Konrad Gräfe

Konzeption und prototypische Implementierung eines Programms zur Interpretierung von Merkmaldateien nach DIN V 4001 und Speicherung der Daten in einem leistungsfähigeren Format

BACHELORARBEIT

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Fakultät Mathematik-Physik-Informatik

Mittweida, 2009

Konrad Gräfe

Konzeption und prototypische Implementierung eines Programms zur Interpretierung von Merkmaldateien nach DIN V 4001 und Speicherung der Daten in einem leistungsfähigeren Format

eingereicht als

BACHELORARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)

UNIVERSITY OF APPLIED SCIENCES

Fakultät Mathematik-Physik-Informatik

Mittweida, Oktober 2009

Erstprüfer:	Prof. Dr. rer. nat. Günter Werner
Zweitprüfer:	Dipl.-Inf. Chris Hübsch

Bibliografische Beschreibung:

Konrad Gräfe:

Konzeption und prototypische Implementierung eines Programms zur Interpretierung von Merkmaldateien nach DIN V 4001 und Speicherung der Daten in einem leistungsfähigeren Format. - 2009. - 61 Seiten. Mittweida, Hochschule Mittweida (FH) - University of Applied Sciences, Fakultät Mathematik-Physik-Informatik, Bachelorarbeit, 2009

Referat:

Die vorliegende Arbeit beschäftigt sich mit der Interpretierung von Merkmaldateien nach DIN V 4001 und der Speicherung der Daten in einem leistungsfähigeren Format. Dieses Format ist zu evaluieren und dessen Wahl zu begründen. Weiterhin wird geprüft, ob das Konvertieren der Merkmaldateien möglich ist und ein Weg zur Konvertierung aufgezeigt. Die besondere Schwierigkeit besteht darin, dass in den Merkmaldaten Algorithmen in der Programmiersprache *FORTRAN77* eingepflegt sind. Es wird ein Weg auf Basis von *ANTLR*, einem Framework, das u.a. auf die Übersetzung strukturierter Daten spezialisiert ist, entwickelt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

ANSI	American National Standards Institute
ANTLR	ANother Tool for Language Recognition
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
BSF	Bean Scripting Framework
BSV	Bravo Schraub Verbindung(en)
CAD	Computer Aided Design
CSM	Component Supply Management
DIN	Deutsches Institut für Normung
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
ISO	International Standards Organisation
RCP	Rich Client Platform
RegExp	Regulärer Ausdruck (regular expression)
Tcl	Tool command language
XML	Extensible Markup Language

Kapitel 1

Einleitung

1.1 Die *ARC Solutions GmbH*

Die *ARC Solutions GmbH* ist ein inhabergeführtes Software-Unternehmen. Das Unternehmen beschäftigt zehn Festangestellte und eine wechselnde Zahl an Praktikanten. Neben der Installation und Wartung verschiedener Systeme im Bereich Computer Aided Design (CAD) bietet die Firma mit *Remarc* ein Werkzeug zur Generierung und Verwaltung von Bauteilen für verschiedene CAD-Systeme. Unter anderem bietet es die Möglichkeit, Normteile nach Vorschriften des Deutschen Institutes für Normung (DIN) zu erzeugen.

1.2 Motivation und Ziele

Remarc bezieht die Daten zur Generierung der Normteile aus Dateien der DIN V 4001, welche von der DIN gepflegt werden und nach der Vorschrift (*CAD-Normteiledatei nach DIN* 1990) aufgebaut sind. Dabei greift es auf ein etwa 15 Jahre altes Programm namens *Bravo Schraub Verbindung(en) (BSV)* zurück, welches zustandsbehaftet arbeitet. Das bedeutet, dass beim Anlegen einer Instanz jede Änderung auf dem alten Zustand aufbaut. Durch die Umstellung von *Remarc* auf die *Eclipse* Rich Client Platform (RCP) ist dies nicht mehr möglich, der jeweils letzte Zustand muss bei jeder Änderung neu aufgebaut werden. Dies lässt den ganzen Vorgang sehr träge erscheinen, was sich in zwei Situationen äußert:

- Wird ein Datensatz von Hand ausgefüllt, wird auf Basis von vorgegebenen Datensätzen aus den DIN V 4001-Dateien eine Liste möglicher Werte vorgeschlagen. Außerdem wird nach jeder Eingabe überprüft, ob ein gültiger Wert eingegeben wurde. Beides führt dazu, dass der Arbeitsfluss nach jeder Eingabe für vier bis fünf Sekunden unterbrochen wird.

- Die Wartezeiten summieren sich, wenn eine Normreihe oder mehrere Datensätze z. B. aus einer Excel-Tabelle angelegt werden. So dauert das komplette Anlegen der größten verfügbaren Normreihe (DIN 65526) mit über 7000 Datensätzen etwa 45 Minuten.

Ein weiteres Problem stellt nach Aussage der Mitarbeiter der *ARC Solutions GmbH* die schlechte Wartbarkeit des Programms dar. Das langfristige Ziel ist es daher, *BSV* zu ersetzen.

Im Zuge der Umstellung soll ebenfalls das Format nach (*CAD-Normteiledatei nach DIN* 1990) aufgegeben werden, da nicht sicher ist, wann die DIN die Pflege der Dateien einstellen wird. Als größte Schwierigkeit sind dabei die in die Merkmaldateien eingebetteten Algorithmen zu sehen, da diese in *FORTRAN77* geschrieben sind. Sie müssen in eine Skriptsprache umgewandelt werden, um sie einfacher änderbar und für *Remarc* ausführbar zu machen.

Die Aufgabe ist es, die Realisierbarkeit der Umstellung zu prüfen und einen Weg zur Umsetzung aufzuzeigen. Weiterhin soll ein neues, leistungsfähigeres Format zum Speichern der Daten evaluiert und ein Programm zur Konvertierung der Merkmaldateien in das neue Format prototypisch implementiert werden. Der Fokus der Bachelorarbeit liegt dabei auf dem Darlegen eines Weges. Die vollständige Abbildung der Merkmaldaten ist nicht Ziel der Arbeit. Das Anpassen von *Remarc* an das neue Datenformat ist Thema einer zweiten Bachelorarbeit, welche zur gleichen Zeit bearbeitet wird.

1.3 Aufbau der Arbeit

Die Arbeit ist in fünf Kapitel gegliedert. Dabei erfolgt in Kapitel 1 zunächst eine kurze Vorstellung der Firma *ARC Solutions GmbH*, eine Problembeschreibung sowie die Darstellung der Aufgaben und Ziele. In Kapitel 2 werden Grundlagen zu *Remarc*, *EMF*

und den Merkmaldateien betrachtet und danach wird die grundlegende Funktionsweise eines *ANTLR*-Interpreters anhand zweier Beispiele erläutert. Kapitel 3 behandelt die Konzeption eines Konverters. Dazu werden zunächst verschiedene mögliche Dateiformate zur Speicherung der konvertierten Daten diskutiert. Anschließend werden die Datenstruktur, die Konvertierungsmethoden sowie die Anforderungen an die Benutzeroberfläche festgelegt. Am Ende des Kapitels wird die Struktur des Programms beschrieben. Die konkrete Umsetzung des Prototypen wird in Kapitel 4 vorgestellt. Dabei wird zunächst noch einmal genauer auf den Programmaufbau eingegangen. Danach werden die beiden Teile der *ANTLR*-Grammatik zur Konvertierung der DIN V 4001-Dateien und die Grammatik zum Umwandeln der *FORTRAN77*-Algorithmen ausführlich erläutert. Abschließend wird in Kapitel 5 eine Zusammenfassung der Ergebnisse dargestellt sowie ein Ausblick auf weitere Arbeiten und Entwicklungen gegeben.

Kapitel 2

Grundlagen

2.1 Remarc

Remarc ist ein Programm, das Teile für verschiedene CAD-Programme verwalten und generieren kann. Es basiert auf der RCP von *Eclipse* und ist als *Eclipse*-Plugin in *Java* implementiert. Die *Remarc*-interne Datenstruktur Component Supply Management (CSM) wurde mittels *Eclipse Modeling Framework (EMF)* erstellt.

2.2 EMF

Das *EMF* bietet die Möglichkeit, mit Hilfe einer grafischen Oberfläche komplexe Datenstrukturen zu entwerfen. So können Klassen relativ einfach von anderen Klassen abgeleitet und um Attribute oder Methoden erweitert werden. *EMF* generiert *Java*-Klassen, die den Vorgaben des *EMF*-Datenmodells entsprechen. Außerdem können Objekte von *EMF*-generierten Klassen serialisiert und als Extensible Markup Language (XML)-Datei gespeichert werden. Ein weiterer Vorteil ist die gute Integration in *Eclipse*.

2.3 CAD-Normteiledatei nach DIN

Der Fachbericht 14 (*CAD-Normteiledatei nach DIN* 1990) beschreibt, in welchem Format Merkmal- und Geometriedaten von standardisierten Normteilen gespeichert werden müssen. Diese Arbeit beschäftigt sich dabei ausschließlich mit den Merkmaldaten. Grundsätzlich werden die Daten in einem Textformat nach dem American Standard Code for Information Interchange (ASCII) gespeichert. Dadurch sind sie mit Hilfe eines beliebigen Texteditors änderbar. Die Informationen sind zeilenweise strukturiert. Jede Zeile beginnt mit einem Code (Tag) aus zwei oder vier Großbuchstaben. Listing A.1 zeigt eine beispielhafte Merkmaldatei der Norm „DIN ISO 14“. Die gesamte Datei wird von den Tags `BD` und `ED` umschlossen. Bei Zeilen, die mit `C` beginnen, handelt es sich um Kommentare, diese werden beim Parsen nicht beachtet.

Von Zeile 2 bis Zeile 23 sind allgemeine Informationen zur Norm zu finden, wie z. B. die Norm-Nummer (`ID,NN`), der Norm-Titel (`ID,NT`) und die Norm-Bezeichnung (`ID,NB`). Letztere besteht aus einem *FORTRAN77*-Algorithmus, welcher interpretiert werden muss.

Auf die allgemeinen Daten folgen die Beschreibungen der Merkmale. Die Merkmalbeschreibungssätze beginnen mit `MM` und weiteren zwei Großbuchstaben, die allerdings für den Interpreter nicht relevant sind und nur dem Verständnis dienen (vgl. *CAD-Normteiledatei nach DIN* 1990, S.85f.). Darauf folgen vier Zahlen: Version, Identifikationsnummer, verantwortliche Stelle und Status. Letzterer ist für das Programm wichtig, da er Auskunft darüber gibt, ob und an welcher Stelle Daten zu diesem Merkmal vorhanden sind. Sie können entweder gar nicht (`Status=0`), in den Merkmaldatensätzen (`Status=1`), als Konstante oder Algorithmus (`Status=2`), als Wertebereich (`Status=3`) oder als Referenz auf eine andere Norm (`Status=4`) gespeichert sein. Die Kennung des Merkmals, welche als interner Name bei Berechnungen benutzt wird, schließt sich dem Status an. Nach der Kennung stehen die drei Textangaben „Merkmalbenennung“, „Maßbuchstaben“ und „Maßeinheit“. Der Datentyp des Merkmals schließt die Zeile ab: `T` für „Text“ oder `Z` für „Zahl“.

Der letzte Abschnitt der Merkmaldatei umfasst den Inhalt der Merkmale. Er beginnt mit den Merkmalalgorithmen. Diese bestehen immer aus dem Anfangscode `MA`, gefolgt von einem Platzhalter, der Merkmalkennung des Zielmerkmals und dem eigentlichen *FORTRAN77*-Algorithmus (vgl. *CAD-Normteiledatei nach DIN* 1990, S.88f.).

Mit Hilfe der darauf folgenden Wertebereiche können die möglichen Werte des Merkmals durch ein oder mehrere Intervalle oder Wertvorgaben eingeschränkt werden. Auf diese Weise können eingegebene Werte auf Gültigkeit geprüft werden. Die Wertebereiche können von anderen Merkmalen abhängen (vgl. *CAD-Normteiledatei nach DIN* 1990, S.89ff.). Die letzte Möglichkeit, Werte vorzugeben, besteht im Festlegen von Datensätzen. Datensätze beginnen mit dem Tag `DA`. Darauf folgen drei Zahlen, welche die Version des Datensatzes, dessen Identifikationsnummer sowie die verantwortliche Stelle beinhalten. Die darauf folgenden Werte stellen die Daten aller Merkmale mit dem Status `1` („in der Datei“) durch Kommas getrennt dar. Die Reihenfolge entspricht dabei der, in der sie definiert wurden.

2.4 ANTLR

ANother Tool for Language Recognition (ANTLR) ist ein Werkzeug, um Texte auf Basis einer Grammatik zu interpretieren. Es kann dabei helfen, Programmcode zu interpretieren oder Datenformate systematisch einzulesen. Dabei trennt *ANTLR* den Vorgang in zwei Teilschritte: das Aufteilen der Zeichenkette in Tokens durch einen Lexer und das Interpretieren dieser Tokens mit Hilfe eines Parsers. Für beide Teilaufgaben generiert der *ANTLR*-Compiler aus der Grammatikdatei je eine Klasse in einer der objektorientierten Sprachen *Java*, *C++* oder *Sather*. Durch den stromorientierten Ansatz ist *ANTLR* in der Lage, sehr große Dateien einzulesen, da sich immer nur eine begrenzte Anzahl Zeichen im Arbeitsspeicher befindet. Im Rahmen dieser Arbeit wurde *ANTLR* in der Version 2.7.7 (PARR 2006) verwendet, da es für diese Version bereits eine Grammatikdatei für *FORTRAN77* (DRAGON 2007) gibt.

Anhand zweier einfacher Beispiele wird im Folgenden die grundlegende Arbeitsweise von *ANTLR* erläutert. Dabei soll eine Datei interpretiert werden, die, jeweils durch Kommas getrennt, den Namen, den Vornamen und das Alter von verschiedenen Personen enthält. Jede Zeile enthält je eine Person, wie die Beispieldatei A.4 zeigt. Da der Interpreter nur als Beispiel dient, soll an dieser Stelle die Einschränkung gelten, dass die Namen der Personen ausschließlich Groß- und Kleinbuchstaben des englischen Alphabetes enthalten. Außerdem ist die Datei in jedem Fall mit einem Zeilenumbruch abzuschließen.

2.4.1 Lexer

Sowohl der Quellcode eines Programms als auch der Inhalt der Beispieldatei oder der Merkmaldateien bestehen aus Schlüsselwörtern. Allerdings sind diese Dateien aus Sicht des Konverters zunächst nur eine einzige Zeichenfolge. Die Aufgabe des Lexers ist es, diese Zeichenfolge in Teilstücke, sogenannte Tokens, zu zerlegen. Ein Token ist eine *ANTLR*-Datenstruktur, die neben dem Text der Teilstücke auch deren Tokentyp sowie die Zeile und die Spalte, an der das Token im Gesamttext stand, speichert.

