPPClass();
}
```

adacalling.ads bietet die Schnittstelle für Ada an:

```
use System;
package CPP_Class is
  type CPP_Class is private;
  function Create_CPPClass return CPP_Class;

private
  type CPP_Class is record
    This : C_Pointer;
  end record;
end CPP_Class;
```

adacalling.adb enthält die Implementierung der Schnittstelle:

```
package body CPP_Class is
  procedure Call_Create_CPPClass(this : out C_Pointer);
  pragma Import(C, create_cppclass);

  function Create_CPPClass return CPP_Class) is
    Result : CPP_Class;
  begin
    Call_Create_CPPClass(Result.This);
    return Result;
  end Create_CPPClass;
end CPP_Class;
```

Das Destruieren funktioniert nach demselben Prinzip. Man sollte lediglich daran denken, konstruierte Objekte auch tatsächlich zu destruieren.

4.3. Typsicherheit

Bezüglich der Typsicherheit trägt der Autor des Bindings die volle Verantwortung. Der Linker ist nicht in der Lage, über Sprachgrenzen hinweg die Typen der Parameter zu überprüfen. Behaupten wir, übergebene Daten gehörten zu einem bestimmten Typ, wird dies auch so verwendet.

Deswegen ist eine Kapselung hier besonders wichtig: so kann vermieden werden, dass der Benutzer aus Versehen Methodenaufrufe an das falsche Objekt schickt. Außerdem verhindert man damit jegliche Zeigerarithmetik auf der Ada 95-Seite, da die Adressen selber nicht zugänglich sind.

Um jedoch darüber hinaus weitere Sicherheiten zu unternehmen - etwa, dass nur initialisierte Objekte auch verwendet werden dürfen - muss der Autor des Bindings hier selber weitere Sicherheitsmaßnahmen einbauen. In den oben genannten Beispielen etwa wird *This* stets mit *C_Null* initialisiert. So kann dann bei jedem Methodenaufruf geprüft werden, ob das aufgerufene Objekt nicht etwa Null ist. Dies kann, je nach Belieben, auf der Ada 95- oder der C++-Seite geschehen.

In XML4Ada95 wird auf der C++-Seite die übergebene Adresse geprüft, und sollte sie Null sein wird dann mit der in Abschnitt 4.5 vorgestellten Methode eine Ausnahme signalisiert.

4.4. Polymorphie

Aus den vorhergehenden Abschnitten folgt noch ein weiteres Problem beim Binding. Angenommen die C-Schnittstelle sieht wie folgt aus:

```
interface "C"
void call_cppclass_memberfunction(CPPClass* this, int i) {
  this->memberfunction(i);
}
```

In diesem Fall muss der Zeiger *this* auch wirklich auf ein Objekt der Klasse *CPPClass* zeigen, und nicht etwa auf eines der Klasse *CPPDerivedClass*, selbst wenn *CPPDerivedClass* von *CPPClass* abgeleitet sein sollte.

Dies bedeutet leider, dass Folgendes in Ada 95 nicht möglich ist. Gegeben seien folgende zwei Ada-Typen:

```
package A is
  type A_Type is tagged ...;
  procedure Do_It(Self : A_Type);
end A;

package B is
  type B_Type is new A.A_Type with ...;
end B;
```

Weiterhin existieren folgende C++-Klassen.

```
class AClass {
public:
  Do_It() { ... }
};

class BClass : public AClass {};
```

Angenommen *Do_It* rufe wie oben beschrieben die C++-Memberfunktion *Do_It* eines Objekts der Klasse *AClass* auf. Der Benutzer kann die Ada 95-Schnittstelle dann wie folgt verwenden:

```
declare
  B_Object : B.B_Type;
begin
  B.Do_It(B_Object);
end;
```

Dieser Code wird ohne Fehlermeldungen kompilieren und linken, mit etwas Glück würde er sogar laufen. Doch leider ist das Ergebnis nicht das erhoffte: es wird nicht die abgeleitete Funktion *Do_It* aufgerufen, sondern der Aufruf zeigt auf einen undefinierten Ort im Speicher. Um Polymorphie zu simulieren, müssen wir sie per Hand (oder Generator) nachprogrammieren. Jede abgeleitete Prozedur und Funktion muss explizit auf ein entsprechendes C-Binding verweisen. Allein für dieses einfache Beispiel würde das wie folgt aussehen:

```
package A is
  type A_Type is tagged ...;
  procedure Do_It(Self : A_Type);
end A;

package B is
  type B_Type is new A.A_Type with ...;
  procedure Do_It(Self : B_Type);
end B;

package body A is
  procedure Call_Do_It(this : in C_Pointer);
  pragma Import(C, call_atype_do_it);

  procedure Do_It(Self : A_Type) is
  begin
    Call_Do_It(Self.This);
  end Do_It;
end A;
```

```
package body B is
  procedure Call_Do_It(this : in C_Pointer);
  pragma Import(C, call_btype_do_it);

  procedure Do_It(Self : B_Type) is
  begin
    Call_Do_It(Self.This);
  end Do_It;
end B;
```

Dazu folgender C-Wrapper:

```
interface "C" void call_aclass_do_it(AClass* this) {
  this->Do_It();
}

interface "C" void call_bclass_do_it(BClass* this) {
  this->Do_It();
}
```

Die C++-Implementierung bleibt die selbe wie oben angegeben.

Wir sehen, dass dies bei realistisch komplexen Klassen oder gar Klassenhierarchien sehr schnell zu sehr vielem und äußerst schwer wartbarem Code führen würde.

In XML4Ada95 wird deshalb auf Vererbung verzichtet. Dies wird in Kapitel 5 nochmal ausführlicher dargestellt. Natürlich ist das keine allgemeine Lösung. Die Spezifikation der DOM verzichtet jedoch darauf, Funktionen zu überschreiben, und dadurch ist es möglich, die Schnittstellen ohne Vererbung zu implementieren. Andere Umstände werden entsprechend angepasste Lösungen verlangen, die für den Einzelfall ausgearbeitet werden müssen.

4.5. Ausnahmebehandlung

Es besteht keine Möglichkeit, Ausnahmen als solche über die Sprachgrenzen hinweg zu propagieren. Zudem kennt C das Konzept von Ausnahmen gar nicht, also muss auch hier wieder auf der C++-Seite das Konzept C-gerecht verpackt und dann in Ada 95 wieder aufgelöst werden. Dazu wurde folgendes Muster verwendet.

Angenommen, `void func() {};` sei die aufzurufende C++-Bibliotheksfunktion. Sie könne eine *funcException* auslösen. Dann erstellen wir folgenden C-Wrapper:

```
const int OK = 0;
const int FUNCEXCEPTION = 1;
const int UNEXPECTED_EXCEPTION = -1;
```

```
extern "C" void call_func(int& errorcode) {
  errorcode = OK;
  try {
    func();
  } catch (funcExcpetion x) {
    errorcode = FUNCEXCEPTION;
  } catch (...) {
    errorcode = UNEXPECTED_EXCEPTION;
  }
}
```

Jede zu identifizierende Ausnahme wird mit einer Zahl codiert, die dann innerhalb eines weiteren Parameters an Ada 95 weitergereicht wird. Die Codierung sollte natürlich in einer symbolischen Konstante festgehalten werden, und keineswegs nur als eine Zahl im Code.

Die Ada 95-Schnittstelle ändert sich bei diesem Muster im Vergleich zu den vorhergehenden Schnittstellen nicht. Die Implementierung der Schnittstelle jedoch durchaus:

```
procedure Call_Func(Error_Code : out Interfaces.C.Int);
pragma Import(C, call_func);

procedure Func is
  Error_Code : Interfaces.C.Int;
begin
  Call_Func(Error_Code);
  if not (Error_Code = 0) then
    if Error_Code = 1) then
      raise Func_Exception;
    end if;
    if Error_Code = -1 then
      raise Unknown_Exception;
    end if;
  end if;
end Func;
```

Um den Programmtext übersichtlicher zu halten, und damit die Ausnahmebehandlung nicht durch ihren schieren Umfang die eigentliche Programmlogik überdeckt, sollte die Ausnahmebehandlung ausgelagert werden.

