

VHDL-Komplexitätsanalyse

Benjamin Menhorn und Frank Slomka | 19. Dezember 2015
Intitut für Eingebettete Systeme/Echtzeitsysteme, Universität Ulm

Kurzzusammenfassung: Diese Arbeit stellt ein Werkzeug vor, um die Komplexität und die Entwurfsgröße eines in VHDL gegebenen Entwurfs automatisiert berechnen zu können. Als Metrik hierfür wird die Entwurfsentropie eingesetzt, welche auf der Nachrichtentechnik beruht.

1 Einleitung

Bei der Planung, dem Entwurf und der Bewertung von Projekten ist es von zentraler Bedeutung, quantitative Aussagen über das Produkt treffen zu können. Im Bereich der Informatik und speziell im Bereich der digitalen Hardware beziehungsweise der integrierten Schaltungen bedeutet dies, die adäquate Messung der Quantität (Größe) und der Komplexität eines Entwurfes. Für heutige Entwicklungen ist es dabei nicht mehr ausreichend, Transistoren zu zählen oder das Gewicht eines Entwurfes zu bestimmen.

Anhand dieser Motivation wurde das Konzept der Entwurfsentropie vorgestellt [16] [17]. Es erlaubt die Berechnung der Entwurfsgröße und der Entwurfskomplexität anhand der Entwurfsentropie [17]. Die Berechnung basiert auf dem Ansatz, dass Komponenten eines Systems, wie beispielsweise Entitäten, miteinander über Verbindungen Informationen austauschen [17]. Dieser Informationsaustausch muss während des Entwurfes so realisiert werden, dass der Entwurf die vorgesehenen Aufgaben erfüllen kann. Durch die Messung des Informationsaustausches und den hier erforderlichen Kommunikationskanälen kann die in einem Entwurf liegende Komplexität und Entwurfsgröße bestimmt werden [17].

Grundsätzlich ist das Konzept der Entwurfsentropie abstrakt formuliert und wird erst durch den Anwendungsbezug konkretisiert. Dabei erfolgt innerhalb dieser Arbeit die Anwendung auf die digitale Hardware. Als Basis für den Entwurf digitaler Hardware wird hierzu eine Hardwarebeschreibungssprache (Hardware Description Language – HDL) gewählt. Das hier vorgestellte Analysewerkzeug erlaubt die automatische Berechnung der Entwurfsentropie eines Hardwareentwurfes.

2 Hardwarebeschreibungssprache

Hardwarebeschreibungssprachen sind Computersprachen, welche speziell entwickelt wurden, um die Struktur, Konstruktion und den Betrieb elektrischer Schaltungen und insbesondere digitaler Logikschaltungen zu beschreiben [14, S. 244]. Gegenüber Softwaresprachen sind die Hauptmerkmale einer Hardwarebeschreibungssprache die Beschreibungsmöglichkeit der Nebenläufigkeit von Hardwarekomponenten sowie die Modellierung des Zeitverhaltens [15, S. 65]. Hardwarebeschreibungssprachen eignen sich insbesondere für eine Beschreibung auf der algorithmischen, der Gatter- und der Registertransferebene [14, S. 243]. Sie ermöglichen die Synthese, um die Hardwarebeschreibung in der geplanten Zieltechnologie realisieren zu können [14, S. 244].

Daneben existieren noch weitere Werkzeuge und Sprachen, mit welchen ein Entwurf auf höheren Ebenen spezifiziert werden kann, die sogenannten Systembeschreibungssprachen [14, S. 243]. Mit SystemC lassen sich Entwürfe auch auf Systemebene beschreiben [15, S. 87] und es gibt zunächst keine Trennung zwischen Hardware und Software [14, S. 277]. Für die Beschreibung auf einer sehr hohen Abstraktionsebene kann MATLAB/Simulink eingesetzt werden, mit welchem die Systemanforderungen und die Signalverarbeitungsalgorithmen beschrieben werden können [3, S. 15]. Dabei werden Hochsprachensynthesewerkzeuge (High-Level Synthesis – HLS) eingesetzt, um eine RTL-Beschreibung (Register Transfer Level – RTL) von dem Entwurf in einer Systembeschreibungssprache zu erzeugen [13, S. 13]. Abbildung 1 zeigt die hier angesprochenen Entwicklungssprachen und deren Einsatzmöglichkeit auf verschiedenen Ebenen beziehungsweise in unterschiedlichen Bereichen.

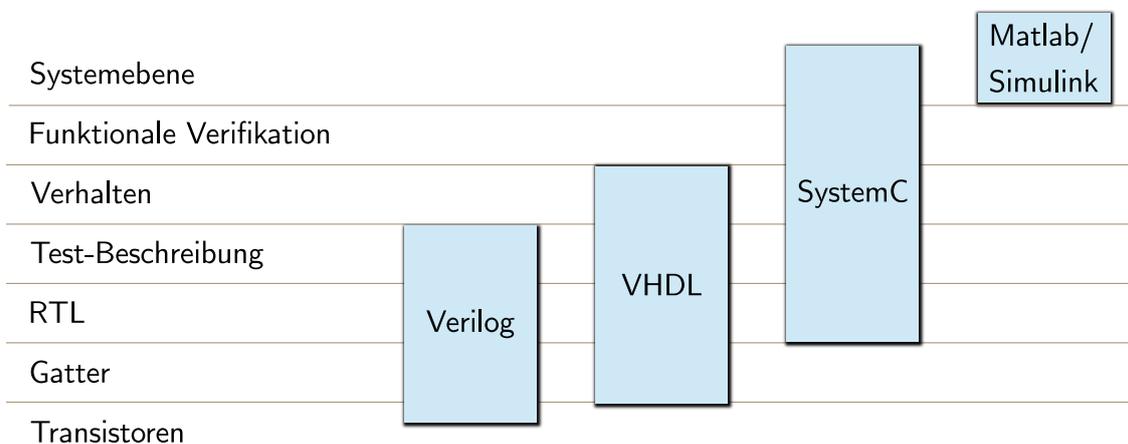


Abbildung 1: Vergleich von verschiedenen Entwicklungssprachen (nach [3, S. 15] und [14, S. 279])