Zum Zerlegen der Beispieldatei A.4 wird der ExampleLexer A.2 benutzt. Zu Beginn wird in Zeile 1 der Lexer deklariert und der spätere Klassenname festgelegt. Die Option `exportVocab` definiert einen Namen für das Vokabular, das die vom Lexer verwendeten Tokentypen enthält. Sie ist nur wichtig, falls Lexer und Parser in getrennten Grammatikdateien enthalten sind, da der Parser sonst nicht darauf zugreifen kann. Der `k`-Wert legt die Anzahl der Zeichen fest, die der Lexer einlesen darf, bevor er den Tokentyp festlegen muss. Ein Tokentyp wird durch einen Namen und ein Muster festgelegt. Das Muster bestimmt, welche Teile des Textes auf einen Tokentyp passen und orientiert sich dabei an Regulären Ausdrücken (vgl. *ANTLR Reference Manual* 2005, Kapitel „ANTLR Meta-Language“). Es kann aus anderen Mustern zusammengesetzt sein. Zusätzlich können vor und nach dem Muster in geschweiften Klammern *Java*-Befehle stehen. Wenn sie davor stehen, werden sie vor dem Prüfen des Musters ausgeführt. Stehen sie dahinter, werden sie nach dem Prüfen und nur, wenn das Muster passt, ausgeführt. Außerdem besteht noch die Möglichkeit, Bedingungen zu bestimmen, die erfüllt sein müssen, damit das Muster überhaupt geprüft wird.

Der Tokentyp `NEWLINE` wird in Zeile 7 ff. definiert. Da unter den Plattformen *Windows*, *Linux* und *Macintosh* die Zeilenumbrüche durch unterschiedliche ASCII-Zeichen gekennzeichnet werden, sind die drei Möglichkeiten zu berücksichtigen und mit einem logischen Oder (`|`) zu verknüpfen. Wenn das Muster passt, das aktuelle Zeichen also tatsächlich ein Zeilenumbruch ist, wird die Funktion `newline()` der Klasse `Lexer` aufgerufen, die dafür sorgt, dass der Lexer den Tokens die Zeilen und Spalten richtig zuordnen kann. Dieser Tokentyp verlangt nach einem `k`-Wert von mindestens zwei, da der Lexer sonst nicht zwischen den beiden Optionen `"\r\n"` und `'\r'` unterscheiden kann. `ZEICHENKETTE` und `ZAHL` setzen sich aus einem oder mehreren (+) `BUCHSTABE`n bzw. `ZIFFER`n zusammen. Das `KOMMA` ist in jedem Fall ein einzelnes „`,`“ und dient zum Trennen der Datenfelder. Ein `BUCHSTABE` wiederum ist entwe-

der ein Klein- (`'a'..'z'`) oder (`|`) ein Großbuchstabe (`'A'..'Z'`) des englischen Alphabetes. Das Schlüsselwort `protected` sagt aus, dass dieser Typ nur intern, d. h. zur Kombination mit anderen Tokentypen, verwendet werden kann. Tokens dieses Typs können nicht im resultierenden `TokenStream` auftauchen. `ZIFFER` besteht analog zu `BUCHSTABE` aus einer Ziffer zwischen Null und Neun (`'0'..'9'`).

Der `TokenStream`, den der Lexer aus der Beispieldatei A.4 erzeugt, ist in Abbildung 2.1 teilweise dargestellt. Dabei entspricht ein Kästchen einem Token. Der obere Teil des Kästchens enthält den Text des Tokens, darunter findet sich der Tokentyp.

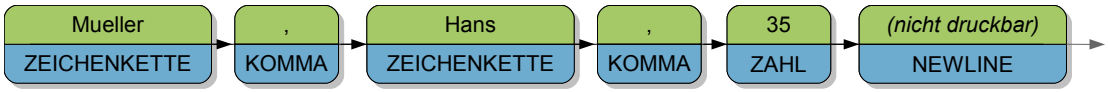

Abbildung 2.1: Schematische Darstellung des resultierenden TokenStreams einer Zeile der Beispieldatei A.4

2.4.2 Parser

Der Parser interpretiert den vom Lexer erzeugten `TokenStream`. Abhängig vom Typ der Ausgangsdaten existieren zwei grundlegend verschiedene Ansätze:

- Das Einlesen in eine beliebige Datenstruktur, falls es sich um Daten handelt (siehe Abbildung 2.2).
- Das Aufbauen eines Abstract Syntax Tree (AST), falls es sich um einen Algorithmus handelt (siehe Abbildung 2.3).

Da es sich bei den Merkmaldateien um Daten handelt und für die enthaltenen *FORTRAN77*-Algorithmen bereits eine Grammatik (DRAGON 2007) existiert, wird das Erstellen eines AST in dieser Arbeit nicht behandelt.

Um das Einlesen der Beispieldatei A.4 fortzusetzen, wird ein Parser benötigt, der den `TokenStream` (Abb. 2.1) interpretiert. Dieser Parser geht im Prinzip genauso vor wie der Lexer. Der Unterschied besteht darin, dass für den Lexer die kleinste Größe ein einzelnes Zeichen ist, während es beim Parser die Tokens sind.

In Zeile 1 wird der Parser deklariert und darunter werden die Optionen festgelegt. Die Option `importVocab` bestimmt, dass das Vokabular des ExampleLexers verwendet

werden soll. Danach werden analog zum Lexer die Regeln definiert, welche wieder aus Name und Muster bestehen. Sie werden nach dem Übersetzen in eine *Java*-Klasse als öffentliche Methode verfügbar sein. Um die Regeln von den Tokentypen unterscheiden zu können, werden Tokentypen in Großbuchstaben und Parserregeln in Kleinbuchstaben geschrieben. Die initiale Regel des Parsers ist die `file`-Regel, die sich aus keiner, einer oder mehreren Zeilen zusammensetzt und von einem Dateiende-Token (`EOF`) abgeschlossen wird, wobei auf jede Zeile ein `NEWLINE` folgen muss. Dabei stellt die Zeile (`line`) eine weitere Regel dar. Diese besteht aus einer `ZEICHENKETTE`, gefolgt von einem `KOMMA` sowie einer weiteren `ZEICHENKETTE`, einem `KOMMA` und einer `ZAHL`. Der entscheidende Punkt bei dieser Regel ist die Tatsache, dass es die Möglichkeit gibt, die Tokens aus dem `TokenStream` einer Variablen zuzuweisen. Auf diese Weise können die Daten weiterverarbeitet werden, falls das Muster der Regel zutrifft.

Der ExampleParser gibt z. B. das Alter der in der Beispieldatei A.4 gelisteten Personen auf der Standardausgabe aus (A.5).

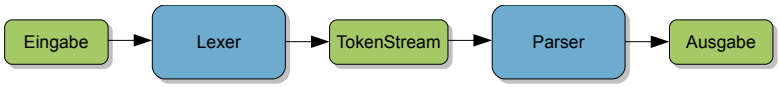

Abbildung 2.2: Arbeitsablauf zum Einlesen einer Datei mit ANTLR

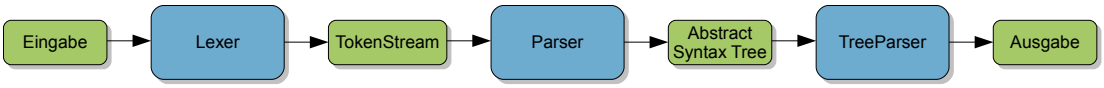

Abbildung 2.3: Arbeitsablauf zum Einlesen eines Algorithmus mit ANTLR

2.4.3 TreeParser

Als zweites Beispiel soll ein einfacher Rechner für mathematische Ausdrücke nach (MILLS 2005) dienen. Im Quellcode A.6 ist von Zeile 1 bis Zeile 11 ein einfacher Lexer zu sehen. Durch die Option `buildAST=true` wurde der Parser (Zeile 13 bis 20) um eine spezielle Syntax erweitert. Die Regeln des Parsers funktionieren weiterhin wie im Abschnitt 2.4.2 beschrieben, allerdings werden alle Tokens in einen AST eingehangen. Dieser Vorgang kann durch zwei Modifizierer gesteuert werden. Wird an einen Tokentyp ein Caret (`^`) angehängt, so wird das entsprechende Token die Wurzel

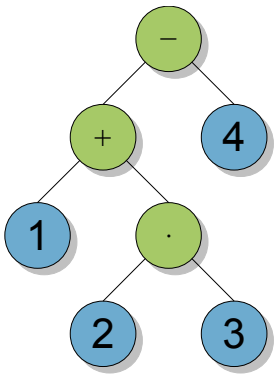

Abbildung 2.4: Darstellung eines Syntaxbaumes

des aktuellen Syntax-Unterbaumes. Durch ein nachgestelltes Ausrufezeichen (`!`) gekennzeichnete Tokentypen werden nicht in den AST eingehangen. Im Beispiel dient etwa das Semikolon (`SEMI!`) zum Abschließen des Ausdrucks und ist deshalb nicht für die Berechnung von Bedeutung. Abbildung 2.4 zeigt den AST für den Ausdruck „$1+2\cdot 3-4$;“.

In Zeile 22 wird die *Java*-Klasse `java.lang.Math` eingebunden, da diese später zur Berechnung der Potenzen benötigt wird. Die Klasse `ExpressionTreeWalker` wird danach deklariert. Sie besteht nur aus einer Regel `expr`, die sich selbst rekursiv aufruft. Die Option `returns [double r]` legt fest, dass die Regel, die durch den *ANTLR*-Compiler in eine *Java*-Funktion umgewandelt wird, einen Rückgabewert vom Typ `double` besitzt. Außerdem schreibt sie vor, dass das Endergebnis der Variablen `r` zugewiesen werden muss. In Zeile 26 werden die drei Variablen `a`, `b` und `r` deklariert. Die Variable `r` wird außerdem mit einem Wert initialisiert, da es sonst möglich ist, dass auf die Variable zugegriffen wird, obwohl sie noch nicht initialisiert wurde. Danach folgt das Muster der Regel, welches sich aus mehreren alternativen Untermustern zusammensetzt. Das Untermuster `#(PLUS a=expr b=expr)` bedeutet, dass es nur dann passt, wenn der aktuelle Token vom Typ `PLUS` ist und mindestens zwei Kinder besitzt. Die beiden Kinder sind ebenfalls ASTs, werden durch je einen weiteren Aufruf der Methode `expr` interpretiert und der resultierende Zahlenwert wird den Variablen `a` und `b` zugewiesen. Falls diese Regel passt, wird `r=a+b;` aufgerufen. Die Funktion `expr` gibt nun also die Summe der Werte der beiden Teilbäume zurück. Die Untermuster für `MINUS`, `MUL`, `DIV`, `MOD` und `POW` folgen dem vorhergehenden analog. Handelt es sich bei dem aktuellen Token um ein Token vom Tokentyp `INT`, trifft keines der anderen Untermuster zu. Über die Variable `i` lässt sich mit `i.getText()`

der Inhalt des Tokens auslesen und in eine Zahl vom Typ `double` umwandeln, welche gleichzeitig den Rückgabewert von `expr` darstellt. Durch die Definition im Lexer ist sichergestellt, dass alle Tokens vom Typ `INT` eine als Integer interpretierbare Zeichenkette enthalten.

Aufgrund der Anordnung der Regeln im `ExpressionParser` wird der AST so aufgebaut, dass niederwertige Rechenoperationen im Baum über den höherwertigen stehen. Da der `ExpressionTreeWalker` die Kinder vor dem eigentlichen Knoten berechnet, ist die Einhaltung der Prioritäten der Operationen gewährleistet. Das Ergebnis des Beispiels „$1+2\cdot 3-4$;“ gibt der `TreeParser` richtigerweise mit „3“ an.

Kapitel 3

Konzeption

Der Konverter soll als *Eclipse*-Plugin auf Basis des *Eclipse* RCP in *Java* implementiert werden. Das RCP bietet eine Reihe von Hilfestellungen bei der Erstellung der Anwendung und der Dialoge. Von Vorteil ist außerdem die gute Integration des *EMF* in *Eclipse*.

3.1 Zielformat

Zunächst gilt es festzulegen, in welches Format die Merkmaldateien umgewandelt werden sollen. Für das Zielformat gelten folgende Anforderungen:

- Es muss sowohl Daten als auch Algorithmen aufnehmen können,
- die enthaltenen Daten sollen unproblematisch änderbar sein, um Anpassungen zu erleichtern und
- es soll möglichst einfach an *Remarc* angebunden werden können.

Um die Änderung der *FORTRAN77*-Algorithmen zu erleichtern, sind diese in eine Skriptsprache umzuwandeln, die z. B. durch das *Bean Scripting Framework (BSF)* interpretiert werden kann. Auf diese Weise entfällt der Kompiliervorgang nach jeder Änderung oder das Schreiben eines *FORTRAN77*-Interpreters. Im Folgenden sollen vier verschiedene grundlegende Ansätze zur Entwicklung des Zielformates bezüglich ihrer Anpassbarkeit und ihrer Integration in *Remarc* diskutiert werden.

3.1.1 Einfaches Datenformat

Eine Möglichkeit zur Speicherung wäre die Entwicklung eines eigenen, textbasierten Formates. Dies hätte den Vorteil, dass die Daten einfach mit Hilfe eines Texteditors änderbar wären. Die Durchsuchbarkeit lässt sich nicht pauschal bewerten, da sie von der genauen Gestaltung des Formates abhängt. Allerdings besitzt ein solches Format den gravierenden Nachteil, dass zur Einbindung in *Remarc* ein eigener Interpreter für diese Dateien geschrieben werden müsste.

3.1.2 Binäres Format von serialisierten *Java*-Objekten

Ein solches Format basiert auf Klassen, die das Interface `java.io.Serializable` implementieren. Dieses Interface deklariert keine Methoden und dient ausschließlich der Kennzeichnung serialisierbarer Klassen. Das Besondere an diesen Klassen ist, dass Objekte dieser Klassen in einer Datei abgespeichert werden können. Dieser Vorgang ist unkompliziert, allerdings entstehen dabei binäre Dateien. Um diese zu verändern, wird ein eigens für diese Dateien entwickelter Editor benötigt.

3.1.3 XML

Formate, die auf XML basieren, zeichnen sich durch eine hierarchische Struktur aus. Diese kann genutzt werden, um die Daten strukturiert zu speichern. Außerdem existieren für *Java* fertige Klassen, die das Speichern in XML erleichtern. Da XML textbasiert ist, kann es mit Hilfe eines beliebigen Texteditors bearbeitet werden. Allerdings wird, wie unter 3.1.1 beschrieben, eine weitere Zwischenschicht benötigt, die die Daten aus der XML-Struktur in eine interne *Remarc*-Struktur bringt.