```
  procedure Handle_Error(Error_Code : in Interfaces.C.Int) is
  begin
    if not (Error_Code = 0) then
      ... wie oben ...
    end if;
  end Handle_Error;
```

```
  procedure Call_Func(Error_Code : out Interfaces.C.Int);
  pragma Import(C, call_func);

  procedure Func is
    Error_Code : Interfaces.C.Int;
  begin
    Call_Func(Error_Code);
    Handle_Error(Error_Code);
  end Func;
```

Durch ein `pragma Inline(Handle_Error);` sollten die Performanzeinbußen verringert werden können. Diese zweite Lösung hat zudem das Problem, dass das Rückverfolgen der Ausnahmen stets dazu führt, dass die Ausnahmen in der Prozedur *Handle_Error* ausgelöst werden.

4.6. Strings

Wie üblich sind Strings ein besondere Beachtung verdienendes Thema. Wieder verfügen sowohl Ada 95 wie auch C++ über ausgereifte Datentypen, die sich mit Strings beschäftigen, während C hier seit jeher mit Zeigern auf Zeichen und einem Abschlusszeichen arbeitet.

Diesmal unterscheiden sich die Methoden, um Strings von einer Sprache in die andere zu übertragen. Dies liegt an den sich unterscheidenden Lösungen, die C++ und Ada 95 für ihre eigenen Strings verwenden.

Um einen String von Ada 95 nach C++ zu bringen, geht man wie folgt vor. Hier die Ada-Implementierung:

```
procedure Call_Set_String(S : Interfaces.C.Strings.Chars_Ptr);
pragma Interface(C, call_set_string);

procedure Set_String (S : in Standard.String) is
  C_String : Interfaces.C.Strings.chars_ptr
               := Interfaces.C.Strings.New_String(S);
  begin
    Call_Set_String(C_String);
    Interfaces.C.Strings.Free(C_String);
  end Set_String;
```

Und hier die C-Schnittstelle, die den String einliest:

```
interface "C" void call_set_string(char* s) {
  std::string str = s;
  ...
}
```

Man beachte, dass in *str* nunmehr eine Kopie des ursprünglich übergebenen Strings *s* enthalten ist und deswegen der Speicher von *s* nun ruhigen Gewissens in der Ada-Prozedur freigegeben werden darf.

Die umgekehrte Richtung ist ein wenig umständlicher gelöst, was an den Stringtypen von Ada 95 liegt. Hier die C-Schnittstelle:

```
interface "C" void call_get_string(char*& s) {
  std::string str = ...;
  s = str.c_str();
}
```

Und hier die Implementierung des Ada-Bindings:

```
with Interfaces.C.Strings;
use type Interfaces.C.Strings.chars_ptr;

function Call_Get_String return Interfaces.C.Strings.Chars_Ptr;
pragma Import(C, call_get_string);

function Get_String return Standard.String is
  C_String : Interfaces.C.Strings.Chars_Ptr;
begin
  C_String := Call_Get_String;
* if C_String = Interfaces.C.Strings.Null_Ptr then
    return From_Standard_String("");
* else
*   return From_Standard_String(Interfaces.C.Strings.Value(C_String));
* end if;
end Get_String;
```

Ein Null-String (ein *char** das auf Null zeigt) kann bei der hier vorgestellten Methode nicht vorkommen und demnach können die mit Sternchen markierten Zeilen entfallen. Da aber in C (aber auch in C++ statt dem in der Standardbibliothek zur Verfügung gestelltem *std::string*) häufig noch einfache *char** zur Stringrepräsentation verwendet werden, wird die Abfrage auf Null-Strings doch noch in vielen Fällen relevant werden. Statt einem leeren String kann freilich auch ein bestimmter, einen Null-String markierender String zurückgegeben werden.

Wird die Funktion *Interfaces.C.Strings.Value* auf einen Null-String angewandt, so wird ein *Dereference_Error* ausgelöst.

4.7. Funktionsvariablen

In XML4Ada95 kann ein eigener *ErrorHandler* eingehängt werden, eine Funktion, die dann von der Implementierung aufgerufen wird, wenn es beim Lesen oder Schreiben eines XML-Dokuments zu einem Fehler kommt. Diese Funktion implementiert dann, wie der Parser sich verhält, wenn z.B. das gelesene Dokument nicht validiert.

Wie ist das Einhängen dieser Funktion implementiert? Schließlich muss diese Funktion, die in Ada 95 geschrieben ist, mit den Mitteln von Ada 95 so an eine C++-Bibliothek übergeben werden, dass diese sie aufrufen kann.

Auch hier wird von der in XML4Ada95 tatsächlich vorkommenden Komplexität abstrahiert und nur das Muster aufgezeigt, nach welchem diese Aufgabe bewältigt werden kann.

Zunächst die Schnittstelle, wie sie dem Benutzer erscheint.

```
type ErrorHandlerFunction is access
  pocedure (E : Integer);

procedure set_ErrorHandler
  (ErrorHandler : ErrorHandlerFunction);
```

Der Benutzer muss nichts weiter kennen als diese einfache Schnittstelle. Mit ihrer Hilfe kann er die entsprechende Funktion (vom Typ *ErrorHandlerFunction*) einfügen.

Die Implementierung dieser Schnittstelle sieht wie folgt aus:

```
procedure Call_ErrorHandler (E : Integer);
pragma Export(C, call_errorhandler);

Handler : ErrorHandlerFunction;

procedure Call_ErrorHandler(E : Integer) is
begin
  Handler(E);
end Call_ErrorHandler;

procedure Call_Set_ErrorHandler;
pragma Interface(C, call_domparser_set_errorhandler);

procedure Set_ErrorHandler
  (ErrorHandler : ErrorHandlerFunction) is
begin
  Handler := ErrorHandler;
  Call_Set_ErrorHandler;
end Set_ErrorHandler;
```

Sie enthält also auch eine exportierte Funktion. Diese ruft die durch *Set_ErrorHandler* in der Variable *Handler* gespeicherte Funktion auf.

Die exportierte Funktion kann dann einfach vom C/C++-Code aus aufgerufen werden. Im folgenden Beispiel gehen wir davon aus, dass die C++-Bibliothek ein Objekt einer vorgegebenen Bibliotheksklasse abgeleiteten Klasse erhält, in welcher die entsprechende Funktion überschrieben ist. So ist es auch in XML4Ada95 gelöst. Die vorgegebene Klasse heißt hier *LibErrorHandlerClass*. Benutzt die Bibliothek eine andere Vorgehensweise, muss die Methode natürlich an diese angepasst werden.

```
extern "C" void call_errorhandler(int e);

class AdaErrorHandler : public LibErrorHandlerClass {
public:
  void handleError(int e) {
    call_errorhandler(e);
  }
};

AdaErrorHandler* adaErrorHandler;

extern "C" void call_set_errorhandler() {
  setErrorHandler(adaErrorHandler);
}
```

setErrorHandler sei hierbei die Funktion, mit der in der Bibliothek das Objekt mit der überschriebenen Methode - hier *adaErrorHanlder* - registriert wird. Nun kann die C++-Bibliothek problemlos die in Ada 95 geschriebene Funktion verwenden.

5. Designentscheidungen

Ilsa Lund: "A franc for your thoughts."
Rick Blaine: "In America they'd bring only a penny, and,
huh, I guess that's about all they're worth."
Ilsa Lund: "Well, I'm willing to be overcharged. Tell me."
(Casablanca)

Dieses Kapitel begründet die Entscheidungen, die bei dem Entwurf der DOM-Schnittstelle für Ada 95 gefällt wurden. Dies dient zugleich als eine Darstellung der bei der Erstellung des Bindings gemachten Erfahrungen. Dabei geht es hier weniger um Ergebnisse, sondern um den Weg zu diesen.

5.1. Das kurze Leben von *Acidboi*

Das W3C bietet Standards für die DOM-Schnittstelle in Java, in ECMAScript und in IDL an, nicht aber für Ada 95. OMG wiederum bietet mit [OMG01] einen Standard an, um in IDL definierte Schnittstellen nach Ada 95 zu übersetzen. Der Entwurf der Schnittstelle folgte eng diesen beiden Standards. Alle etwaigen Abweichungen davon sind in diesem Kapitel erklärt.

Der ursprüngliche Plan war, das Binding weitestgehend aus den IDL-Dateien zu generieren. Dazu war das Tool **Acidboi** (***A**da 95/**C**++ **I**nterface **D**eveloper **b**ased **o**n **I**DL-specifications*) gedacht. Es sollte nicht nur die Schnittstellen für Ada 95 erstellen, sondern auch das C-Interface für die C++-Implementierung, die ja ursprünglich auch auf der IDL-Definition beruhte.

Als Grundlage für Acidboi wurde der **IDLAC** (***IDL** to **A**da **C**ompiler*) gewählt, Teil des Polyorb/AdaBroker-Projekts der Universität Paris ([HPQ]). IDLAC war für die Anwendung von CORBA mit Ada 95 gedacht. Nach kurzer Entwicklung jedoch wurde festgestellt, dass die Arbeit an Acidboi zu lange dauern würde, und zudem in einem vernünftigen Rahmen kein zufrieden stellendes Ergebnis liefern würde.

Das lag an mehreren Gründen:

- *Xerces*' Schnittstelle gründet in *Level 3* auf einer veralteten Version der DOM-Spezifikation. Eine Nachbearbeitung der Ergebnisse von Acidboi per Hand wäre unumgänglich gewesen. Generierte Ergebnisse, die von Hand nachbearbeitet werden müssen, sind jedoch aufgrund der folgenden Punkte nicht tragbar.

- Bei der Erweiterung von XML4Ada95 um weitere Level oder Module der DOM würden auch früher generierte Schnittstellen überschrieben werden. Dadurch würden von Hand hinzugefügten Änderungen zunichte.

- *Level 3 Core* und *Level 3 Load and Save* der DOM-Spezifikation sind nicht vollständig implementiert (weder in XML4Ada95 noch in *Xerces*). Dies hätte Acidboi nicht ohne weitreichende Erweiterung umsetzen können.

Also wurde auf Acidboi verzichtet. Stattdessen wurde mit einem nur leicht modifizierten IDLAC eine erste Grundlage für den Code erzeugt, und dieselbe dann stark per Hand modifiziert (etwa das Entfernen CORBA-spezifischen Codes oder das Hinzufügen der C aufrufenden Prozeduren). Da der IDLAC dem Standardmapping der OMG folgt, kann so gewährleistet werden, dass das Ada 95 Binding der DOM sich eng an die Standards hält.

5.2. Das ist doch kein Ada!

Die vom Standard vorgeschriebene und von IDLAC automatisch erzeugte Schnittstelle hält sich wiederum nicht an den von Ada 95 gewohnten *Coding Standard*. Wir erwarten, dass Funktionen Namen wie *Has_Child_Nodes* haben - und nicht *hasChildNodes*. Wir erwarten keine symbolische Konstanten, sondern Enums. Wir erwarten eine für Ada 95 typische Ausnahmebehandlung.

All das leisten die Standards nicht. Dies wird besonders auffällig bei der Ausnahmebehandlung. Aus dem IDL generiert sieht sie wie folgt aus:

```
type DOMException_Members is
  new CORBA.IDL_Exception_Members with
    record
      code : CORBA.Unsigned_Short;
    end record;

DOMException : exception;

procedure Get_Members
  (From : Ada.Exceptions.Exception_Occurrence;
   To   : out DOMException_Members);

INDEX_SIZE_ERR     : constant CORBA.Unsigned_Short := 1;
DOMSTRING_SIZE_ERR : constant CORBA.Unsigned_Short := 2;
```

Üblicherweise würden wir in Ada folgendes erwarten:

```
Index_Size_Error     : exception;
Domstring_Size_Error : exception;
```

Natürlich leisten die beiden Ausschnitte nicht dasselbe: das obere, generierte Ergebnis sieht für die gesamte DOM nur eine Art Ausnahme vor, die *DOMException*, während die untere über ein Dutzend verschiedene Ausnahmen hat (im Beispiel sind nur zwei aufgelistet). Um also alle Ausnahmen des DOM-Pakets zu behandeln, müssen sie auch explizit alle abgefangen werden. Die obere Möglichkeit hingegen erlaubt es, einfach nur die *DOMException* abzufangen und dieselbe dann bei Interesse genauer zu untersuchen.

XmlAda geht hierbei den zweiten Weg und definiert separate Ausnahmen. Auch die Namen der Funktionen und Prozeduren sind an die üblichen Schreibweise von Ada 95 angepasst. *XmlAda* erreicht so einen hohen Grad an „Ada-artigkeit“.

Die eigentliche Frage ist: wie erhält man eine möglichst hohe Akzeptanz der Schnittstelle? Standardtreue oder „Ada-artigkeit“? Um diese Frage zu beantworten, stellte ich sie sowohl in meinem Zwischenvortrag wie auch auf *comp.lang.ada* zur Diskussion. Obwohl für beide Seiten argumentiert wurde, entschloss ich mich schließlich, mich eng an den Standard zu halten. Folgende Gründe waren dabei ausschlaggebend:

- sich eng an Standards zu halten führt unmittelbar zu einer höheren Akzeptanz als Veränderungen, die wahrscheinlich nur subjektiv als Verbesserungen gesehen werden. Die Angriffsfläche für Kritik schrumpft ungemein.
- die Ausarbeitung von Standards dauert häufig sehr lange und zahlreiche Experten haben sich mit dem Thema beschäftigt. Es wäre anmaßend, eigene Änderungen zu unternehmen statt diese Vorarbeit auszunutzen.
- das exakte Beibehalten der Schreibweise des Standards erlaubt es, schneller Informationen zu einer Schnittstelle oder zu einer Prozedur dieser Schnittstelle zu finden (durch einfaches Suchen), wenn man die Spezifikation des Standards vor sich hat. Bei einer Veränderung der Schreibweise müsste man raten, wie die Funktion ursprünglich hieß.
- sollte doch ein Acidboi ähnliches Werkzeug eines Tages das Binding erstellen, ändert sich an der Schnittstelle nichts.

5.3. Zirkuläre Abhängigkeit

Die DOM enthält in ihrer Spezifikation Konstrukte, die sich bei der Übersetzung in Ada 95 zu zirkulären Abhängigkeiten verwandeln. So verfügt der Typ *Node* etwa über die Funktion *getOwnerDocument* welchselbige ein Objekt des Typs *Document* zurückgibt. Dieses wiederum erbt von *Node*.

Das Standardmapping sieht hierfür ein generisches Paket *Forward_Declaration* vor. Durch dieses werden für alle Typen auch entsprechende Vorwärtsdeklarationen instantiiert. So gibt es für den Typ *Document* einen passenden Typen *Document_Forward*. Da die Vorwärtsdeklarationen im Basispaket (hier *org_w3c_dom*) instantiiert werden, ist *Document_Forward* auch für *Node* sichtbar. Für das Konvertieren von *Document* in *Document_Forward* und umgekehrt stehen dann entsprechende Funktionen zur Verfügung (eine genauere Beschreibung dieses Vorgehens findet sich in [OMG01]).

In XML4Ada95 wurde das Problem der zirkulären Abhängigkeit anders gelöst. Der oben dargestellte Lösungsweg hätte zu sehr unschönem Code geführt, da ständig zwischen den Vorwärtsdeklarationen und den eigentlichen Typen hin- und herkonvertiert werden würde. Stattdessen wurden einfach alle Typen im Basispaket *org_w3c_dom* definiert statt in ihren eigenen Paketen. Dadurch sind sie auch von überall einsehbar und die Schnittstellen können wie gewohnt spezifiziert werden.

Dies setzt allerdings den Vererbungsmechanismus von Ada 95 außer Kraft. Wie wir jedoch im nächsten Abschnitt sehen werden, ist dies in unserem Fall sogar von Vorteil.

5.4. Alle enterbt!

Die IDL-Spezifikation der DOM sieht vor, dass zahlreiche Schnittstellen wie *Element*, *Document* und andere von der Schnittstelle *Node* erben. Doch das Vererben führt in unserem Fall gleich in zwei schwerwiegende Probleme.

Das erste Problem hängt mit dem Vererbungsmechanismus von Ada 95 zusammen, der bei dem Entwurf der IDL offensichtlich nicht beachtet wurde.

Gegeben ist zum Beispiel folgende Schnittstelle:

```
package org_w3c_dom.Node is
  type DOMNode is ...;
  ...
  function replaceChild
    (Self : DOMNode;
     newChild : DOMNode;
     oldChild : DOMNode)
    return DOMNode;
  ...
end org_w3c_dom.Node;
```

Document ist wie folgt spezifiziert:

```
package org_w3c_dom.Document is
  type DOMDocument is new DOMNode with ...;
  ...
end org_w3c_dom.Document;
```

Document verfügt nun nicht - wie vom W3C vorgesehen - über eine geerbte Funktion mit der folgenden Signatur:

```
function replaceChild
  (Self : DOMDocument;
   newChild : DOMNode;
   oldChild : DOMNode)
  return DOMNode;
```

Sondern über eine Funktion folgender Signatur:

```
function replaceChild
  (Self : DOMDocument;
   newChild : DOMDocument;
   oldChild : DOMDocument)
  return DOMDocument;
```

Eine vollkommen unsinnige Funktion, da *Document* per Standardspezifikation (siehe [ABC+98]) gar nicht über Kinder vom Typ *Document* verfügen kann.

Das zweite Problem, welches durch die Vererbung ausgelöst wird, wird in Abschnitt 4.4 dargestellt.

Beide Probleme werden dadurch gelöst, dass die Vererbung mit den Mitteln von Ada 95 schlicht nicht stattfindet. Stattdessen werden explizite und sichere Konvertierungsfunktionen angeboten, um *Node* in eines seiner Erben und wieder zurück zu verwandeln (`To_Node` und `From_Node`, jeweils definiert in der entsprechenden, sonst erbenden Schnittstelle).

5.5. Alles *Ref*

Eine kleine syntaktische Änderung betrifft die Benennung der Typen. Der Standard sieht eine Benennung des Pakets mit dem Namen des in IDL definierten *Interfaces* vor, der eigentliche Typ heißt immer *Ref*.

Dazu ein einfaches Beispiel. Gegeben ist der folgende IDL-Ausschnitt:

```
module org_w3c_dom {
  interface Node { ...
  };
};
```

Daraus würde folgende Spezifikation erstellt werden:

```
package org_w3c_dom.Node is
  type Ref is new ...;
};
```

Stattdessen wurde in XML4Ada95 eine alternative Namensgebung gewählt. Das Paket heißt immer wie das *Interface* (sollte das Paket mit den Buchstaben *DOM* anfangen, so werden diese weggelassen). Der eigentliche Typ heißt dann wie das Paket, bloß mit einem vorangestellten *DOM*. Oberes Beispiel ergibt sich also zu:

```
package org_w3c_dom.Node is
  type DOMNode is new ...;
};
```

Diese Umbenennung ist nicht willkürlich, sondern notwendige Folge aus der in Abschnitt 5.3 vorgestellten Entscheidung, alle Typen im Basispaket *org_w3c_dom* zu deklarieren. Dadurch können sie natürlich nicht alle *Ref* heißen, und entsprechende generische Namen mussten gefunden werden.

Als willkommenen Nebeneffekt lässt es die Schnittstelle und sie benutzende Anwendungen eleganter aussehen.

6. Noch eine Abkürzung: GXL

„Wer nicht weiß, zu welchem Hafen er segelt,
für den ist kein Wind von Vorteil“
(Seneca)

GXL (***G****raph* ***Ex****change* ***L****anguage*) wurde entworfen, um eine auf XML basierende Austauschsprache für Graphen zu standardisieren ([HWS00]).

Wie bereits in der Einführung dargelegt, ist das Benutzen von GXL durch das Bauhaus-Projekt die treibende Motivation zur Erstellung des Bindings gewesen. In diesem Kapitel wird die Vorgehensweise, mit der XML4Ada95 in das Bauhaus-Projekt eingebunden wurde, dargelegt. Dies dient zugleich als Anwendungsbeispiel für XML4Ada95.

6.1. Die Schnittstelle

Das Bauhaus-Projekt enthielt folgendes Paket, um RFGs als GXL-Dokument zu serialisieren und daraus einzulesen:

```
with RFGs;
with Bauhaus.Text_IO;

package RFGs.IO.GXL is

   procedure Put_All
     (The_RFG  : RFG;
      Filename : String);

   procedure Put -- (1)
     (The_RFG  : RFG;
      The_View : View;
      Filename : String);

   procedure Put -- (2)
     (The_RFG  : RFG;
      The_View : View;
      File     : Bauhaus.Text_IO.File_Type
                   := Bauhaus.Text_IO.Standard_Output);
```

```
   procedure Get
     (The_RFG    : RFG;
      Filename   : String;
      Viewname   : String;
      RFG_Exists : Boolean := True);

   procedure Get
     (The_RFG    : RFG;
      Filename   : String;
      RFG_Exists : Boolean := True);

end RFGs.IO.GXL;
```

Um die Veränderungen durch das neue XML-Paket möglichst klein zu halten, wurde die Schnittstelle weitgehend beibehalten. Lediglich die Prozedur *Put (2)* wurde entfernt und durch die folgenden zwei ersetzt:

```
   procedure Put -- (3)
     (The_RFG  : RFG;
      The_View : View;
      File     : Bauhaus.Text_IO.File_Type);

   procedure Put -- (4)
     (The_RFG  : RFG;
      The_View : View);
```

Hierbei verhält sich *Put (4)* genauso wie *Put (2)* mit dem Standardparameter als dritten Parameter. Warum diese Aufteilung gemacht wurde, wird im folgenden Kapitel erklärt.

Diese Veränderung verlangt keine Anpassungen am Code des restlichen Projekts, lediglich eine Neukompilierung ist notwendig (und eine Anpassung des *makefiles*).

6.2. Die Implementierung

Große Teile der vorliegenden Implementierung wurden weiter verwendet. Insbesondere zum Auslesen von Informationen aus dem RFG zur Ausgabe des GXL-Dokuments, sowie zum Erstellen von RFGs, Sichten, Knoten und Kanten aus dem eingelesenen GXL-Dokument wurde auf die bestehende Implementierung zurückgegriffen.

6.2.1. *Put*

Put_All und *Put (1)* öffneten die mit *filename* vorgegebene Datei und benutzten dann *Put (2)* zur eigentlichen Ausgabe. *Put (2)* schrieb dann das GXL-Dokument einfach Wort für Wort aus, ohne überhaupt ein XML-Paket zu verwenden. Dabei musste der Programmierer

ständig auch auf die syntaktischen Regeln von XML-Dokumenten achten, statt sich auf die eigentliche Aufgabe - das Konvertieren von RFGs - konzentrieren zu können.

Durch den Einsatz von XML4Ada95 war es möglich, die Konvertierung innerhalb einer neuen Prozedur, *Create_GXL_DOMTree* zu bündeln. Diese erstellt eine DOM und bevölkert sie mit den Informationen aus dem RFG. Die verschiedenen *Put*-Prozeduren haben so nur noch die Aufgabe, *Create_GXL_DOMTree* aufzurufen und dann die serialisierte DOM entsprechend auszugeben.

Das Serialisieren übernimmt dabei XML4Ada95, welches sich auch um die sonstigen syntaktischen Details von XML-Dokumenten kümmert. Die verschiedenen Möglichkeiten zur Serialisierung begründen auch die Änderung der Schnittstelle: die Schnittstelle *org_w3c_dom.Serializer* bietet die Funktionen *writeToString*, *writeToFile* und *writeToStdOut* an. *writeToFile* erhält als Parameter einen Dateinamen.

Während also *Put (1)* durch *writeToFile* und *Put (3)* durch *writeToStdOut* realisiert werden kann, gibt es keine Möglichkeit, die Datei, die man in *Put (4)* erhält an Xerces weiterzureichen und zu verwenden. Stattdessen wird das GXL-Dokument in einem String über *writeToString* serialisiert und derselbe dann in die offene Datei geschrieben.

Theoretisch sollte diese Möglichkeit keine Schwierigkeiten bereiten, in der Praxis jedoch sind Probleme vorstellbar. Bei sehr großen Dokumenten, die ohnehin viel Speicher benötigen, einmal als RFG, einmal als GXL, käme noch ein entsprechend langer String hinzu, was insgesamt einen sehr hohen, kurzfristigen Speicherbedarf bedeutet. Da also diese Variante möglicherweise zu Problemen führt, wurde für die Ausgabe zum *Standard Output* eine eigene, stabilere Prozedur geschrieben, die *writeToStdOut* verwendet.

6.2.2. *Get*

Die wirklichen Vorteile spielt das Verwenden von XML4Ada95 nicht beim Schreiben von XML-Dokumenten, sondern bei deren Einlesen. Der bislang verwendete *XMLParser*, ein Teil von *GtkAda* ([BBCS]), ist eine sehr einfache Lösung. Sie verfügt über ein ungenügendes Fehlerverhalten und keine Möglichkeiten zur Validierung.

Bei der Implementierung der *Get*-Prozeduren wurde der eingesetzte *XMLParser* durch XML4Ada95 ersetzt. Dies vereinfachte die Implementierung zwar nur geringfügig, bietet aber ohne weiteres Zutun folgende Vorteile:

- die eingelesenen GXL-Dokumente werden validiert. Wenn die Validierung nicht erfolgreich ist, wird das gemeldet (mitsamt den Fehlern), und XML4Ada95 versucht fortzusetzen. Das Ergebnis ist in dem Fall nicht definiert.

- bei einem nicht wohlgeformten oder validierbaren Dokument stürzt der bisher eingesetzte *XMLParser* ohne weitere Hinweise ab. XML4Ada95 meldet die möglichen Fehler und versucht fortzusetzen (auch hier ist ein Absturz möglich, aber immerhin gab es zuvor schon die entsprechenden Fehlermeldungen)

Dies vereinfacht die Wartung des Programms, da man im Falle eines Fehlers über dessen Ursache besser informiert wird.

6.3. Syntaktische Veränderung

Bei der Arbeit an der GXL-Schnittstelle wurde festgestellt, dass die bisherige Implementierung kein dem Standard genügendes GXL erstellte.

Folgender Ausschnitt soll das illustrieren. Zunächst das GXL-Dokument, wie es bisher erstellt wurde.