3.1.4 CSMXML

Remarc benutzt intern eine Datenstruktur namens CSM. Da diese, wie unter 2.1 beschrieben, mit Hilfe von *EMF* erstellt wurde, sind alle CSM-Objekte in eine XML-Datei serialisierbar. Eine von CSM abgeleitete *EMF*-Struktur wäre ebenfalls serialisierbar und würde sich ohne Aufwand in *Remarc* integrieren lassen. Außerdem sind

die wichtigsten Klassen und deren Attribute bereits in CSM vorhanden, sodass diese wiederverwendet werden können.

Damit erfüllt CSMXML die genannten Forderungen am besten. Es ist einfach anzuwenden, lässt sich dank XML mit Hilfe eines beliebigen Texteditors bearbeiten und erfordert die wenigsten Änderungen in *Remarc*, da dieses intern bereits mit CSM arbeitet.

3.2 Datenstruktur

Die Datenstruktur, in der die Daten der Merkmaldateien gespeichert werden, soll von der in *Remarc* verwendeten Struktur abgeleitet werden. Dazu wird ein eigenes Paket `de.htwm.kgraefe.tabfileconverter.tabfile` angelegt. Die Wurzel der Struktur bildet dabei die Klasse `CSMXML`, welche alle Datenelemente direkt oder indirekt referenziert. Dadurch ist sichergestellt, dass alle Elemente enthalten sind, wenn ein Objekt der Klasse `CSMXML` serialisiert wird. Abbildung 3.1 zeigt das Klassendiagramm der verwendeten Klassen. Tatsächlich ist die CSM-Struktur umfangreicher, es wird aber nur ein Teil der dort definierten Klassen benötigt.

Die `CSMXML` kann mehrere CSM-Klassen aufnehmen, allerdings wird der Konverter für jede Norm eine eigene Datei anlegen, da die Normen so auf Basis des Dateinamens leichter wiederzufinden sind. Außerdem referenziert die `CSMXML` eine Liste aller primitiven Datentypen, die in der Datei vorkommen. Obwohl CSM weitaus mehr Datentypen anbietet, werden lediglich die beiden Typen `StringType` und `DoubleType` verwendet, da die Merkmaldateien nur zwischen Text und Zahlen unterscheiden.

Jede Norm wird in einer CSM-Klasse gespeichert. Diese enthält alle allgemeinen Attribute der Norm wie Normnummer und Normtitel. Darüber hinaus enthält sie eine Liste aller Merkmalbeschreibungen als Referenz. `CSMTabfileClass` erweitert die CSM-Klasse um eine Liste aller Merkmaldatensätze.

Objekte der Klasse `CSMTabfileCharacteristicDef` repräsentieren die Merkmalbeschreibungen. Sie erweitern die Klasse `CSMCharacteristicDefinition` um die Attribute `unit` für die Maßeinheit und `dimensionalLetters` zum Speichern der Maßbuchstaben sowie je einer Referenz auf den Typ des Merkmals und auf

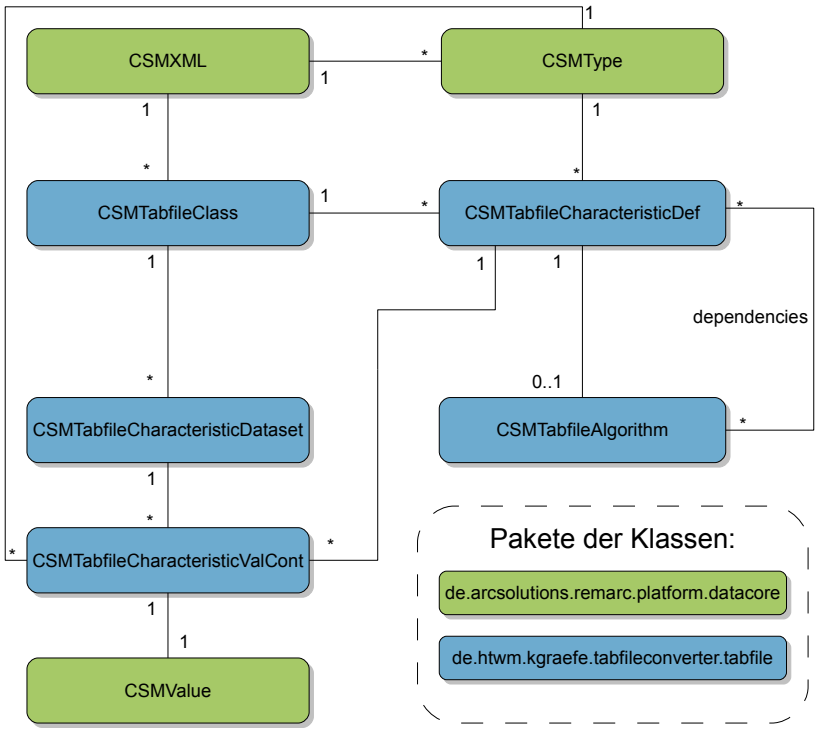

Abbildung 3.1: Klassendiagramm der zur Speicherung der Daten verwendeten Klassen

einen Merkmalalgorithmus. Ist das Merkmal nicht durch einen Algorithmus definiert, so ist die Referenz `null` zu setzen.

Die Merkmalalgorithmen enthalten ein Attribut, in dem die Skriptsprache des Algorithmus angegeben wird sowie eines, welchen den eigentlichen Algorithmus als Zeichenkette enthält. Weiterhin beinhaltet es eine Referenz auf die Merkmalbeschreibung des Merkmals, dessen Wert mit Hilfe des Algorithmus errechnet werden soll und eine Liste aller Merkmale, von denen der Algorithmus abhängt.

Instanzen der Klasse `CSMTabfileCharacteristicDataset` erfassen die Merkmaldatensätze, welche lediglich aus einer Liste von Wertecontainern bestehen.

Diese Wertecontainer enthalten den konkreten Wert und den Typ des Wertes sowie einige andere Attribute, die aber für die umgewandelten Normen nicht relevant sind. `CSMTabfileCharacteristicValCont` erweitert die vorhandene CSM-Klasse `CSMCharacteristicValCont` um eine Referenz auf den zugehörigen Merkmaldatensatz.

`CSMValue` und `CSMType` sind abstrakte Klassen, deren Methoden für jeden von CSM unterstützten Datentyp separat implementiert wurden. Auf diese Weise können die Container jeden dieser Typen enthalten.

3.3 Konvertierungsmethode

Die *BSV* hat die Dateien zeichenweise eingelesen und interpretiert. Allerdings ist dieser Weg wenig flexibel und wird sehr komplex werden. Besser ist es daher, ein auf diesen Anwendungsfall spezialisiertes Framework wie *ANTLR* zu benutzen. Mit Hilfe von *ANTLR* kann das Einlesen der Datei von einer höheren Ebene aus gesteuert werden. Anstatt jedes Zeichen einzeln einzulesen und in einem aktuellen Kontext zu interpretieren, erstellt man mit der *ANTLR*-Grammatik eine Beschreibung der Datei. Natürlich funktionieren die generierten Klassen ähnlich wie der Ansatz der *BSV*, allerdings ist es die Aufgabe des *ANTLR*-Compilers, sicherzustellen, dass die eingelesenen Zeichen korrekt interpretiert werden. Dadurch werden Fehler vermieden und der Aufwand reduziert.

3.3.1 Daten

Zum Einlesen der Merkmaldateien ist eine eigene *ANTLR*-Grammatik zu schreiben. Dabei sollen nichtinterpretierbare Zeilen ignoriert, auf Wunsch aber auf der Standardausgabe ausgegeben werden.

3.3.2 Algorithmen

Die Algorithmen sind in *FORTRAN77* implementiert. Sie sollen mit Hilfe der *FORTRAN77*-Grammatik für *ANTLR* von (DRAGON 2007) in einen AST eingelesen werden. Der AST soll durch einen eigenen `TreeParser` in die Skriptsprache *Tool command language (Tcl)* umgewandelt werden. *Tcl* eignet sich besonders, da es über eine sehr einfache Syntax verfügt und somit leicht automatisiert geschrieben werden kann. Außerdem werden in *Tcl* keine Datentypen unterschieden, wodurch das Parsen erleichtert wird, da die Tokens im AST generell nur Text enthalten.

3.4 Benutzeroberfläche

Die Benutzeroberfläche soll möglichst einfach gehalten werden. Sie soll aus zwei Ansichten bestehen, eine zur Eingabe der Parameter und eine zur Anzeige des Fortschritts. Auf eine Übersetzung der Benutzeroberfläche wird an dieser Stelle verzichtet. Die Standardsprache ist Englisch.

3.4.1 Eingabeansicht

Da der Konverter mehrere Merkmaldateien in einem Durchlauf bearbeiten soll, muss es eine Eingabemöglichkeit für mehrere Dateien geben. Dazu bietet sich ein normaler `FileDialog` aus dem Paket `org.eclipse.swt.widgets` an, welcher mit der speziellen Eigenschaft `SWT.MULTI` ausgestattet ist, um das Auswählen mehrerer Dateien zu ermöglichen. Die ausgewählten Dateien sind in einer Liste anzuzeigen. Neben den Eingabedateien soll das Ausgabeverzeichnis über einen `DirectoryDialog` des selben Paketes ausgewählt werden. Zur Angabe des eingestellten Ausgabeverzeichnisses ist ein deaktiviertes `Text`-Feld vorgesehen. Der Konverter kann auf Wunsch ignorierte Zeilen auf der Standardausgabe ausgeben. Damit der Benutzer dieses Verhalten steuern kann, soll ein Ankreuzfeld angeboten werden.

Die drei Abschnitte Eingabe, Ausgabe und Optionen sind zur besseren Übersicht in Gruppen einzuteilen, wie sie in Abbildung 3.2 zu sehen sind.

3.4.2 Fortschrittsanzeige

Die Fortschrittsanzeige soll ein mehrzeiliges Textfeld erhalten, welches das Protokoll anzeigt. In diesem Protokoll wird der Name der Datei, die der Konverter gerade bearbeitet, ausgegeben. Unter diesem Textfeld soll ein Fortschrittsbalken angezeigt werden, damit der Benutzer ungefähr abschätzen kann, wann der Konverter fertig sein wird. Die komplette Fortschrittsanzeige ist in Abbildung 3.3 zu sehen.

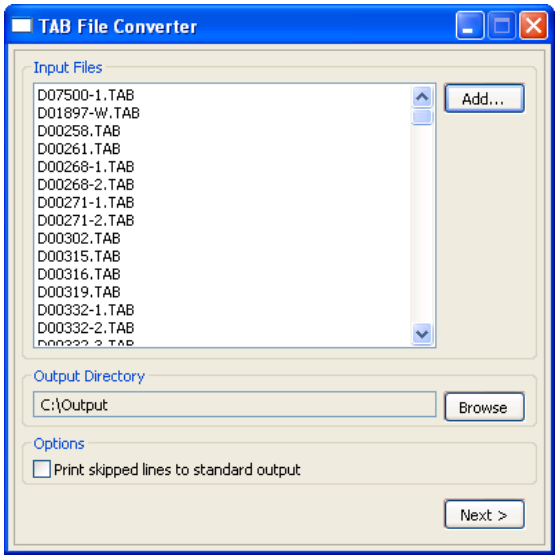

Abbildung 3.2: Bildschirmfoto der Eingabeansicht

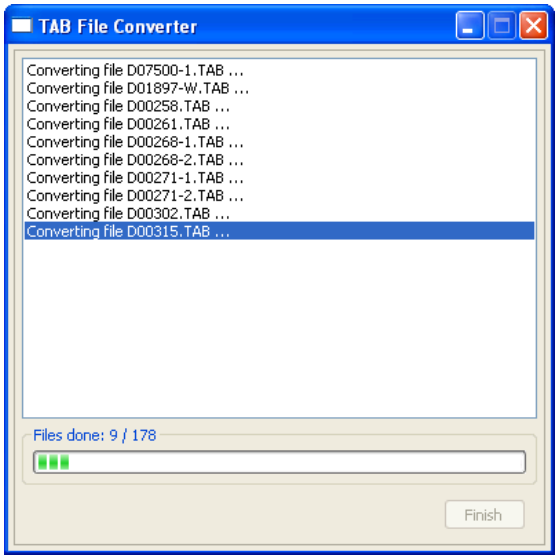

Abbildung 3.3: Bildschirmfoto der Fortschrittsanzeige

3.5 Programmaufbau

Die Struktur des Programmes teilt sich, wie in Abbildung 3.4 dargestellt, in drei große Teile: den Steuerungsteil, die grafische Oberfläche und die Verarbeitungskomponente. Dabei koordiniert die Klasse `Control` sämtliche kommunikativen Vorgänge und übernimmt zusätzlich einige einfache Funktionen wie das Auswählen der Eingangsdateien oder das Prüfen der Eingabeparameter. Die Klasse `ConverterThread` verarbeitet alle angegebenen Dateien mit Hilfe der von *ANTLR* erzeugten Lexer- und Parser-Klassen. Der `ConverterThread` wird in einem eigenen Thread ausgeführt, da sonst die Anzeige des Fortschritts blockiert wäre. Er teilt der `Control`-Klasse den aktuellen Fortschritt nach jeder fertiggestellten Datei mit.

Die Klasse `InputView` dient der Anzeige der bereits ausgewählten Dateien, des Zielverzeichnisses und der Option zum Ausgeben der ignorierten Zeilen. Außerdem fängt sie Klick-Ereignisse auf die Schaltflächen ab, überlässt die Reaktion darauf allerdings der Steuerung. Die `OutputView` zeigt den aktuellen Fortschritt an. Sie wird dazu per Methodenaufruf von der Steuerung informiert, wenn diese vom Verarbeitungsthread einen neuen Fortschritt erhalten hat.

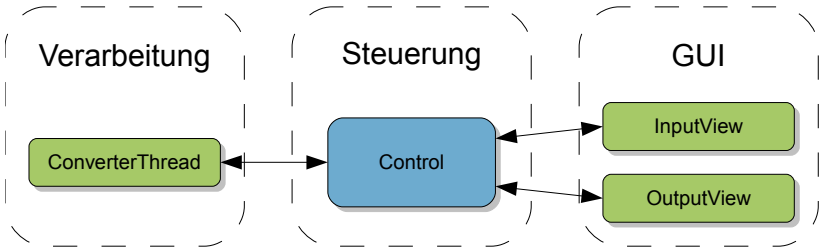

Abbildung 3.4: Verknüpfung der aktiven Klassen zur Laufzeit

Kapitel 4

Realisierung

Dieses Kapitel beschäftigt sich mit der Umsetzung des in Kapitel 3 diskutierten Konzeptes. Dabei wird zunächst der Aufbau des Programms genauer beleuchtet. Danach steht die Funktionsweise der *ANTLR*-Grammatiken im Mittelpunkt. Der Quellcode sowie eine unter *Windows* lauffähige Version des Programms befinden sich auf der beiliegenden CD.

Die Grammatiken werden mit Hilfe des *ANTLR*-Compilers in *Java*-Klassen übersetzt. Unter der Bedingung, dass sich der Pfad zum *Java*-Paket `antlr` in der Umgebungsvariablen `CLASSPATH` befindet, wird der Compiler mit Hilfe des Kommandos `java antlr.Tool grammatik.g` ausgeführt. Da für *ANTLR* 2 keine kostenlose Integration in *Eclipse* zur Verfügung steht, muss dieser Schritt separat erfolgen. Um diese Tätigkeit zu erleichtern, wurde ein Makefile erstellt, welches durch das Programm `make` interpretiert wird. Das Kommado `make tabfileconverter` übersetzt die *ANTLR*-Grammatiken in *Java*-Klassen und kompiliert diese so, dass die Klassen in den von *Eclipse* erwarteten Verzeichnissen erzeugt werden. Das Kommandozeilenprogramm `make` stammt aus dem *UNIX*-Umfeld und ist unter *Windows* über *Cygwin* zu installieren und auszuführen.

4.1 Programmaufbau

Das Programm ist als RCP-Anwendung implementiert. Diese hat zwei Perspektiven `InputPerspective` und `OutputPerspective` mit je einer View `InputView`

(Abb. 3.2) bzw. `OutputView` (Abb. 3.3). Die zentrale Klasse `Control` kommuniziert dabei mit den Fenstern. Sie überprüft die Eingaben und steuert die Datei- und Ordner-Auswahldialoge. Wird in der `InputView` auf den Button „Next“ geklickt, schaltet `Control` die Perspektive um und startet den Konverter.

Der Konverter läuft als Objekt der Klasse `ConverterThread` in einem eigenen Thread, damit die grafische Oberfläche aktualisiert werden kann. Er koordiniert die Konvertierung jeder Eingangsdatei, indem er sie zuerst öffnet, den Lexer und den Parser initialisiert, die Methode `file()` des Parsers aufruft und die resultierende CSMXML-Struktur in einer XML-Datei abspeichert. Um mit dem `Control`-Objekt zu kommunizieren, holt sich der Konverter mittels `Display.getDefault()` den Graphical User Interface (GUI)-Thread, in dem auch die Steuerung läuft. Mittels `asyncExec()` können Befehle im Kontext des GUI-Threads ausgeführt werden, allerdings wird als Übergabeparameter ein Objekt einer Klasse, die das Interface `Runnable` implementiert hat, benötigt. Zu diesem Zweck verwendet der Konverter die Hilfsklassen `Logger`, `ProgressSetter` und `Finisher`.

4.2 Lexer

Der Lexer für die DIN V 4001-Dateien im Quellcode A.7 ist prinzipiell genauso aufgebaut wie der ExampleLexer in A.2. In diesem Abschnitt sollen die wichtigsten Lexer-Regeln näher erläutert werden.

4.2.1 Die Regel „NEWLINE“

Laut (*CAD-Normteiledatei nach DIN* 1990) werden in den Dateien drei verschiedene Zeilenumbrüche verwendet. Ein normaler Zeilenumbruch kennzeichnet das Ende einer Datenzeile. Beginnt die nächste Zeile allerdings mit einem `STERN`, so ist der Zeilenumbruch als nicht existent zu werten, die Datenzeile geht also mit dem nächsten Parameter weiter. Ein Beispiel ist in der Beispieldatei A.1 in den Zeilen 41 und 42 zu sehen. Falls die neue Zeile mit einem `MINUS` beginnt, wie in A.1 (Zeile 5 bis 9) zu erkennen ist, bedeutet dies, dass die Zeichenkette, die aus einzelnen Teilen zu bestehen scheint, zu einem Parameter zusammengesetzt werden muss.

Generell passt das Muster der Regel auf einen normalen Zeilenumbruch der verschiedenen Betriebssysteme (`"\\r\\n" | '\\r' | '\\n'`). Danach kann entweder ein `STERN` oder ein `MINUS` folgen. Falls das zutrifft, wird mit Hilfe des *ANTLR*-Makros `$setType` ein neuer Typ für das Token gesetzt. Die beiden Typen `NONEWLINE` und `TEXTNEWLINE` werden dabei automatisch in das Vokabular des Lexers übernommen und müssen nicht noch einmal definiert werden.

Wenn das Muster der Regel passt, wird die Funktion `newline()` ausgeführt. Dies ist wichtig zum Zählen der Spalten- und Zeilennummern, welche zum einen für die korrekte Ausgabe von fehlerhaften Stellen in der Datei, zum anderen aber für die Funktion `getColumn()` bedeutend sind.

4.2.2 Die Regel „KOMMENTAR“

Kommentarzeilen beginnen immer mit einem `C` und beinhalten alle Zeichen bis zum Zeilenende (vgl. *CAD-Normteiledatei nach DIN* 1990, S.74f.). Da nicht alle möglichen Zeichen in einer Regel definiert werden können, wird ein negierender Operator in dem RegExp des Musters benötigt. Dazu dient die Tilde (`~`). Der negierende Operator darf nur verwendet werden, wenn vorher festgelegt wurde, welche Zeichen erlaubt sind. Hierfür muss in den Optionen die Zeile `charVocabulary='\3'..'\377';` eingefügt werden. Das Muster `'C' (~('\r'|'\n'))*` passt auf alle Zeichenketten, die mit `C` beginnen und mit einer beliebigen Anzahl beliebiger Zeichen außer (`~`) einem Zeilenumbruch (`'\\r'|'\\n'`) fortgeführt werden.

Um einen Kommentar handelt es sich aber nur, wenn das `C` in der ersten Spalte der Zeile steht. Die Zeile `{getColumn() == 1}?` vor dem Muster veranlasst *ANTLR*, in die generierte Klasse vor dem Prüfen des Musters einen Test einzubauen, um zu ermitteln, ob sich das aktuelle Zeichen in der ersten Spalte befindet. Wenn das nicht der Fall sein sollte, wird das Muster gar nicht erst geprüft.

4.2.3 Die Regel „IDNN“

Wenn eine Datenzeile mit der Zeichenkette `ID,NN` beginnt, handelt es sich um ein Token vom Typ `IDNN`. Diese Zeile beinhaltet den Eintrag der Normnummer der in der Datei beschriebenen Norm (vgl. *CAD-Normteiledatei nach DIN* 1990, S.77).

4.2.4 Die Regel „IDNT“

Der Titel der Norm wird in einer Datenzeile angegeben, die mit `ID,NT` gekennzeichnet ist (vgl. *CAD-Normteiledatei nach DIN* 1990, S.77). Die Kennung wird in einem Token vom Typ `IDNT` gespeichert.

4.2.5 Die Regel „MERKMAL“

Die Kennzeichen `MM,GA`, `MM,SM`, `MM,GM`, `MM,EM`, `MM,AT` und `MM,FM` markieren allesamt Datenzeilen, die eine Beschreibung eines Merkmals enthalten. Die unterschiedlichen Kennzeichen beschreiben verschiedene Merkmaltypen, welche allerdings nur das Verständnis der Merkmaldateien unterstützen sollen und für die maschinelle Interpretation nicht von Bedeutung sind (vgl. *CAD-Normteiledatei nach DIN* 1990, S.84ff.).

4.2.6 Die Regel „DA“

Datenzeilen, die Merkmaldatensätze enthalten, beginnen mit der Markierung `DA` (vgl. *CAD-Normteiledatei nach DIN* 1990, S.98f.). Dies entspricht dem gleichnamigen Tokentypen `DA`.

4.2.7 Die Regel „MA“

`MA` ist sowohl die Kennzeichnung für Datenzeilen, die Merkmalalgorithmen enthalten (vgl. *CAD-Normteiledatei nach DIN* 1990, S.88f.) als auch die Bezeichnung des Tokentypes, der für diese Kennzeichnung verwendet wird.

4.2.8 Die Regel „ZEICHENKETTE“

Zeichenketten werden in den DIN V 4001-Dateien von `APOSTROPH`en umschlossen. Zwischen den Apostrophen dürfen sich beliebig viele `ZEICHEN` und geschützte Apostrophe befinden. Ein `GESCHUETZTER_APOSTROPH` ist eine Zeichenkette, die aus genau zwei aufeinanderfolgenden Apostrophen besteht. Die Regel ist als `protected`

markiert, sodass GESCHUETZTER_APOSTROPH nicht als Tokentyp aufzufassen ist, sondern ausschließlich der Bildung weiterer, komplexerer Lexer-Regeln dient. Die Regel ZEICHEN wird in diesem Abschnitt noch diskutiert.

Wenn das Muster der Regel ZEICHENKETTE passt, werden von dem im Token enthaltenen Text zuerst die äußeren Apostrophe abgeschnitten (vgl. Quelltext A.7, Zeile 72) und anschließend die doppelten, maskierten Apostrophe durch einfache ersetzt (Zeile 75).

4.2.9 Die Regel „ZAHL“

Eine ZAHL kann mit einem Vorzeichen beginnen. Darauf folgt in jedem Fall mindestens eine ZIFFER. Danach kann ein Dezimal-PUNKT stehen, welchem allerdings mindestens eine weitere ZIFFER folgen muss.

In (*CAD-Normteiledatei nach DIN* 1990) heißt es auf Seite 75 im Abschnitt „Regeln für Zahlenangaben“ u.a. „Zahlen sind in der FORTRAN-Notation anzugeben, wahlweise in Integer, Real oder Exponentialdarstellung“. Allerdings wird dieser Festlegung in demselben Bericht im Abschnitt „Syntaxregeln der Merkmaldatei“ (S.169ff.) in der Regel „Zahl“ (S.171) widersprochen. Nach dieser Regel sind ausschließlich die Integer- und Realdarstellung erlaubt. Aus diesem Grund wurde die Exponentialschreibweise nicht berücksichtigt.

4.2.10 Die Regel „KENNUNG“

Kennungen werden für Namensangaben verwendet. Sie beginnen immer mit einem BUCHSTABEn, der von einer beliebigen Anzahl von BUCHSTABEn oder ZIFFERn gefolgt wird (vgl. *CAD-Normteiledatei nach DIN* 1990, S.75).

4.2.11 Die Regel „ZEICHEN“

Die ZEICHEN-Regel ist als protected gekennzeichnet und dient damit lediglich der Übersichtlichkeit der übrigen Regeln. Sie passt auf alle Zeichen, die innerhalb einer ZEICHENKETTE auftreten können.

4.3 Parser

Nachdem im letzten Abschnitt die Tokentypen festgelegt wurden, gilt es nun, den vom Lexer erzeugten TokenStream zu interpretieren. Dazu wird der passende Parser benötigt, welcher in Quellcode A.8 zu finden ist.

Im ersten Abschnitt von Zeile 1 bis Zeile 12 wird das *Java*-Paket festgelegt, in dem sich die generierte Klasse befinden soll. Außerdem werden die Pakete zum Aufbauen der internen Datenstruktur sowie ein paar Pakete, die zur Umwandlung der Daten notwendig sind, geladen. Danach folgen die Klassendefinition und Festlegung der Anzahl der Zeichen, die der Parser vorrausschauen soll (`k=3;`). Die Zeilen 19 bis 56 zeigen eingebetteten *Java*-Code. Es werden Variablen, ein eigener Konstruktor sowie eine Funktion zur Rückgabe des Ergebnisses definiert. Der Konstruktor baut die grundlegende Datenstruktur auf. Er erzeugt ein `CSMXML`- und ein `CSMTabfileClass`-Objekt und verknüpft diese miteinander. Außerdem initialisiert er `characteristicDefsInFile` sowie `characteristicDefById`. Die Liste `characteristicDefsInFile` referenziert alle Merkmalbeschreibungen, die den Status `1` haben und damit für die Interpretation der Merkmaldatensätze wichtig sind. Es handelt sich um eine Liste, da bei der Interpretation die Reihenfolge der Merkmalbeschreibungen von Bedeutung ist. `characteristicDefById` speichert Referenzen auf alle Merkmalbeschreibungen und wird benutzt, um das Feld aufzubauen, das die Abhängigkeiten eines Merkmalalgorithmus beinhaltet. Dazu wird die Merkmalkennung des jeweiligen Merkmals als Schlüssel verwendet. Nach dem eingebetteten *Java*-Code werden die Parser-Regeln definiert, welche im Folgenden dargelegt werden sollen.

4.3.1 Die Regel „file“

Diese Regel beschreibt die gesamte Datei. Sie erlaubt es, die Datei zeilenweise zu behandeln. Eine Datei beginnt immer mit einer Zeile, die durch die Regel `line` repräsentiert wird. Danach können beliebig viele Sequenzen von `NEWLINE` und `line` kommen. Abschließend kann ein weiteres `NEWLINE` vor dem zwingenden `EOF` (end of file) folgen.

Die Option `greedy=true` sorgt dafür, dass bei einem `NEWLINE` im Stream zuerst die obere Regel `NEWLINE line` geprüft wird. Würde die Option nicht gesetzt werden,

so würde der Lexer in den unteren Zweig springen und eine Fehlermeldung werfen, da er das erwartete `EOF` nicht bekommen hat.

4.3.2 Die Regel „line“

Diese Regel prüft die aktuelle Zeile gegen alle bisher implementierten Datenzeilen. Das Muster passt entweder auf einen `KOMMENTAR` oder eine der speziellen Regeln `normnummer`, `normtitel`, `merkmalbeschreibung`, `merkmalalgorithmus` oder `merkmaldatensatz`. Zuletzt wird noch die spezielle Regel `unbehandelt` geprüft, welche alle Zeilen auffängt, die noch nicht behandelt werden können. Da es sich bei dem `KOMMENTAR` nicht um eine Parser-Regel, sondern um einen Token handelt und die Regel `line` keine Aktion ausführt, falls das Muster gepasst hat, wird der Kommentar einfach ignoriert.

4.3.3 Die Regel „zusammengesetzte_zeichenkette“

Diese Regel aus Zeile 298 wurde vorgezogen, da ihr Verständnis für das Verstehen der darauffolgenden Regeln wichtig ist. Wie bereits in Abschnitt 4.2.1 erwähnt, müssen Zeichenketten, die durch ein `TEXTNEWLINE` getrennt wurden, wieder zusammengesetzt werden. Da die zusammengesetzten Zeichenketten in der Datei an jeder Stelle auftreten können, in denen eine Textangabe erwartet wird, ist es zweckmäßig, eine separate Regel zu erstellen.