```
...
<gxl>
  <graph ...>
    <node id="N1001" type="TYPE">
      <attr ...> ... </attr>
    </node>  ...
  </graph>
</gxl>
```

Die entsprechenden Stellen in der DTD sehen wie folgt aus:

```
...
<!ATTLIST type
xlink:type (simple) #FIXED "simple"
xlink:href CDATA #REQUIRED
> ...
<!ELEMENT node (type? , attr*, graph* %node-extension;) >
<!ATTLIST node
id ID #REQUIRED
%node-attr-extension;
> ...
```

Das Element *node* verfügt also nicht über ein Attribut *type*, sondern über ein Element gleichen Namens. Diese syntaktische Änderung wurde durchgeführt, und das oben abgedruckte GXL-Dokument sieht jetzt wie folgt aus:

```
...
<gxl>
  <graph ...>
    <node id="N1001">
      <type xlink:href="TYPE"/>
      <attr ...> ... </attr>
    </node> ...
  </graph>
</gxl>
```

Die selbe Änderung wurde für das Element *edge* durchgeführt.

Die mit der neuen Implementierung erstellten GXL-Dokumente validieren dadurch fehlerlos gegen die *gxl-1.0.dtd* ([HWS00]).

7. Die Alternative: SAX

"The absence of alternatives clears the mind marvelously."
(Henry A. Kissinger)

"And now to something completely different."
(John Cleese)

Wie bereits in Abschnitt 2.4 dargelegt, existiert neben der DOM noch eine zweite, wichtige Schnittstelle für die Arbeit mit XML-Dateien: die **SAX - *S**imple **A**PI for **X**ML*. In diesem Kapitel wird eine kurze Einführung in die SAX gegeben, daraufhin werden verschiedene Möglichkeiten dargelegt, wie die SAX für Ada 95 benutzt werden kann.

7.1. Wie funktioniert die SAX?

Ebenso wie die DOM wird die SAX von vielen XML-Paketen unterstützt, im Gegensatz zu jener ist hier jedoch die Spezifikation nicht derart formal gegeben. Die SAX verfolgt folgende Vorgehensweise: das XML-Dokument wird als ein Datenstrom durch den Parser geschleust. Dabei erkennt der Parser die syntaktischen Konstrukte von XML, wie etwa öffnende und schließende *Element-tags, Attribute* oder *Entities*. Der Benutzer der Schnittstelle muss bereits vor dem Parsen des Elements seine Funktionen eingehängt haben, welche die entsprechenden Konstrukte und ihren Inhalt bearbeiten. Diese eingehängten Funktionen werden dann entsprechend aufgerufen.

Über Status und Kontext Buch zu halten ist Aufgabe des Benutzers. Er kann nicht - wenn er sich die Daten nicht selber gemerkt hat - auf bereits geparste, und erst recht nicht auf noch nicht bearbeitete Teile des Dokuments zugreifen.

Daraus erwachsen folgende Vor- und Nachteile dieses Verfahrens:

Eine schnelle und speichereffiziente Implementierung ist einfacher zu realisieren. Sie ist jedoch nicht automatisch gegeben: die Standards beschreiben die Schnittstelle, nicht die Implementierung. Viele Pakete basieren sowohl für die DOM wie auch für die SAX auf dem selben Parser. Des Weiteren basieren zwar die meisten DOM-Schnittstellen darauf, dass stets der vollständige Baum im Speicher gehalten wird, notwendig ist das jedoch nicht: *PullDOM* ([vRFLD]) ist eine Python-Implementierung, die erst dann die entsprechenden Teile eines XML-Dokuments einliest, wenn der Benutzer darauf zugreift. *StAX* (*Streaming API for XML*, [Fry]) für Java ebenso wie *XML::Twig* ([Pro]) für Perl gehen im Prinzip ebenfalls einen ähnlichen Weg, und versuchen damit die Geschwindigkeit von SAX mit der Einfachheit der DOM zu vereinen.

Potenziell können größere Dateien bearbeitet werden. *Xerces* kann über die DOM-Schnittstelle laut [Chi03] mit 64 MB großen Dateien umgehen - wird die SAX-Schnittstelle verwendet, kommt das Paket auch mit bis 512 MB großen Dateien zurecht. Bei *libxml* liegen die Grenzen bei 64 MB unter der DOM, verwendet man die SAX können die Dateien bis zu 256 MB groß werden. Dies hängt natürlich direkt mit dem vorgenannten Punkt zusammen.

Die Syntax des Dokuments sollte der Erfordernissen der SAX gerecht entworfen werden. In der DOM ist die Reihenfolge bestimmter Knoten von geringer Relevanz, da ja wahlfreier Zugriff besteht. Für die SAX hingegen kann die Reihenfolge der Daten bedeutend die Komplexität des Bearbeitens des Dokuments beeinflussen. Ob etwa bestimmte Daten, die einen Kontext für referierte Daten setzen und damit deren Bearbeitung beeinflussen, im Dokument tatsächlich vor den referierten Daten gegeben sind oder nicht, wird das Schreiben der Anwendung, welche die SAX verwendet, enorm beeinflussen.

7.2. SAX und Ada 95

SAX ist etwas weniger streng reglementiert als die DOM: *"Outside of Java, SAX is whatever programmers in that language decide it should be."* ([Bro]). Daraus ergibt sich, dass jede Schnittstelle gültig ist, die von sich behauptet, eine Implementierung der SAX zu sein und die dann als solche innerhalb der Gemeinschaft der Programmierer Akzeptanz findet.

Emmanuel Briots Paket *XmlAda* ([Bri]) bietet eine Implementierung der SAX an. Dies ist die am weitesten entwickelte, frei verfügbare native Realisierung der SAX in Ada 95. Das Paket ist unter der GMGPL veröffentlicht und somit für jeden frei verwendbar. Zu überprüfen, ob diese Implementierung adäquat ist, ist nicht Teil dieser Diplomarbeit.

Im folgenden wird untersucht, ob sich die SAX-Schnittstelle ähnlich der DOM für Ada 95 umsetzen lässt.

Auch hier bietet sich *Xerces* als zugrunde liegende Implementierung an. Dabei spielen zwar auch die Gründe aus Kapitel 3 eine Rolle, doch sucht man nach XML-Paketen, welche die SAX implementieren, ohne die Bedingung zu stellen, dass sie auch die DOM implementieren müssen, so kann man aus einer wesentlich größeren Menge auswählen. Für Xerces würde sprechen, dass es bereits für das Binding der DOM verwendet wird, und dass es eine vollständige Implementierung der SAX anbietet.

Die Realisierung der SAX durch Xerces beruht vor allem auf zwei Klassen, *SAXParser* und *DocumentHandler*. Der Benutzer der SAX erstellt dabei eine von *DocumentHandler* abgeleitete Klasse, in welcher die Funktionen zur Behandlung der entsprechenden XML-Konstrukte überschrieben werden. Ein Beispiel: möchte der Benutzer herausfinden, wie viele Elemente das Dokument enthält, überschreibt er die Funktion *startElement*, worin er eine Variable mitzählen lässt. Ein Objekt dieser abgeleiteten Klasse reicht der Benutzer dann an *SAXParser* und startet das Parsen. Im Beispiel würde dann jedes Mal, wenn im Dokument ein öffnendes *Elementtag* vom Parser erkannt wird, die überschriebene Funktion aufgerufen werden. Nach dem Parsen kann der Benutzer dann die Variable abfragen.

Die eben beschriebene Vorgehensweise kommt so auch in der DOM vor: der *DOMParser* wird auf diese Art mit einem *ErrorHandler* konfiguriert. Wie das im Binding realisiert wurde, kann in Kapitel 4.7 nachgelesen werden. Auf diese Art und Weise sollte es jedoch