Das Muster der Regel erwartet zunächst eine `ZEICHENKETTE`. Danach können beliebig viele Folgen `KOMMA TEXTNEWLINE ZEICHENKETTE` stehen. Das aktuelle Token wird dabei durch die Syntax `z:ZEICHENKETTE` der Variablen `z` zugewiesen, sodass im eingebetteten *Java*-Code damit gearbeitet werden kann. Weil Regeln keinen Rückgabewert haben können, muss auf eine Objektvariable zurückgegriffen werden. Diese Variable wird beim ersten Token des Musters neu gesetzt und bei allen weiteren `NEWLINE`-Tokens entsprechend erweitert. Bei der Anwendung der Regel muss darauf geachtet werden, dass das Ergebnis aus der Objektvariablen `zeichenkette` in eine lokale Variable kopiert wird, da die Objektvariable beim nächsten Aufruf der Regel überschrieben wird.

4.3.4 Die Regel „normnummer“

Am Anfang der Regel wird per eingebettetem *Java*-Code die lokale Variable `nummer` deklariert, welche beim Durchlaufen der Regel gesetzt wird. Die Datenzeile, die die Normnummer enthält, beginnt mit einem `IDNN`-Token. Danach stehen eine oder mehrere Sequenzen aus einem `KOMMA` und einer zusammengesetzten Zeichenkette. Die Zeile `nummer=zeichenkette;` kopiert, wie unter 4.3.3 beschrieben, den Inhalt der Objektvariable `zeichenkette` in die lokale Variable `nummer`.

Allerdings wird nur die erste Textangabe beachtet, da dies dem Verhalten der zu ersetzenden *BSV* entspricht. Sollen, wie unter (*CAD-Normteiledatei nach DIN* 1990, S.76) beschrieben, alle Textangaben mit einem Zeilenumbruch zusammengefügt werden, so ist die Zeile `{nummer += "\n" + zeichenkette;}` hinter dem letzten der beiden Aufrufe der Regel `zusammengesetzte_zeichenkette` einzufügen. Zum Abschluss wird die Normnummer in der `CSMClass`, die die Norm enthält, als `ID` gespeichert.

4.3.5 Die Regel „normtitel“

Zuerst wird wieder eine lokale Variable definiert, die den Titel der Norm speichert. Nach der Kennung der Datenzeile `IDNT` folgt eine Abfolge aus einem `KOMMA` und einer zusammengesetzten Zeichenkette. Wie bereits geschildert, wird der Inhalt der Objektvariable `zeichenkette` der lokalen Variable `normtitel` zugewiesen.

Über die Häufigkeit der Abfolge gibt es verschiedene Angaben. Im Abschnitt „Norm-Titel“ in (*CAD-Normteiledatei nach DIN* 1990, S.77) heißt es, dass diese Abfolge nur ein einziges Mal auftreten darf. Demgegenüber steht in Anhang 6 „Syntax der Merkmaldatei“ (S.169ff.) auf Seite 178 sinngemäß, dass sich der Titel aus mehreren Textangaben zusammensetzen kann, die durch ein `KOMMA` oder durch ein `KOMMA` und ein `NONEWLINE` getrennt sind. Letztere Variante wurde implementiert, da diese die erste impliziert. Die einzelnen Textangaben werden dabei direkt aneinandergehangen. Anschließend wird der so extrahierte Normtitel der `CSMClass` als Beschreibung (`description`) zugeordnet.

4.3.6 Die Regel „merkmalbeschreibung“

Die Merkmalbeschreibungssätze beginnen immer mit einem `MERKMAL`-Token. Darauf folgen die Version, die Identifikationsnummer und die Kennung der verantwortlichen Stelle. Diese drei `ZAHL`en sind für die Verarbeitung nicht relevant und werden deshalb ignoriert. Danach müssen eine `ZAHL` und eine `KENNUNG` stehen, die den Variablen `status_` bzw. `kennung` zugewiesen werden.

Der Status gibt dabei Auskunft darüber, wie und wo das Merkmal festgelegt ist. Dabei wird unterschieden, ob sich die Definition des Merkmals in der aktuellen Datei befindet oder nicht (Status `0`). Im ersten Fall wird dabei noch einmal differenziert, ob das Merkmal durch einen Wertebereich (Status `3`), einen Merkmalalgorithmus (Status `2`), innerhalb der Datensätze am Ende der Datei (Status `1`) oder als Referenz auf eine andere Norm (Status `4`) definiert ist. Die `kennung` enthält das Kürzel, das für das Merkmal intern in Algorithmen, Wertebereichen und Ähnlichem verwendet wird. Die drei darauffolgenden Textangaben Benennung, Maßbuchstaben und Maßeinheit sind optional. Allerdings muss das `KOMMA` zur Trennung der Parameter zwingend vorhanden sein, selbst wenn die Angaben weggelassen werden. Die `benennung` enthält den vollständigen Namen des Merkmals, während die Maßbuchstaben, also die Abkürzung in Zeichnungen und Ähnlichem in `massbuchstaben` gespeichert wird. Die Variable `einheit` enthält die Maßeinheit, in der die Werte des Merkmals angegeben werden. Am Schluss der Datenzeile ist der Typ des Merkmals anzugeben. Dabei kann es sich entweder um ein Merkmal vom Typ „Text“ (`T`) oder um eine „Zahl“ (`Z`) handeln. Dieser Typ wird in der Variablen `type` aufgenommen. Alle Angaben werden durch ein `KOMMA` und wahlweise durch ein zusätzliches `NONEWLINE` getrennt (vgl. *CAD-Normteiledatei nach DIN* 1990, S.84ff.).

Wenn das Muster passt, werden die Daten verarbeitet. Dazu wird der Status in eine Integer-Variable umgewandelt. Merkmalbeschreibungen mit dem Status `4` werden nicht behandelt, da dazu weitere Datenzeilen interpretiert werden müssten, was noch nicht realisiert wurde. Außerdem kommen in den vorliegenden Dateien der DIN V 4001 keine Merkmale mit dem Status `4` vor. Diese Aussage wurde mit Hilfe des Shell-Kommandos `grep ^MM *.TAB | cut -d, -f6 | grep 4` überprüft. Dieses Kommando filtert zuerst alle Merkmalbeschreibungssätze aus allen Merkmaldateien (`grep ^MM *.TAB`), schneidet dann die sechste Spalte, welche den Status enthält,

heraus (`cut -d, -f6`) und filtert ein weiteres Mal nach Zeilen, welche die Zahl `4` beinhalten (`grep 4`). Das Kommando gibt nichts aus.

Wenn das Merkmal einen Status ungleich `4` hat, wird zunächst ein Objekt der Klasse `CSMTabfileCharacteristicDef` angelegt, welches die Daten der Merkmalbeschreibung aufnimmt. Dieses Objekt wird dann mit der `CSMClass`, welche die aktuelle Norm enthält, verknüpft. Danach werden die Eigenschaften `ID` (Kennung), `Name` (Benennung), `Unit` (Maßeinheit) sowie `DimensionalLetters` (Maßbuchstaben) gesetzt. Wenn das Token `type` den Text „T“ enthält, handelt es sich um ein Merkmal vom Typ „Text“, andernfalls ist es vom Typ „Zahl“. Die Typen werden dem Objekt als Referenz auf einen der beiden im Konstruktor erzeugten `CSMType`n zugewiesen. Zum Abschluss der Regel muss das Merkmal in die Felder `characteristicDefById` und `characteristicDefsInFile` eingetragen werden, Letzteres allerdings nur, wenn das Merkmal den Status `1` hat, da dieses Feld dazu benutzt wird, die Werte in den Merkmaldatensätzen den Merkmalen zuzuordnen.

4.3.7 Die Regel „merkmalalgorithmus“

Das Token `MA` leitet eine Datenzeile ein, die einen Merkmalalgorithmus enthält. Danach folgen ein Platzhalter vom Tokentyp `ZAHL`, die `KENNUNG` des Merkmals sowie eine `zusammengesetzte_zeichenkette`, welche den eigentlichen Algorithmus enthält (vgl. *CAD-Normteiledatei nach DIN* 1990, S.88f.).

Wenn das Muster passt, wird aus dem Feld `characteristicDefById` die Merkmalbeschreibung gesucht, die dem Merkmal entspricht, welches durch den Algorithmus errechnet werden soll. Wenn diese gefunden wurde, wird ein Objekt der Klasse `CSMTabfileAlgorithm` erzeugt, das den konvertierten Algorithmus aufnehmen und mit der Merkmalbeschreibung verknüpfen soll. Als Nächstes ist die Sprache zu setzen, in die der *FORTRAN77*-Algorithmus umgewandelt werden soll. Wie im Abschnitt 3.3.2 beschrieben, soll dabei *Tcl* zum Einsatz kommen.

Da die Textangaben in den Merkmaldateien keine Zeilenumbrüche enthalten dürfen, selbige in *FORTRAN77* aber an einigen Stellen vorgeschrieben sind, müssen diese ergänzt werden. Außerdem finden sich in den vorliegenden DIN V 4001-Dateien einige Unsauberheiten, die ebenfalls bereinigt werden müssen. Das *FORTRAN77*-Gleichheitszeichen `.EQ.` musste in zusätzliche Leerzeichen eingebettet werden. Darüber hinaus

musste die *FORTRAN77*-Grammatik (DRAGON 2007) angepasst werden, da einige Variablen als Kommentar, Label oder als das Schlüsselwort „form“ erkannt wurden. Es ließ sich nicht abschließend klären, ob dies Fehler in den Merkmaldateien oder in der *FORTRAN77*-Grammatik sind, aber ein Deaktivieren der Regeln für Kommentare und Labels sowie des Schlüsselwortes „form“ in der Grammatik löste das Problem. Diese *FORTRAN77*-Konstrukte werden in den Merkmalalgorithmen ohnehin nicht verwendet.

Nach den Anpassungen wird ein `Fortran77Lexer` initialisiert, dem der in einen `InputStream` umgewandelte Algorithmus übergeben wird. Nach dem Lexer wird ein `Fortran77Parser` erzeugt, dessen Konstruktor den Lexer benötigt. Da es sich bei dem Algorithmus nicht um ein vollständiges *FORTRAN77*-Programm handelt, muss die Parserfunktion `executableStatement()` und nicht, wie vorgesehen, die Funktion `program()` aufgerufen werden. Der Parser baut aus dem `TokenStream`, den der Lexer erzeugt, einen AST auf. Dieser AST wird der Funktion `statement()` des `Fortran77Treeparsers` übergeben, nachdem er durch den Konstruktor mit dem Feld `characteristicDefById` initialisiert wurde. Das Feld benötigt er, um Zugriffe auf andere Merkmale zu registrieren und diese als Abhängigkeiten zu speichern. Die Funktion `statement()` gibt den übersetzten Algorithmus als `String` zurück. Als Letztes wird dieser Algorithmus und die Abhängigkeiten in dem Objekt der Klasse `CSMTabfileAlgorithm` gespeichert. Wie der `Fortran77Treeparser` genau funktioniert, wird in Abschnitt 4.4 diskutiert.

4.3.8 Die Regel „merkmaldatensatz“

Ein Objekt der Klasse `CSMTabfileCharacteristicDataset` wird vor dem Prüfen des Musters angelegt und mit der `CSMClass` verknüpft. Außerdem wird ein Zähler initialisiert, der bei jedem `KOMMA` hochgezählt und zur Identifikation der richtigen Merkmalbeschreibung genutzt wird.

Die Datenzeile beginnt mit einem `DA`-Token, welches von drei `ZAHL`en gefolgt wird, die allerdings keine Bedeutung für die Benutzung der Norm haben. Danach folgen beliebig viele Sequenzen aus einem `KOMMA`, einem optionalen `NONEWLINE` sowie einer ebenfalls optionalen Angabe. Diese Angabe kann sowohl eine `ZAHL` als auch eine zusammengesetzte Zeichenkette sein (vgl. *CAD-Normteiledatei nach DIN* 1990, S.98f.).

In beiden Fällen wird ein Trägerobjekt für den Wert erzeugt, welches in einen Container vom Typ `CSMTabfileCharacteristicValCont` gepackt wird. Der Container enthält sowohl den Wert als auch eine Referenz auf das zugehörige Merkmal. Danach wird er dem `CSMTabfileCharacteristicDataset` zugeordnet. Diese Prozedur wiederholt sich für jeden in der Datenzeile verzeichneten Wert. Nachdem die Datenzeile abgearbeitet wurde, wird das `CSMTabfileCharacteristicDataset` an die `CSMClass` der Norm angehängt.

4.3.9 Die Regel „unbehandelt“

Bei dieser Regel handelt es sich um eine besondere Regel. Sie dient dazu, alle Zeilen abzufangen, die nicht vom Parser interpretiert werden. Dabei passt das Muster auf eine beliebige Anzahl aller Tokens, die im `TokenStream` vorkommen können mit Ausnahme des Zeilenumbruchs `NEWLINE` und dem Dateiende `EOF`. Hierbei wird der Text jedes einzelnen Tokens an die Zeichenkette `line` angehängt. Dadurch entsteht eine Zeichenfolge, aus der auf die Originalzeile geschlossen werden kann. Weil der Lexer Zeichen, die er nicht deuten kann, wie z. B. überflüssige Leerzeichen, einfach ignoriert, ist es nicht möglich, die Zeile komplett wiederherzustellen. Allerdings ist es möglich, aus den Tokens die Zeilennummer in der Originaldatei auszulesen. Da sich diese durch die Tokens `TEXTNEWLINE` und `NONEWLINE` ändert, wird lediglich die Zeilennummer des ersten Tokens verwendet. Dies wird erreicht, indem die Variable `lineIndex` mit `-1` initialisiert und vor jedem Schreiben auf diesen Wert geprüft wird. Am Ende der Zeile, also nachdem das Muster komplett durchlaufen wurde, wird die Zeile samt Zeilennummer ausgegeben, sofern dem Konstruktor die Option `printSkippedLines` mit `true` übergeben wurde und die Zeile nicht leer ist.

4.4 Treeparser für FORTRAN77

Der Treeparser interpretiert den AST, der vom `Fortran77Parser` von (DRAGON 2007) erzeugt wurde. Die Grammatik des `Fortran77Treeparser` ist im Quellcode A.9 zu finden. Am Anfang wird wieder das *Java*-Package gesetzt und einige

Java-Bibliotheken, die zur weiteren Verarbeitung notwendig sind, eingebunden. Danach wird die Klasse definiert und mit der Option `importVocab = Fortran77;` das Vokabular des `Fortran77Lexer` eingebunden.

Nach den Optionen werden per eingebundenem *Java*-Code zwei Objektvariable deklariert und einige Funktionen sowie der Konstruktor der Klasse definiert. Die Variable `characteristicDefById` wird dem Konstruktor übergeben und enthält alle Merkmalbeschreibungen, welche über deren `ID` indexiert sind, sodass auf diese schnell zugegriffen werden kann. Sie werden benötigt, um das Feld `dependencies` zu füllen. Dieses ist als Menge (`Set`) deklariert, da jede Abhängigkeit nur einmal darin vorkommen soll. Es wird im Konstruktor mit einer leeren Menge initialisiert. Die öffentliche Methode `getDependencies()` gibt die Abhängigkeiten des Algorithmus als `ArrayList` zurück, da diese vom Datenmodell als solche benötigt werden. Weitere Methoden sind `isString()` und `replaceVariable().isString()` testet, ob ein übergebener Ausdruck für *Tcl* eine Zeichenkette ist. Dazu prüft sie das erste Zeichen des übergebenen Strings, ob dieser ein Anführungszeichen (`"`) oder ein Dollarzeichen (`$`) ist. Wenn der übergebene String mit einem Anführungszeichen beginnt, handelt es sich auf jeden Fall um eine Zeichenkette. Ein Dollarzeichen kennzeichnet allerdings Variable allgemein. Daher muss gprüft werden, ob das Merkmal vom Typ String ist, wozu das Feld `characteristicDefById` benötigt wird. Dies ist notwendig, obwohl *Tcl* eigentlich nicht zwischen verschiedenen Datentypen unterscheidet. Allerdings dürfen Zeichenketten nicht mit `expr` ausgewertet werden, da dieser Befehl versucht, den Wert der Variablen als Zahl zu interpretieren. Dies kann zu einem Programmabbruch, im ungünstigsten Fall aber zu falschen Ergebnissen führen. Aus dem gleichen Grund müssen Zeichenketten in *Tcl* mit `string equal` verglichen werden.

Nach (*CAD-Normteiledatei nach DIN* 1990, S.74) können Werte von Merkmalen eingebunden werden, indem die Kennung des Merkmals eingeleitet durch ein Dollarzeichen (`$`) und abgeschlossen durch einen Punkt (`.`) in den Algorithmus eingebaut werden. Die eingebundenen Werte sind vor dem Interpretieren des Algorithmus zu ersetzen. Diese Festlegung zielt darauf ab, die komplizierten Stringoperationen in *FORTRAN77* zu vermeiden. In Skriptsprachen wie *Tcl* ist dies nicht mehr notwendig, sodass diese Ausdrücke durch einen normalen Aufruf einer Variablen ersetzt werden können. Diese Aufgabe übernimmt die Funktion `replaceVariables()`. Außerdem maskiert sie alle Dollarzeichen und aufgehenden eckigen Klammern (`[`), da diese

in *Tcl* eine spezielle Bedeutung haben. Nach diesen Vorarbeiten folgen die Regeln des `Fortran77Treeparser`. Alle Regeln werden vom *ANTLR*-Compiler in eine *Java*-Methode umgesetzt, wobei diese so geschrieben wurden, dass sie einen String als Rückgabewert haben. Die Regeln werden im Folgenden erläutert.

4.4.1 Die Regel „statement“

Diese Regel ist die initiale Regel des Treeparsers, d. h. sie wird als Erstes aufgerufen und bearbeitet den gesamten Algorithmus. Zuerst werden alle benötigten Variablen deklariert.

Wenn der Wurzelknoten des aktuellen AST vom Typ `LITERAL_if` ist, wird das erste Kind als `condition` und das zweite als `thenblock` interpretiert. `LITERAL_if` bezeichnet dabei die bedingte Verzweigung `if`, wobei `condition` die Bedingung enthält und `thenblock` die Befehle, die ausgeführt werden sollen, wenn die Bedingung zutrifft. Die Rückgabewerte der Regeln enthalten bereits konvertierten *Tcl*-Code, der am Ende der Regel zusammengesetzt werden muss. Danach folgt eine beliebige Anzahl an `elseif`-Blöcken. Da die Variable `elseif` nach jedem Block geschrieben wird, ist die Regel `elseif` so aufgebaut, dass deren Rückgabewerte einfach zusammengefügt werden können. Am Ende folgt noch ein optionaler `elseblock`, welcher Befehle enthält, die nur ausgeführt werden sollen, wenn alle der vorher aufgeführten Bedingungen fehlgeschlagen sind. Nachdem alle Regeln durchlaufen worden sind, wird der *Tcl*-Code kombiniert.

Falls der Wurzelknoten des AST ein Token vom Typ `ASSIGN` ist, handelt es sich um eine Zuweisung. In diesem Fall muss das erste Kind vom Typ `NAME` sein und den Namen einer Variablen beinhalten. Das zweite Kind kann wieder ein AST sein, der in der Regel `expr_statement` ausgewertet und umgewandelt wird. Auch hier werden am Ende die Teile `name` und `expr` zusammengesetzt.

4.4.2 Die Regel „expr_statement“

Diese Regel ist insofern besonders, als dass sie die Regel `expr` kapselt. Sie wertet das Ergebnis dieser Regel aus. Wenn das Ergebnis eine Zeichenkette oder eine Variable

vom Typ einer Zeichenkette ist, dann gibt sie das Ergebnis direkt zurück. Ansonsten wird der Ausdruck in eckigen Klammern der *Tcl*-Funktion `expr` übergeben. Das bewirkt zur Laufzeit des Skripts, dass der Ausdruck als Rechnung ausgewertet wird. Dadurch erreicht die Regel zwei Ziele: Zum einen wird verhindert, dass Zeichenketten versehentlich als mathematischer Ausdruck gewertet werden. Zum anderen werden so unnötige rekursive Aufrufe von `expr` vermieden.

4.4.3 Die Regel „thenblock“

Knoten vom Typ `THENBLOCK` haben als einziges Kind einen AST, der durch die Regel `statement` interpretiert werden soll. Hier können also u.a. weitere bedingte Verzweigungen enthalten sein. Das kann zu einer beliebig langen, aber endlichen Rekursion der Regeln ausgebaut werden. `thenblock` gibt das in *Tcl*-Code umgewandelte `statement` zurück.

4.4.4 Die Regel „elseif“

Ein `ELSEIF`-Knoten hat, analog zum Knoten vom Typ `LITERAL_if`, als erstes Kind eine Bedingung (`condition`) und als zweites Kind wieder einen `thenblock`. Der Rückgabewert ist so aufgebaut, dass mehrere `elseif`-Zweige aneinandergereiht werden können.

4.4.5 Die Regel „elseblock“

Der `elseblock` enthält wie der `thenblock` ein `statement`, welches von der Regel zurückgegeben wird.

4.4.6 Die Regel „condition“

Die Regel `condition` behandelt alle Bedingungen. Sämtliche Bedingungen haben zwei Operanden, die verglichen werden sollen.

Wenn der Wurzelknoten des aktuellen AST vom Typ `EQ` ist, handelt es sich um einen Test auf Gleichheit. Hier stellt sich das Problem, dass in *FORTRAN77* Zeichenketten und mathematische Ausdrücke mit demselben Operator auf Gleichheit getestet werden. Wenn mindestens einer der beiden Operanden mit Hilfe der Funktion `isString()` als Zeichenkette identifiziert wird, werden diese mit `string equal` verglichen, andernfalls mit ==.

Die Vergleichsoperatoren `LT` ($<$), `LE` ($\leq$), `NE` ($\neq$), `GT` ($>$) sowie `GE` ($\geq$) können nur auf mathematische Ausdrücke angewandt werden, sodass sich an dieser Stelle ein Test mit `isString()` erübrigt. Sie sind alle syntaktisch gleich und werden einfach durch ihr *Tcl*-Pendant ersetzt.

Die logischen Operatoren `LAND` und `LOR` haben selbst zwei Bedingungen als Kinder. Diese werden in Klammern gesetzt und mit `&&` bzw. `||` verknüpft.

4.4.7 Die Regel „expr“

Diese Regel behandelt sowohl mathematische Ausdrücke als auch konstante Zeichenketten.

Falls der Wurzelknoten vom Typ `NAME` ist, handelt es sich entweder um eine mathematische Funktion oder einen Zugriff auf eine Variable. Im ersten Fall hat der Knoten ein Kind, welches als `expr` interpretierbar ist. Dabei enthält das `NAME`-Token den Namen der Funktion, sodass sie durch ihr *Tcl*-Pendant ersetzt werden kann. Die meisten Funktionen haben in *Tcl* denselben Namen wie in *FORTRAN77*, mit dem Unterschied, dass sie in *Tcl* in Kleinbuchstaben geschrieben werden. Eine Ausnahme bildet die Funktion `AINT`, die eine Fließkommazahl auf ganze Zahlen abrundet, aber explizit eine Fließkommazahl zurückgibt. Sie wurde ersetzt durch einen Aufruf von `int` und einer Multiplikation mit `1.0`. Wenn eine Funktion benutzt wird, die noch nicht implementiert wurde, wird eine Warnung ausgegeben. Falls es sich bei dem Aufruf um eine Variable handelt, wird deren Name mit einem Dollarzeichen (`$`) vorangestellt zurückgegeben. Dies bewirkt zur Laufzeit des Skriptes einen Austausch mit dem Inhalt der Variablen. Zusätzlich wird das zur Variable gehörende Merkmal in die Liste der Abhängigkeiten aufgenommen.

Die Typen `ICON` und `RCON` stehen für konstante Integer- und Real-Zahlen. Es wird einfach der Text des jeweiligen Tokens zurückgegeben.

Stringkonstanten werden durch den Tokentyp `SCON` gekennzeichnet. Sie werden in Anführungszeichen (`"`) eingebettet. Durch die *Java*-Funktion `replaceVariables()` werden *Tcl*-Schlüsselzeichen maskiert und obsolete Zugriffe auf Stringvariablen durch einen normalen Variablenaufruf ersetzt.

Die Operatoren `PLUS`, `MINUS`, `STAR` und `DIV` haben alle zwei Ausdrücke als Kinder. Diese werden je nach Operator addiert, subtrahiert, multipliziert oder dividiert. Der Division ist eine Multiplikation mit `1.0` vorangestellt, um eine Ganzzahldivision zu vermeiden. Wenn das `MINUS` nur einen Operanden hat, so wirkt es als Negationszeichen. Die Rückgabewerte sind immer in runden Klammern gefasst, um Fehler durch die unterschiedlichen Gewichtungen der Rechenoperationen zu vermeiden.

Der letzte Operator ist der `POW`-Operator zur Berechnung einer Potenz. Er wird durch die Funktion `pow()` ersetzt.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung und Fazit

Im Rahmen der Arbeit wurde ein Weg zur Konvertierung der Merkmaldateien aufgezeigt. Es wurde ein Datenformat evaluiert, das einfach zu handhaben ist und sich problemlos in *Remarc* integrieren lässt, da die Datenstrukturen, die in *Remarc* bereits eingesetzt werden, um relevante Daten erweitert wurden. Außerdem wurde ein Prototyp zur Konvertierung der Daten entwickelt, welcher bereits die wichtigsten Daten der Merkmaldateien in das neue Format umwandelt.

Durch die parallel laufende Arbeit zur Integration der konvertierten Dateien in *Remarc* konnte gezeigt werden, dass die durch die Arbeitsweise der *BSV* verursachte Trägheit von *Remarc* erfolgreich reduziert wurde. Die in Abschnitt 1.2 angesprochenen Unterbrechungen des Arbeitsflusses sind subjektiv komplett verschwunden und die Zeit, die *Remarc* zum Anlegen der kompletten Normreihe DIN 58461 benötigt, wurde von ungefähr 69 Sekunden um den Faktor vier auf etwa 17 Sekunden gesenkt.

5.2 Ausblick

Wie im Abschnitt 1.2 beschrieben, ist es das langfristige Ziel, *BSV* zu ersetzen. Der Prototyp des Konverters der Merkmaldateien ist ein wichtiger Schritt in diese Richtung. Zur vollständigen Implementierung ist es notwendig, fehlende Einträge der Merkmaldateien, wie z. B. die Normbeschreibung zu parsen. Dies erfordert in den meisten Fällen

lediglich eine Erweiterung des Lexers sowie des Parsers. Eine Ausnahme bilden hier die Wertebereiche, da diese vermutlich auf Grund ihrer Komplexität analog zu den Merkmalalgorithmen eine eigene *ANTLR*-Grammatik benötigen.

Ein weiterer Punkt, der die *BSV* bisher unverzichtbar macht, ist die Interpretierung der Geometriedaten, welche allerdings nicht Teil dieser Arbeit waren.

Anhang A

Quelltexte

```
BD
ID,NN,'DIN ISO 14'
ID,NT,'Keilwellen-Verbindungen mit geraden Flanken und Innenzentrierung; ',
-     'Masze Toleranzen Pruefung; Identisch mit ISO 14 Ausgabe 1982'
ID,NB,'IF(EAA .EQ. ''Welle'') THEN',
- 'NB=''Keilwelle ISO 14 - $EAK.''',
-     'ELSE',
- 'NB=''Nut fuer Keilwelle ISO 14 - $EAK.''',
-     'ENDIF'
C
C ---------------------------------------------------------------------------
C Merkmalbeschreibungssaetze
C
ZA,MM,4
MM,GA, 1,  10, 1, 1, EAA, 'Ausfuehrung'                    ,     ,      , T
MM,GA, 1,  20, 1, 1, EAB, 'Reihe'                          ,     ,      , T
MM,GM, 1,  70, 1, 1, AAB, 'Auszendurchmesser'              , 'D' , 'mm' , Z
MM,GM, 1,  80, 1, 1, AAC, 'Keilbreite'                     , 'B' , 'mm' , Z
C
C ---------------------------------------------------------------------------
C Merkmalalgorithmus
C
ZA,MA,2
MA,1,DAA,'DAA = AAA-0.25*(AAB-AAA)'
MA,1,DAB,'DAB = 0.5*AAC'
C
C ---------------------------------------------------------------------------
C Funktionswertebereiche
C
ZA,FW,1
FW,1,CAA,0,(0:MAX]
C
C ---------------------------------------------------------------------------
C Merkmaldatensaetze
C
C  AE IDNR FA    EAA      EAB            EAK       AAA  AAB   AAC  AAD
ZA,DA,4
DA, 1,  10, 1, 'Welle', 'Leicht',    '6 x 23 x 26',  23,  26,  6.0,  6
DA, 1,  20, 1, 'Welle', 'Leicht',    '6 x 26 x 30',  26,  30,  6.0,  6
DA, 1,  30, 1, 'Welle', 'Leicht',    '6 x 28 x 32',  28,  32,  7.0,  6
DA, 1, 700, 1,  'Nabe',  'Mitte',
*                                   '10 x 112 x 125', 112, 125, 18.0, 10
C
C ---------------------------------------------------------------------------
ED
```