ohne größere Schwierigkeiten möglich sein, die SAX an Ada 95 zu binden.

Durchaus größere Schwierigkeiten dürften hier jedoch Akzeptanz und Schnittstellendesign bereiten.

Damit ein SAX-Binding überhaupt akzeptiert werden kann, muss man zunächst erfolgreich begründen, warum man die native Lösung, *XmlAda*, nicht verwenden kann oder will. Ungeachtet der Tatsache, dass das Paket im Rahmen der Diplomarbeit keine nähere Untersuchung erfahren hat und deswegen hier auch keine Aussagen über Konformität zu SAX oder zum XML-Standard, über die Maximalgröße von XML-Dokumenten oder über die Geschwindigkeit getroffen werden können, kann dennoch festgehalten werden, dass *XmlAda* weder gegen DTDs noch gegen Schemata zu validieren vermag. Validierung kann aber je nach Aufgabe entscheidend sein, womit sich das Verwenden von *XmlAda* verbieten würde.

Das Schnittstellendesign jedoch hat im Vergleich zur DOM sogar höhere Anforderungen: während die DOM sprachunabhängig in IDLs definiert ist, und damit bei der Benennung von Funktionen und Schnittstellen keinen Spielraum lässt, liegt die SAX vor allem als Java-Implementierung vor. Hierbei gilt es, die Java-Implementierung zu analysieren, ihre Prinzipien von den sprachabhängigen Elementen zu trennen und dann dieselben unter Berücksichtigung von Idiomen und Gebräuchen in einer Schnittstelle für Ada 95 zu formulieren.

Gleichzeitig muss die dem Binding zugrunde liegende Implementierung beachtet werden. Im Falle von *Xerces* liegt diese in C++ vor: bei dem Entwurf der Ada 95-Schnittstelle sollte diese unbedingt berücksichtigt werden, damit hier eine leichte Wartbarkeit und ausreichend Effizienz gewährleistet wird. Am Beispiel *Arabica* (Seite 18) kann man sehen, wie zu ehrgeizige Prinzipien dramatisch der Effizienz und der Nutzbarkeit schaden können.

Schließlich sollte auch hier in das Schnittstellendesign einfließen, dass das Binding nur eine vorläufige Lösung ist. Das bedeutet, dass auch hier zu erwarten steht, dass in Zukunft eine geeignete, native Implementierung zur Verfügung stehen wird (die etwa aus *XmlAda* entstehen könnte, oder aber zur Standardausstattung einer kommenden Ada-Version gehört, siehe [Taf02]). Um diesen zu erwartenden Wechsel dann so günstig wie möglich zu ermöglichen, sollte bereits jetzt die entsprechende Schnittstelle für SAX so gestaltet sein, dass nur minimale bis keine Änderungen an den die Schnittstelle verwendenden Programmen nötig werden.

Zusammengefasst kann festgehalten werden, dass die technische Umsetzung des SAX-Bindings einfach, der Entwurf einer akzeptablen Schnittstelle in Ada 95 jedoch keine triviale Aufgabe sein würde.

8. Fazit und Ausblick

Frodo: "Come on Sam... Remember what Bilbo used to say:
It's a dangerous business, Frodo, going out your door.
You step onto the road, and if you don't keep your feet,
there's no knowing where you might be swept off to..."
(Lord of the Rings: The Fellowship of the Ring)

Wäre die Wahl bezüglich der zugrunde liegenden Implementierung nicht auf *Xerces* sondern auf *libxml* gefallen, wären viele interessante Probleme, mit denen sich diese Diplomarbeit auseinandersetzt, gar nicht aufgetaucht. Doch diese Gelegenheit erlaubte eine Betrachtung der Schnittstelle zwischen Ada 95 und C++. Dadurch, dass diese Erfahrungen hier festgehalten sind, kann vielleicht in Zukunft das Verbinden beider Sprachen einfacher gelingen.

Dies war jedoch nur eine Begleiterscheinung der eigentlichen Aufgabe.

Das Hauptergebnis dieser Diplomarbeit ist XML4Ada95. Dadurch haben Programmierer nun auch von Ada 95 aus Zugriff auf ein mächtiges und flexibles XML-Paket. Dies ermöglicht nicht nur, mit einer ständig wachsenden Zahl von Standards umzugehen, sondern auch triviale Daten wie Konfigurationsparameter eines Programms syntaxsicher abzuspeichern und validieren zu können. Ein solches Paket erlaubt dem Benutzer, beliebige Daten in einer selber definierten Syntax zu speichern, ohne dass er einen Parser und einen Validierer für diese Syntax schreiben oder generieren muss.

Auf `http://www.nodix.de/xml4ada95` wird mit Einreichen der Diplomarbeit das Paket zur freien Verfügung stehen. Es enthält, neben der Dokumentation der Schnittstelle, auch Hinweise zur Installation und Beispiele, um den Einstieg zu vereinfachen. Mit Sicherheit wird es in den ersten Wochen und Monaten kritische Anmerkungen und Fehlermeldungen zu dem Paket geben. Diese werden entsprechend eingearbeitet.

Sollte XML4Ada95 größere Verbreitung finden, würde damit der Beweis für den Bedarf nach einem vollständigen XML-Paket erbracht worden sein. Der Ruf nach einer nativen Implementierung wird dabei immer lauter werden, da ein Binding wie XML4Ada95 immer mit größeren Problemen einhergeht als eine native Implementierung - sowohl auf technischer, aber auch persönlicher, subjektiver Ebene seitens der Programmierer.

Dieses native XML-Paket könnte auf verschiedener Weise entstehen.

Ein bereits verwendetes, internes Paket einer Firma oder eines Instituts wird veröffentlicht und weiter ausgebaut. Bei der Bedeutung, die XML heute schon hat, ist es gut möglich, dass entsprechende Pakete tatsächlich schon existieren.

Es könnte auch die Entwicklung von *XmlAda* durch das steigende Interesse weitere Unterstützung finden, entweder durch die Programmierergemeinde oder durch universitäre oder sonstige institutionelle Förderung. Gerade die starke Aufteilung der DOM in Module und *Level* erlaubt es, vergleichsweise kleine Pakete zu schnüren und diese dann auch Studenten im Rahmen von Softwarepraktika, Studien- oder Diplomarbeiten als Aufgabe anzubieten.

Schließlich besteht noch die bereits erwähnte Möglichkeit, dass eine zukünftige Version von Ada von vornherein mit XML-Fähigkeiten ausgestattet sein wird. Eine zu dem Zeitpunkt starke Verbreitung bereits bestehender XML-Schnittstellen würde, um die Interessen der Anwender zu wahren, zu einem Entwurf der dann im Standard enthaltenen Schnittstelle führen, die einen günstigen Wechsel erlaubt.

Egal wie es weitergeht, wir sehen, dass die weite Verbreitung einer Schnittstelle nur zum Vorteil der Benutzer dieser Schnittstelle gereichen kann. XML4Ada95 frei zur Verfügung zu stellen will dieser Entwicklung Vorschub leisten.

Glossar

Die vorliegende Diplomarbeit wimmelt geradezu von Abkürzungen und technischen Begriffen. Die folgende Aufstellung bietet nicht Definitionen oder Erklärungen, sondern lediglich Erinnerungsstützen. Schwierigkeiten beim Verständnis eines der Begriffe sollten mit Hilfe von Quellen außerhalb dieses Glossars behoben werden, damit die Diplomarbeit dem Leser ihren vollen Nutzen entfalten kann.