Quelltext A.1: (Gekürzte) Merkmaldatei der Norm „DIN ISO 14“

```
class ExampleLexer extends Lexer;
options {
  exportVocab=ExampleVocab;
  k=2;
}

NEWLINE:
  (
    "\r\n" //DOS
    | '\r' //MAC
    | '\n' //UNIX
  )
  { newline(); }
  ;

ZEICHENKETTE:
  (BUCHSTABE)+
  ;

ZAHL:
  (ZIFFER)+
  ;

KOMMA:
  ','
  ;

protected BUCHSTABE:
  (
    'a'..'z'
    | 'A'..'Z'
  )
  ;

protected ZIFFER:
  ('0'..'9')
  ;
```

Quelltext A.2: Der ExampleLexer

```
class ExampleParser extends Parser;
options {
  importVocab=ExampleVocab;
    k=1;
}

file:
  (
    line
    NEWLINE
  )*
  EOF
  ;

line:
  (name:ZEICHENKETTE) KOMMA (vorname:ZEICHENKETTE) KOMMA (alter:ZAHL)
  {
    System.out.println(vorname.getText() + " " + name.getText() + " ist " + alter.
        getText() + " Jahre alt.");
  }
  ;
```

Quelltext A.3: Der ExampleParser

```
Mueller,Hans,35
Schmidt,Erika,40
Schulze,Markus,23
```

Quelltext A.4: Eine Beispieldatei für ExampleLexer und ExampleParser

```
Hans Mueller ist 35 Jahre alt.
Erika Schmidt ist 40 Jahre alt.
Markus Schulze ist 23 Jahre alt.
```

Quelltext A.5: Ausgabe des ExampleParsers

```
class ExpressionLexer extends Lexer;

PLUS  : '+' ;
MINUS : '-' ;
MUL   : '*' ;
DIV   : '/' ;
MOD   : '%' ;
POW   : '^' ;
SEMI  : ';' ;
protected DIGIT : '0'..'9' ;
INT   : (DIGIT)+ ;

class ExpressionParser extends Parser;
options { buildAST=true; }

expr     : sumExpr SEMI!;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : atom (POW^ atom)? ;
atom     : INT ;

{import java.lang.Math;}
class ExpressionTreeWalker extends TreeParser;

expr returns [double r]
  { double a,b; r=0; }

  : #(PLUS  a=expr b=expr)  { r=a+b; }
  | #(MINUS a=expr b=expr)  { r=a-b; }
  | #(MUL   a=expr b=expr)  { r=a*b; }
  | #(DIV   a=expr b=expr)  { r=a/b; }
  | #(MOD   a=expr b=expr)  { r=a%b; }
  | #(POW   a=expr b=expr)  { r=Math.pow(a,b); }
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
  ;
```

Quelltext A.6: Grammatik zur Auswertung einfacher mathematischer Ausdrücke (MILLS 2005)

```
header {
    package de.htwm.kgraefe.tabfileconverter.antlr;
}

class TABFileLexer extends Lexer;
options {
    charVocabulary='\3'..'\377';
    filter=true;
    k=5;
}

NEWLINE:
```

```
13      (
14          (
15              "\r\n"   //DOS
16              | '\r'                    //MAC
17              | '\n'                    //UNIX
18          )
19          (
20              (STERN {
21                  $setType(NONEWLINE);
22              })
23              | (MINUS{
24                  $setType(TEXTNEWLINE);
25              })
26          )?
27      )
28      { newline(); }
29      ;
30
31  KOMMENTAR:
32      {getColumn() == 1}?
33      'C' (~('\r'|'\n'))*
34      ;
35
36  IDNN:
37      {getColumn() == 1}?
38      "ID,NN"
39      ;
40
41  IDNT:
42      {getColumn() == 1}?
43      "ID,NT"
44      ;
45
46  MERKMAL:
47      {getColumn() == 1}?
48      (
49      "MM,GA"
50      | "MM,SM"
51      | "MM,GM"
52      | "MM,EM"
53      | "MM,AT"
54      | "MM,FM"
55    )
56      ;
57
58  DA:
59      {getColumn() == 1}?
60      "DA,"
61      ;
62
63  MA:
64      {getColumn() == 1}?
65      "MA,"
66      ;
67
68  ZEICHENKETTE:
69      (APOSTROPH (ZEICHEN|GESCHUETZTER_APOSTROPH)* APOSTROPH)
70      {
71          /* Apostrophe abschneiden */
72          setText(getText().substring(1, getText().length() - 1));
73
74          /* doppelte (maskierte) Apostrophe zusammen fuehren */
75          setText(getText().replace("''", "'"));
76      }
77      ;
78
```

```
ZAHL:
  (
    (PLUS_MINUS)? (ZIFFER)+ (PUNKT (ZIFFER)+)?
  )
    ;

KENNUNG:
    BUCHSTABE (BUCHSTABE|ZIFFER)*
    ;

KOMMA:
    ','
    ;

STERN:
    '*'
    ;

protected ZEICHEN:
    (BUCHSTABE|ZIFFER|WS|KOMMA|PUNKT|PLUS_MINUS|SEMIKOLON|GLEICH|STERN|DIV|KLAMAUF|
        KLAMZU|ECKKLAMAUF|ECKKLAMZU|DOLLAR|DOPPELPUNKT|PROZENT)
    ;

protected BUCHSTABE:
    'A'..'Z'|'a'..'z'
    ;

protected ZIFFER:
    '0'..'9'
    ;

protected APOSTROPH:
    '\''
    ;

protected PUNKT:
    '.'
    ;

protected DOPPELPUNKT:
    ':'
    ;

protected SEMIKOLON:
    ';'
    ;

protected PLUS_MINUS:
    '+'|'-'
    ;

protected MINUS:
    '-'
    ;

protected DIV:
    '/'
    ;

protected GLEICH:
    '='
    ;

protected KLAMAUF:
    '('
    ;
```

```
144
145 protected KLAMZU:
146     ')'
147     ;
148
149 protected DOLLAR:
150     '$'
151     ;
152
153 protected PROZENT:
154     '%'
155     ;
156
157 protected GESCHUETZTER_APOSTROPH:
158     '\'' '\''
159     ;
160
161 protected ECKKLAMAUF:
162     '['
163     ;
164
165 protected ECKKLAMZU:
166     ']'
167     ;
168
169 protected WS:
170     (' '|'\t')
171     ;
```

Quelltext A.7: Lexer für die DIN V 4001-Dateien

```
1  header {
2      package de.htwm.kgraefe.tabfileconverter.antlr;
3
4      import de.arcsolutions.remarc.platform.datacore.*;
5      import de.arcsolutions.remarc.platform.datacore.types.*;
6      import de.arcsolutions.remarc.platform.datacore.types.impl.TypesFactoryImpl;
7      import de.arcsolutions.remarc.product.standard.standardclass.
           StandardEmfCoreFactory;
8      import de.arcsolutions.remarc.product.standard.standardclass.
           StandardEmfCorePackage;
9      import de.htwm.kgraefe.tabfileconverter.tabfile.*;
10     import java.io.*;
11     import java.util.*;
12 }
13
14 class TABFileParser extends Parser;
15 options {
16     k=3;
17 }
18
19 {
20   CSMXML csmXml = null;
21   CSMTabfileClass csmClass = null;
22   ArrayList<CSMTabfileCharacteristicDef> characteristicDefsInFile = null;
23   Map<String, CSMTabfileCharacteristicDef> characteristicDefById = null;
24   String zeichenkette = null;
25   boolean printSkippedLines;
26
27   CSMType csmStringType = null;
28   CSMType csmDoubleType = null;
29
30
31     public TABFileParser(TABFileLexer lexer, boolean printSkippedLines) {
32         this(lexer,1);
33
```

```
        /* Strings und (allgemeine) Zahlen moeglich */
        csmStringType = new TypesFactoryImpl().createStringType();
        csmDoubleType = new TypesFactoryImpl().createDoubleType();

        /*** Datenstruktur aufbauen ***/
        csmXml = DatacoreFactory.eINSTANCE.createCSMXML();
        csmXml.getTypes().add(csmStringType);
        csmXml.getTypes().add(csmDoubleType);

        csmClass = TabfileFactory.eINSTANCE.createCSMTabfileClass();
        csmClass.setCsmXML(csmXml);

        characteristicDefsInFile = new ArrayList<CSMTabfileCharacteristicDef>();

        this.printSkippedLines = printSkippedLines;

        characteristicDefById = new HashMap<String, CSMTabfileCharacteristicDef>();
    }

    public CSMXML getCsmXml() {
        return csmXml;
    }
}


file:
    line
    (options { greedy=true; } :
        NEWLINE line
    )*
    (
        NEWLINE
    )?
    EOF
    ;

line:
    KOMMENTAR
    | normnummer
    | normtitel
    | merkmalbeschreibung
    | merkmalalgorithmus
    | merkmaldatensatz
    | unbehandelt
    ;

normnummer:
    {
    String nummer=null;
  }
    (
        IDNN
        KOMMA
        (zusammengesetzte_zeichenkette {
            nummer=zeichenkette;
        })
        (
            KOMMA
            zusammengesetzte_zeichenkette
        )*
    )
    {
       csmClass.setID(nummer);
    }
    ;

```

```
100 normtitel:
101     {
102         String normtitel = null;
103     }
104     (
105         IDNT
106         KOMMA
107         (zusammengesetzte_zeichenkette {
108             normtitel=zeichenkette;
109         })
110         (
111             KOMMA
112             (NONEWLINE)?
113             zusammengesetzte_zeichenkette {
114                 normtitel+=zeichenkette;
115             }
116         )*
117     )
118     {
119         csmClass.setDescription(normtitel);
120     }
121     ;
122
123 merkmalbeschreibung:
124     {
125         String benennung = "";
126         String massbuchstaben = "";
127         String einheit = "";
128     }
129     (
130         MERKMAL
131         KOMMA
132         ZAHL
133         KOMMA
134         ZAHL
135         KOMMA
136         ZAHL
137         KOMMA
138         (status_:ZAHL)
139         KOMMA
140         (kennung:KENNUNG)
141         KOMMA
142         (NONEWLINE)?
143         (
144             zusammengesetzte_zeichenkette {
145                 benennung=zeichenkette;
146             }
147             (NONEWLINE)?
148         )?
149         KOMMA
150         (NONEWLINE)?
151         (zusammengesetzte_zeichenkette {
152             massbuchstaben=zeichenkette;
153         })?
154         KOMMA
155         (zusammengesetzte_zeichenkette {
156             einheit=zeichenkette;
157         })?
158         KOMMA
159         (type:KENNUNG)
160     )
161     {
162         /*
163          * status:
164          *  0: Daten nicht in der Datei
165          *  1: Daten sind in der Datei
```

```
166          *  2: Daten als Konstante den Datensaetzen vorangestellt oder Daten als
                 Algorithmus in Abhaengigkeit von anderen Tabellenwerten definiert
167          *  3: Merkmal im Wertebereich
168          *  4: Referenz auf andere Norm
169          */
170         int status = Integer.parseInt(status_.getText());
171
172     if(status != 4) {
173       CSMTabfileCharacteristicDef csmTabfileCharacteristicDef = TabfileFactory.
              eINSTANCE.createCSMTabfileCharacteristicDef();
174       csmTabfileCharacteristicDef.setCsmClass(csmClass);
175
176       csmTabfileCharacteristicDef.setID(kennung.getText());
177       csmTabfileCharacteristicDef.setName(benennung);
178       csmTabfileCharacteristicDef.setUnit(einheit);
179       csmTabfileCharacteristicDef.setDimensionalLetters(massbuchstaben);
180       if(type.getText().equals("T")) {
181         csmTabfileCharacteristicDef.setDataType(csmStringType);
182       } else {
183         csmTabfileCharacteristicDef.setDataType(csmDoubleType);
184       }
185
186       characteristicDefById.put(kennung.getText(), csmTabfileCharacteristicDef);
187
188       if(status == 1) {
189         /* in die Liste der characteristicDefsInFile einfuegen, um spaeter Datasets
                aufbauen zu koennen */
190         characteristicDefsInFile.add(csmTabfileCharacteristicDef);
191       }
192     }
193     }
194     ;
195
196 merkmalalgorithmus:
197     {
198         String algorithmus = null;
199     }
200     (
201         MA
202         ZAHL
203         KOMMA
204         (kennung:KENNUNG)
205         KOMMA
206         zusammengesetzte_zeichenkette {
207             algorithmus = zeichenkette;
208         }
209     )
210     {
211         CSMTabfileCharacteristicDef csmTabfileCharacteristicDef =
                characteristicDefById.get(kennung.getText());
212     if(csmTabfileCharacteristicDef != null) {
213       /* Algorithmus-Struktur erstellen und mit CharacteristicDefinition verknuepfen
             */
214       CSMTabfileAlgorithm csmTabfileAlgorithm = TabfileFactory.eINSTANCE.
             createCSMTabfileAlgorithm();
215       csmTabfileAlgorithm.setCsmTabfileCharacteristicDef(csmTabfileCharacteristicDef);
216
217       csmTabfileAlgorithm.setLanguage("tcl");
218
219       /* einige Unsauberheiten sowohl in den TAB-Files als auch im Fortran77-Lexer
             bereinigen */
220       algorithmus = algorithmus.replace("THEN ", "THEN\n");
221       algorithmus = algorithmus.replace(" ELSE ", "\nELSE\n");
222       algorithmus = algorithmus.replace(" ENDIF", "\nENDIF");
223       algorithmus = algorithmus.replace(" ELSEIF", "\nELSEIF");
224       algorithmus = algorithmus.replace(".EQ.", " .EQ. ");
```

```
225       /* zusaetzlich wurden in der FORTRAN77-Grammatik Kommentare, Label und das
              Schluesselwort "form" deaktiviert */
226
227       InputStream bais = new ByteArrayInputStream(algorithmus.getBytes());
228       Fortran77Lexer lexer = new Fortran77Lexer(bais);
229
230       Fortran77Parser parser = new Fortran77Parser(lexer);
231       parser.executableStatement();
232
233       Fortran77Treeparser treewalker = new Fortran77Treeparser(characteristicDefById);
234       String tclAlgorithm = treewalker.statement(parser.getAST());
235
236       csmTabfileAlgorithm.setAlgorithm(tclAlgorithm);
237       csmTabfileAlgorithm.getDependencies().addAll(treewalker.getDependencies());
238     }
239     }
240     ;
241
242 merkmaldatensatz:
243     {
244         CSMTabfileCharacteristicDataset characteristicDataset = TabfileFactory.
                eINSTANCE.createCSMTabfileCharacteristicDataset();
245         characteristicDataset.setCsmClass(csmClass);
246
247         int datasetCounter = -1; /* wird vor der ersten Verwendung inkrementiert */
248     }
249     (
250         DA
251         (version:ZAHL)
252         KOMMA
253         (id:ZAHL)
254         KOMMA
255         (verantwortliche_stelle:ZAHL)
256         (options { greedy=true; } :
257             KOMMA {
258                 datasetCounter++;
259             }
260             (NONEWLINE)?
261             (
262         (zusammengesetzte_zeichenkette {
263           StringValue value = TypesFactory.eINSTANCE.createStringValue();
264           value.setValue(zeichenkette);
265
266           CSMTabfileCharacteristicValCont valueCont = TabfileFactory.eINSTANCE.
                createCSMTabfileCharacteristicValCont();
267           valueCont.setCharacteristicDataset(characteristicDataset);
268           valueCont.setDataType(characteristicDefsInFile.get(datasetCounter));
269
270           valueCont.setConcreteValue(value);
271           valueCont.setValid(true);
272
273           characteristicDataset.getValues().add(valueCont);
274         })
275         |(z:ZAHL {
276           String zahl = z.getText();
277
278           StringValue value = TypesFactory.eINSTANCE.createStringValue();
279           value.setValue(zahl);
280
281           CSMTabfileCharacteristicValCont valueCont = TabfileFactory.eINSTANCE.
                createCSMTabfileCharacteristicValCont();
282           valueCont.setCharacteristicDataset(characteristicDataset);
283           valueCont.setDataType(characteristicDefsInFile.get(datasetCounter));
284
285           valueCont.setConcreteValue(value);
286           valueCont.setValid(true);
```

```
287
288           characteristicDataset.getValues().add(valueCont);
289         })
290       )?
291         )*
292     )
293     {
294         csmClass.getCsmCharacteristicDatasets().add(characteristicDataset);
295     }
296     ;
297
298 zusammengesetzte_zeichenkette:
299     (
300     (
301       z:ZEICHENKETTE {
302         zeichenkette = z.getText();
303       }
304     )
305     (options { greedy=true; } :
306       (KOMMA TEXTNEWLINE s:ZEICHENKETTE) {
307         zeichenkette += ' ' + s.getText();
308       }
309     )*
310     )
311     ;
312
313 unbehandelt:
314     {
315         String line = "";
316         int lineIndex = -1;
317     }
318     (
319         stern:STERN {
320             line += stern.getText();
321             if(lineIndex == -1) lineIndex = stern.getLine();
322         }
323         |zeichenkette:ZEICHENKETTE {
324             line += zeichenkette.getText();
325             if(lineIndex == -1) lineIndex = zeichenkette.getLine();
326         }
327         |zahl:ZAHL {
328             line += zahl.getText();
329             if(lineIndex == -1) lineIndex = zahl.getLine();
330         }
331         |komma:KOMMA {
332             line += komma.getText();
333             if(lineIndex == -1) lineIndex = komma.getLine();
334         }
335         |kennung:KENNUNG {
336             line += kennung.getText();
337             if(lineIndex == -1) lineIndex = kennung.getLine();
338         }
339         |nonewline:NONEWLINE {
340             line += nonewline.getText();
341             if(lineIndex == -1) lineIndex = nonewline.getLine();
342         }
343         |textnewline:TEXTNEWLINE {
344             line += textnewline.getText();
345             if(lineIndex == -1) lineIndex = textnewline.getLine();
346         }
347     )*
348     {
349         if(printSkippedLines && !line.equals("")) {
350             System.out.println("Skipped line " + lineIndex + ": " + line);
351         }
352     }
```

```
353     ;
```