API - ***A**pplication **P**rogrammers **I**nterface*, eine Schnittstelle für Programmierer von Anwendungen. Eine einfache Möglichkeit, Funktionalität, die in anderen Modulen zur Verfügung gestellt wird, zu verwenden. Beispiele sind GUI-Toolkits, standardisierte Bibliotheken wie OpenGL oder eben XML-Pakete

Dokument - Ein Text, dessen Syntax in einer XML- oder SGML-Anwendung definiert ist

DOM - ***D**ocument **O**bject **M**odel*, Darstellung eines XML-Dokuments als Baum sowie die Schnittstelle, auf diesen Baum zuzugreifen. Die Schnittstelle ist in *Level* aufgeteilt

DTD - ***D**ocument **T**ype **D**efinition*, die Definition der Syntax einer XML- oder SGML-Anwendung

GML - ***G**eneralized **M**arkup **L**anguage*, bei IBM entwickelter Vorläufer von SGML

GXL - ***G**raph E**x**change **L**anguage*, auf XML basierende Sprache zum Austausch von Graphen

HTML - ***H**yper**t**ext **M**arkup **L**anguage*, eine SGML-Anwendung, dient im Allgemeinem zum Erstellen von Websiten

Level - Die DOM-Schnittstelle ist in mehrere Level aufgeteilt, wobei Level 1 einfachste Kernfähigkeiten zur Verfügung stellt, auf der die folgenden Level aufbauen. Level 3 steht kurz vor der Verabschiedung, weitere Level sind vorgesehen. Ab Level 2 sind die Level in Module aufgeteilt

Metasprache - Sprache zur Definition von Sprachen

Modul - Die Level der DOM-Schnittstelle sind ab Level 2 in Module aufgeteilt, die dem Anwender bestimmte Möglichkeiten zur Verfügung stellen. Abschnitt 2.4 listet die gegenwärtigen Module auf

SAX - ***S**imple **A**PI for **X**ML*, eine Programmierschnittstelle zur ereignisbasierten Verarbeitung von XML-Dokumenten, vergleiche Kapitel 7

Schema - Definition der Syntax einer XML-Anwendung, mächtigerer Nachfolger von DTD, selbst eine XML-Anwendung

SGML - ***S**tandard **G**eneralized **M**arkup **L**anguage*, Obermenge und praktisch Vorläufer von XML, eine allgemeine Metasprache zur Definition von Sprachen

SGML-Anwendung - Eine Sprache, die auf SGML basiert, also mit einer DTD definiert ist. Jede XML-Anwendung ist auch eine SGML-Anwendung

SGML-Profil - Eine Untermenge von SGML. XML ist ein SGML-Profil

validieren - Ein Dokument gegen die Syntaxdefinition, wie sie in einer DTD oder einem Schema festgehalten ist, prüfen. Jedes validierende Dokument ist wohlgeformt

W3C - Das ***W**orld **W**ide **W**eb **C**onsortium*, unabhängige Gruppe, welche die Techniken, auf denen das World Wide Web basiert, spezifiziert. Ihre Website ist `http://www.w3c.org`

wohlgeformt - Ein Dokument, welches die allgemeinen syntaktischen Regeln von XML erfüllt

XML - ***Ex**tensible **M**arkup **L**anguage*, erweiterbare Markierungssprache. Allgemeine, vom W3C definierte, von SGML abgeleitete Metasprache zur Definition von Markierungssprachen

XML-Anwendung - Eine Sprache, die auf XML basiert, also mit einer DTD oder mit einem Schema definiert ist

XML-Parser - Ein Paket, welches XML-Dokumente lesen kann und über eine API zur Verfügung stellt

Literaturverzeichnis

[ABC+98] APPARAO, VIDUR, STEVE BYRNE, MIKE CHAMPION, SCOTT ISAACS, IAN JACOBS, ARNAUD LE HORS, GAVIN NICOL, JONATHAN ROBIE, ROBERT SUTOR, CHRIS WILSON und LAUREN WOOD: *Document Object Model (DOM) Level 1*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/REC-DOM-Level-1/, 1998.

[ABC+01] ADLER, SHARON, ANDERS BERGLUND, JEFF CARUSO, STEPHEN DEACH, TONY GRAHAM, PAUL GROSSO, EDUARDO GUTENTAG, R. ALEXANDER MILOWSKI, SCOTT PARNELL, JEREMY RICHMAN und STEVE ZILLES: *Extensible Stylesheet Language (XSL) Version 1.0*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/xhtml1/, 2001.

[ACK+00] APPARAO, VIDUR, MIKE CHAMPION, JOE KESSELMAN, JONATHAN ROBIE, PETER SHARPE und LAUREN WOOD: *Document Object Model (DOM) Level 2 Traversal and Range Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-Traversal-Range/, 2000.

[Apa] APACHE: *Xerces C++ Parser*. http://xml.apache.org/xerces-c/index.html.

[Apa99] APACHE: *The Apache Software License, Version 1.1*. http://xml.apache.org/LICENSE, 1999.

[ARM95] *ADA-ARM (ISO/IEC 8652:1995/COR.1:2000)*. http://www.adaic.org/standards/95aarm/html/AA-TOC.html, 1995.

[AWH00] APPARAO, VIDUR, CHRIS WILSON und PHILIPPE LE HEGARET: *Document Object Model (DOM) Level 2 Style Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-Style/, 2000.

[BBCS] BRIOT, EMMANUEL, JOEL BROBECKER, ARNAUD CHARLET und NICOLAS SETTON: *GtkAda home page*. http://libre.act-europe.fr/GtkAda/.

[BHL99] BRAY, TIM, DAVE HOLLANDER und ANDREW LAYMAN: *Namespaces in XML*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/REC-xml-names/, 1999.

[BM01] BIRON, PAUL V. und ASHOK MALHOTRA: *XML Schema Part 2: Datatypes*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/xmlschema-2/, 2001.

[BPSM98] BRAY, TIM, JEAN PAOLI und C.M. SPERBERG-MCQUEEN: *Extensible Markup Language (XML) 1.0*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/REC-xml, 1998.

[BPSMM00] BRAY, TIM, JEAN PAOLI, C.M. SPERBERG-MCQUEEN und EVE MALER: *Extensible Markup Language (XML) 1.0 (Second Edition)*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/1998/REC-xml-19980210, 2000.

[Bri] BRIOT, EMMANUEL: *XML/Ada: a full XML suite*. http://libre.act-europe.fr/xmlada/.

[Bro] BROWNELL, DAVID: *SAX official Website*. http://www.saxproject.org/.

[BSD98] *The BSD License*. http://www.opensource.org/licenses/bsd-license.php, 1998.

[Cas] CASARINI, PAOLO: *Gnome DOM Engine - libgdome, a.k.a. gdome2*. http://gdome2.cs.unibo.it/.

[CD99] CLARK, JAMES und STEVE DEROSE: *XML Path Language (XPath) Version 1.0*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/xpath, 1999.

[CDP] COOPER, CLARK, FRED DRAKE und PAUL PRESCOD: *The Expat XML Parser*. http://expat.sourceforge.net/.

[Chi03] CHILINGARYAN, SUREN A.: *XML Benchmark*. http://xmlbench.sourceforge.net/, 2003.

[CKR03] CHANG, BEN, JOE KESSELMAN und REZAUR RAHMAN: *Document Object Model (DOM) Level 3 Validation Specification Version 1.0*. Candidate Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-3-Val/, 2003.

[CP01] CASARINI, PAOLO und LUCA PADOVANI: *The Gnome DOM Engine*. Technischer Bericht, Department of Computer Science, Mura Anteo Zamboni 7, 40127 Bologna, Italy, http://gdome2.cs.unibo.it/gdome_xmlextreme/, 2001.

[CPX] *CenterPoint XML*. http://www.cpointc.com/XML/.

[CSH02] CARLISLE, MARTIN C., RICKY SWARD und JEFF HUMPHRIES: *Weaving Ada 95 into the .Net Environment*. SigAda 2002 Paper, Department of Computer Science, United States Air Force Academy, http://www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html, 2002.

[Den03] *Internet-Statistiken der Denic Datenbank*. `http://www.denic.de/DENICdb/stats/index.html`, 2003.

[Dot] *DotGNU Project - GNU Freedom for the Internet*. `http://www.dotgnu.org/`.

[EKP+99] EISENBARTH, THOMAS, RAINER KOSCHKE, ERHARD PLÖDEREDER, JEAN-FRANCOIS GIRARD und MARTIN WÜRTHNER: *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen, Workshop Software-Reengineering, Bad Honnef*. Technischer Bericht 7-99, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, `http://www.bauhaus-stuttgart.de/bauhaus/papers/index.html`, 1999.

[ER02] EASTLAKE, DONALD und JOSEPH REAGLE: *XML Encryption Syntax and Processing*. Recommendation, W3C (World Wide Web Consortium), `http://www.w3.org/TR/xmlenc-core/`, 2002.

[ERS02] EASTLAKE, DONALD, JOSEPH REAGLE und DAVID SOLO: *XML-Signature Syntax and Processing*. Recommendation, W3C (World Wide Web Consortium), `http://www.w3.org/TR/smldsig-core/`, 2002.

[Fal01] FALLSIDE, DAVID C.: *XML Schema Part 0: Primer*. Recommendation, W3C (World Wide Web Consortium), `http://www.w3.org/TR/xmlschema-0/`, 2001.

[Fer01] FERRAIOLO, JON: *Scalable Vector Graphics (SVG) 1.0 Specification*. Recommendation, W3C (World Wide Web Consortium), `http://www.w3.org/TR/SVG/`, 2001.

[FRS+03] FLORESCU, DANIELA, JONATHAN ROBIE, JEROME SIMEON, SCOTT BOAG, DON CHAMBERLIN und MARY F. FERNANDEZ: *XQuery 1.0: An XML Query Language*. Working Draft, W3C (World Wide Web Consortium), `http://www.w3.org/TR/xquery/`, 2003.

[Fry] FRY, CHRISTOPHER: *Java Community Process - Java Specification Requests 173: Streaming API for XML*. `http://www.jcp.org/en/jsr/detail?id=173`.

[Gar] GARSHOL, LARS MARIUS: *Ada Reference Manual*. `http://www.ada-auth.org/~acats/arm.html`.

[gcc] *GCC documentation*. `http://gcc.gnu.org/onlinedocs/`.

[Gol86] GOLDFARB, CHARLES: *ISO 8879:1986 TC2. Information technology – Document Description and Processing Languages*. Standard ISO 8879:1986, ISO (International Organization for Standardization), Genf, 1986.

[grm] *GNAT Reference Manual*. http://gcc.gnu.org/onlinedocs/gnat_rm/.

[HB03] HALLAM-BAKER, PHILLIP: *XML Key Management Specification (XKMS) Version 2.0*. Working Draft, W3C (World Wide Web Consortium), http://www.w3.org/TR/xkms2/, 2003.

[HC00] HORS, ARNAU LE und LAURENCE CABLE: *Document Object Model (DOM) Level 2 Views Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-Views/, 2000.

[HHW+00] HORS, ARNAUD LE, PHILIPPE LE HEGARET, LAUREN WOOD, GAVIN NICOL, JONATHAN ROBIE, MIKE CHAMPION und STEVE BYRNE: *Document Object Model (DOM) Level 2 Core Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-Core/, 2000.

[HHW+03] HORS, ARNAUD LE, PHILIPPE LE HEGARET, LAUREN WOOD, GAVIN NICOL, JONATHAN ROBIE, MIKE CHAMPION und STEVE BYRNE: *Document Object Model (DOM) Level 3 Core Specification Version 1.0*. Working Draft, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-3-Core/, 2003.

[Hig] HIGGINS, JEZ: *Arabica Homepage*. http://www.jezuk.co.uk/cgi-bin/view/arabica.

[HP03] HÉGARET, PHILIPPE LE und TOM PIXLEY: *Document Object Model (DOM) Level 3 Events Specification Version 1.0*. Working Draft, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-3-Events/, 2003.

[HPQ] HUGUES, JEROME, LAURENT PAUTET und THOMAS QUINOT: *PolyORB Generic Middleware*. http://libre.act-europe.fr/polyorb/.

[HWS00] HOLT, RICHARD C., ANDREAS WINTER und ANDY SCHÜRR: *GXL: Towards a Standard Exchange Format*. Technischer Bericht 1–2000, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, http://www.gupro.de/GXL/Publications/publications.html, 2000.

[IBM99] IBM: *alphaworks : XML for C++*. http://www.alphaworks.ibm.com/tech/xml4c, 1999.

[IM99] ION, PATRICK und ROBERT MINER: *Mathematical Markup Language (MathML) 1.01 Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/REC-MathML/, 1999.

[Mica] MICROSOFT: *Microsoft .NET*. http://www.microsoft.com/net/.

[Micb] MICROSOFT: *Microsoft XML Core Services (MSXML) 4.0*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/htm/sdk_intro_6g53.asp.

[MIT] *The MIT License*. http://www.opensource.org/licenses/mit-license.php.

[OMG01] *Ada Language Mapping v1.2*. Specification, OMG (Object Management Group), http://www.omg.org/technology/documents/formal/ada_language_mapping.htm, 2001.

[Oraa] ORACLE: *Oracle 9i XML Developer's Kit for C++*. http://otn.oracle.com/tech/xml/xdk_cpp/index.html.

[Orab] ORACLE: *Oracle Technology Network (OTN) Development License Agreement*. http://otn.oracle.com/docs/tech/xml/xdk/doc_library/Beta/license.html.

[Pem00] PEMBERTON, STEVEN: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/xhtml1/, 2000.

[Pix00] PIXLEY, TOM: *Document Object Model (DOM) Level 2 Events Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-Events/, 2000.

[Pro] PRONOVICI, KENNETH J.: *libxml-twig-perl*. http://packages.qa.debian.org/libx/libxml-twig-perl.html.

[SH03] STENBACK, JOHNNY und ANDY HENINGER: *Document Object Model (DOM) Level 3 Load and Save Specification Version 1.0*. Working Draft, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-3-LS/, 2003.

[SHH03] STENBACK, JOHNNY, PHILIPPE LE HEGARET und ARNAUD LE HORS: *Document Object Model (DOM) Level 2 HTML Specification*. Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-2-HTML/, 2003.

[Sta91] STALLMAN, RICHARD: *GNU General Public License*. Lizenz, FSF (Free Software Foundation), http://www.fsf.org/licenses/gpl.html, 1991.

[Taf02] TAFT, S. TUCKER: *Ada 200Y – What and Why*. Vortrag, ISO (International Organization for Standardization), http://anubis.dkuug.dk/JTC1/SC22/WG9/n420.pdf, 2002.

[TBMM01] THOMPSON, HENRY S., DAVID BEECH, MURRAY MALONEY und NOAH MENDELSOHN: *XML Schema Part 1: Structures*. Recommendation, W3C

(World Wide Web Consortium), http://www.w3.org/TR/xmlschema-1/, 2001.

[Vei] VEILLARD, DANIEL: *The XML C parser and toolkit of Gnome - libxml*. http://www.xmlsoft.org/.

[vRFLD] ROSSUM, GUIDO VAN und JR. FRED L. DRAKE: *Python Library Reference*. http://www.python.org/doc/current/lib/.

[Whi03] WHITMER, RAY: *Document Object Model (DOM) Level 3 XPath Specification Version 1.0*. Candidate Recommendation, W3C (World Wide Web Consortium), http://www.w3.org/TR/DOM-Level-3-XPath/, 2003.

[Win02] WINER, DAVE: *RSS 2.0 Specification*. Technischer Bericht, Berkman Center for Internet and Society at Harvard Law School, http://www.w3.org/TR/REC-xml-names/, 2002.

[Xim] XIMIAN: *mono*. http://www.go-mono.com/.