Quelltext A.8: Parser für die DIN V 4001-Dateien

```
1  header {
2      package de.htwm.kgraefe.tabfileconverter.antlr;
3
4      import java.util.*;
5      import de.htwm.kgraefe.tabfileconverter.tabfile.*;
6  }
7
8  class Fortran77Treeparser extends TreeParser;
9  options {
10     importVocab = Fortran77;
11 }
12
13 {
14     Map<String, CSMTabfileCharacteristicDef> characteristicDefById = null;
15     Set<CSMTabfileCharacteristicDef> dependencies = null;
16
17     public Fortran77Treeparser(Map<String, CSMTabfileCharacteristicDef>
           characteristicDefById) {
18         this.characteristicDefById = characteristicDefById;
19         dependencies = new HashSet<CSMTabfileCharacteristicDef>();
20     }
21
22     public ArrayList<CSMTabfileCharacteristicDef> getDependencies() {
23         ArrayList<CSMTabfileCharacteristicDef> ret = new ArrayList<
               CSMTabfileCharacteristicDef>();
24         ret.addAll(dependencies);
25         return ret;
26     }
27
28     private boolean isString(String s) {
29         boolean ret = false;
30
31         if(s.length() > 0) {
32       if(s.charAt(0) == '$' && characteristicDefById != null) {
33         CSMTabfileCharacteristicDef csmTabfileCharacteristicDef =
               characteristicDefById.get(s.substring(1));
34         if(csmTabfileCharacteristicDef != null) {
35           ret = (csmTabfileCharacteristicDef.getDataType() instanceof de.arcsolutions.
                 remarc.platform.datacore.types.StringType);
36         }
37       } else if(s.charAt(0) == '"') {
38         ret = true;
39       }
40     }
41
42         return ret;
43     }
44
45     private String replaceVariables(String s) {
46     /* alle $ und [ muessen maskiert werden */
47     s = s.replace("$", "\\$");
48     s = s.replace("[", "\\[");
49
50         /* Ausdruecke der Form \$xyz. muessen ersetzt werden durch ${xyz} */
51         boolean changed = true;
52         int start=0, posDollar, posDot;
53
54         while(changed) {
55             changed = false;
56
57             posDollar = s.indexOf("\\$", start);
58
```

```
59              if(posDollar >= start) {
60                  start = posDollar + 2;
61                  posDot = s.indexOf(".", start);
62
63                  if(posDot > start) {
64                      String var = s.substring(posDollar + 2, posDot);
65
66                      s = s.substring(0, posDollar) + "${" + var + "}" + s.substring(
                            posDot + 1);
67
68                      start = posDollar + var.length() + 3;
69
70                      changed = true;
71                  }
72              }
73          }
74
75          return s;
76      }
77  }
78
79  statement returns [String ret]
80      {
81          String condition, thenblock, elseif, elseif_gesamt="", elseblock=null, expr;
82          ret=null;
83      }
84      :#(
85          LITERAL_if
86          condition=condition
87          thenblock=thenblock
88          (
89              elseif=elseif {
90                  elseif_gesamt += elseif;
91              }
92          )*
93          (elseblock=elseblock)?
94      ){
95          ret = "if {" + condition + "} {\n" + thenblock + "\n}" + elseif_gesamt;
96          if(elseblock != null) {
97              ret += " else {\n " + elseblock + " \n}";
98          }
99      }
100     |#(ASSIGN name:NAME expr=expr_statement) {
101         ret = "set " + name.getText() + " " + expr;
102     }
103     ;
104
105 expr_statement returns [String ret]
106     {
107         ret= null;
108         String expr = null;
109     }
110     :expr=expr {
111         if(isString(expr)) {
112             ret = expr;
113         } else {
114             ret = "[expr " + expr + "]";
115         }
116     }
117     ;
118
119 thenblock returns [String ret]
120     {
121     String s;
122     ret=null;
123     }
```

```
124     :#(THENBLOCK s=statement) {
125         ret = s;
126     }
127     ;
128 
129 elseif returns [String ret]
130     {
131         String condition, thenblock, elseblock=null;
132         ret = null;
133     }
134     :#(
135         ELSEIF
136         condition=condition
137         thenblock=thenblock
138     ){
139         ret = " elseif {" + condition + "} {\n" + thenblock + "\n}";
140     }
141     ;
142 
143 elseblock returns [String ret]
144     {
145         String s;
146         ret=null;
147     }
148     :#(ELSEBLOCK s=statement) {
149         ret = s;
150     }
151     ;
152 
153 condition returns [String ret]
154     {
155         String a, b;
156         ret=null;
157     }
158     :#(EQ a=expr b=expr) {
159         /*
160          * Problem: Stringvergleiche werden in FORTRAN77 mit
161          * demselben Operator ausgefuehrt, muessen aber unterschieden werden
162          *
163          * Loesung: Stringkonstanten beginnen mit "\"", Variablen mit "$"
164          * (Variablen muessen genauer untersucht werden)
165          */
166 
167         if(isString(a) || isString(b)) {
168             ret = "[string equal " + a + " " + b + "]";
169         } else {
170             ret = a  + " == " + b;
171         }
172     }
173     |#(LT a=expr b=expr) {
174         ret = a  + " < " + b;
175     }
176     |#(LE a=expr b=expr) {
177         ret = a  + " <= " + b;
178     }
179     |#(NE a=expr b=expr) {
180         ret = a  + " != " + b;
181     }
182     |#(GT a=expr b=expr) {
183         ret = a  + " > " + b;
184     }
185     |#(GE a=expr b=expr) {
186         ret = a  + " >= " + b;
187     }
188     |#(LAND a=condition b=condition) {
189         ret = "("+ a + ") && (" + b +")";
```

```
190     }
191     |#(LOR a=condition b=condition) {
192         ret = "("+ a + ") || (" + b +")";
193     }
194     ;
195
196
197
198 expr returns [String ret]
199     {
200         ret=null;
201         String a=null, b=null;
202     }
203     :#(n:NAME (a=expr)?) {
204         if(a != null) {
205             if(n.getText().equals("SQRT")) {
206                 ret = "sqrt(" + a + ")";
207             } else if(n.getText().equals("ATAN")) {
208                 ret = "atan(" + a + ")";
209             } else if(n.getText().equals("TAN")) {
210                 ret = "tan(" + a + ")";
211             } else if(n.getText().equals("COS")) {
212                 ret = "cos(" + a + ")";
213             } else if(n.getText().equals("SIN")) {
214                 ret = "sin(" + a + ")";
215             } else if(n.getText().equals("INT")) {
216                 ret = "int(" + a + ")";
217             } else if(n.getText().equals("AINT")) {
218                 ret = "(1.0 * int(" + a + "))";
219             } else if(n.getText().equals("ABS")) {
220                 ret = "abs(" + a + ")";
221             } else {
222                 System.err.println("Math function not implemented: " + n.getText());
223             }
224         } else {
225             ret = "$" + n.getText();
226             dependencies.add(characteristicDefById.get(n.getText()));
227         }
228     }
229     |i:ICON { /* integer constant */
230         ret = i.getText();
231     }
232     |y:RCON { /* real constant */
233         ret = y.getText();
234     }
235     |s:SCON { /* string constant */
236         ret = "\"" + replaceVariables(s.getText()) + "\"";
237     }
238     |#(MINUS a=expr (b=expr)?) {
239         if(b != null) {
240             ret = "(" + a + " - " + b + ")";
241         } else {
242             ret = "(-" + a + ")";
243         }
244     }
245     |#(PLUS a=expr b=expr) {
246         ret = "(" + a + " + " + b + ")";
247     }
248     |#(STAR a=expr b=expr) {
249         ret = "(" + a + " * " + b + ")";
250     }
251     |#(DIV a=expr b=expr) {
252         ret = "(1.0 * " + a + " / " + b + ")";
253     }
254     |#(POWER a=expr b=expr) {
255         ret = "(pow(" + a + ", " + b + "))";
```

```
256         }
257         ;
```

Quelltext A.9: Treeparser für die FORTRAN77-Algorithmen

Glossar

Datensatz
Ein Datensatz ist eine Menge von Werten verschiedener Parameter eines Bauteils.

Datenzeile
Eine Datenzeile umfasst sämtliche Parameter eines Eintrags in einer Datei nach DIN V 4001. Sie endet nur dann am nächsten Zeilenumbruch, wenn die nächste Zeile weder mit einem „-“ noch mit einem „*“ beginnt.

DIN V 4001
Die DIN V 4001 enthält genormte CAD-Teile, welche nach der Vorschrift (*CAD-Normteiledatei nach DIN* 1990) erstellt wurden.

FORTRAN77
FORTRAN ist eine der ersten höheren Programmiersprachen. 1977 wurde durch das American National Standards Institute (ANSI) mit FORTRAN77 ein Standard etabliert, der die zahlreichen entstandenen Dialekte zusammenführen sollte. Dieses FORTRAN77 wurde später von der International Standards Organisation (ISO) als Standard übernommen. (vgl. PAGE 2005)

Grammatik
Eine *ANTLR*-Grammatik enthält *ANTLR*-Regeln, die als Grundlage für die Generierung von Klassen zur Interpretation der Eingangsdaten verwendet werden.

Makefile
Ein Makefile ist eine Art Skript zum Übersetzen von Programmen aus dem Quellcode. Dabei kann der Vorgang in kleine Einzelschritte zerlegt werden, sodass immer nur der Teil einer Anwendung übersetzt wird, der sich tatsächlich geändert hat.

Merkmaldaten
Merkmaldaten beinhalten sowohl grundsätzliche Daten normierter DIN-Teile, wie z. B. den Namen der Norm, als auch Beschreibungen wichtiger Merkmale mit Name, Datentyp und, falls notwendig, gültige Intervalle oder einen Algorithmus zur Berechnung der Werte. Die Merkmaldateien können außerdem auch

Merkmaldatensätze vorgeben, die eine Auswahl an normierten Teilen mit konkreten Werten darstellen.

Normreihe
Eine Normreihe enthält mehrere Bauteile derselben Norm, welche sich in den Werten der Parameter unterscheiden.

Normteil
Ein Normteil ist ein durch eine Norm in allen Parametern festgelegtes technisches Bauteil.

Sather
Sather ist eine objektorientierte Programmiersprache. (KLAWITTER 1999)

Shell
Die Shell ist eine textorientierte Kommandozeile.

Literatur

ANTLR Reference Manual. 2005. URL: http://antlr2.org/doc/index.html (besucht am 08. 09. 2009).

CAD-Normteiledatei nach DIN. DIN-Fachbericht 14. 1990.

DRAGON, Olivier: *FORTRAN77-Grammatik*. ANTLRv2-Grammatikdatei. Juni 2007. URL: http://antlr2.org/grammar/1183077163756/f77-antlr2.g (besucht am 08. 09. 2009).

KLAWITTER, Holger: *Sather Homepage*. Okt. 1999. URL: http://www.icsi.berkeley.edu/~sather/ (besucht am 14. 09. 2009).

MILLS, Ashley J.S: *ANTLR Tutorial*. The University Of Birmingham, UK. 2005. URL: http://supportweb.cs.bham.ac.uk/docs/tutorials/docsystem/build/tutorials/antlr/antlr.html (besucht am 08. 09. 2009).

PAGE, Clive G.: *Professional Programmer's Guide to Fortran77*. The University of Leicester, UK. 2005. URL: http://www.star.le.ac.uk/~cgp/prof77.html (besucht am 08. 09. 2009).

PARR, Terence: *ANTLR*. Softwareprogramm. Nov. 2006. URL: http://www.antlr2.org (besucht am 08. 09. 2009).

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, den 8. Oktober 2009

_____

Konrad Gräfe