Der Ansatz der Entwurfsentropie wird für die Verhaltens- und Strukturbeschreibung auf der Gatter- und Registertransferebene angewendet. Daher wird eine Hardwarebeschreibungssprache als Grundlage der Analyse gewählt. Somit basiert auch das Analysewerkzeug auf der Eingabe einer Beschreibung in einer Hardwarebeschreibungssprache. Als Sprachen kommen dabei entweder Verilog oder VHDL infrage, welche am weitesten verbreitet sind [13, S. 24].

Bei VHDL werden die Datentypen und Komponenten immer explizit angegeben. Dies hat für die automatische Analyse den Vorteil, dass diese Informationen nicht erst aus der Beschreibung extrahiert werden müssen. Wie in dem Codebeispiel 1 für einen Halbaddierer zu sehen ist, genügt es bei Verilog die Richtung der Ports zu definieren. Im Gegensatz hierzu ist bei VHDL zusätzlich auch der Datentyp anzugeben, wie in Codebeispiel 2 zu sehen ist.

```

1  module halbaddierer(x,y,s,c);
2  input  x,y;
3  output s,c;
4  assign s = x ^ y;
5  assign c = x & y;
6  endmodule

```

Code 1: Halbaddierer in Verilog

```

1  ENTITY halbaddierer IS
2  PORT ( x,y  : IN  BIT;
3         s,c  : OUT BIT);
4  END halbaddierer;
5
6  ARCHITECTURE Structure OF halbaddierer IS
7  BEGIN
8    s <= x XOR y;
9    c <= x AND y;
10 END Structure;

```

Code 2: Halbaddierer in VHDL

Zudem erfordert VHDL, dass bei der Einbindung von Komponenten (anderen Entitäten) diese mit ihren Ports explizit genannt werden, wie Codebeispiel 3 für den Volladdierer bei der Verwendung des Halbaddierers zeigt. Bei Verilog kann die Einbindung/Instanziierung anderer Module direkt über den Modulnamen erfolgen, wie Codebeispiel 4 zeigt.

```

1  ENTITY volladdierer IS
2  PORT ( x,y,c_in : IN  BIT;
3         s,c_out  : OUT BIT);
4  END volladdierer;
5
6  ARCHITECTURE Structure OF volladdierer IS
7  COMPONENT halbaddierer
8  PORT ( x,y  : IN  BIT;
9         s,c  : OUT BIT);
10 END COMPONENT;
11 SIGNAL s1, c1, s2: BIT;
12
13 BEGIN
14  HA1: halbaddierer PORT MAP(x,y,s1,c1);
15  HA2: halbaddierer PORT MAP(c1,c_in,s2,c_out);
16  s <= s1 OR s2;

```

17 **END** Structure;

Code 3: Volladdierer in VHDL

```
1 module volladdierer(x, y, c_in, s, c_out);
2   input  x, y, c_in;
3   output s, c_out;
4   wire   s1, c1, s2;
5   halbaddierer ha1(x, y, s1, c1);
6   halbaddierer ha2(c1, c_in, s2, c_out);
7   assign s = s1 | s2;
8 endmodule
```

Code 4: Volladdierer in Verilog

Für die Berechnung der Entwurfsentropie müssen grundsätzlich die Unterkomponenten analysiert werden. Daher gestaltet sich die Umsetzung eines automatischen Analysewerkzeuges in VHDL durch die explizite Komponentendeklaration wesentlich effizienter. Zudem ist VHDL in Europa verbreiteter als Verilog, das in den USA sehr verbreitet ist [13, S. 24] [15, S. 82]. Aus praktischen Gründen kam für die Auswahl von VHDL hinzu, dass der Verfasser der vorliegenden Arbeit wesentlich vertrauter mit VHDL als mit Verilog ist. Hilfreich bei der Erprobung des Analysewerkzeuges war auch der für VHDL verfügbare quelloffene Simulator GHDL [8] und das Programm GTKWave [5] zur Darstellung des Signalverlaufs, welche beide unter der GNU General Public License [7] veröffentlicht wurden und für verschiedene Betriebssysteme zur Verfügung stehen. Somit können Entwürfe unabhängig von der Verfügbarkeit kommerzieller Lizenzen implementiert und getestet werden, um diese dann als Grundlage der Analysen zu verwenden. Denn das Analysewerkzeug setzt einen (grammatisch/syntaktisch) korrekten Quellcode voraus und nimmt nur eine sehr grundlegende Prüfung der Grammatik vor. Somit sprachen die besseren Argumente für eine Verwendung von VHDL als Hardwarebeschreibungssprache.

Um dennoch praktisch festzustellen, dass das Konzept der Entwurfsentropie auch mit anderen Sprachen funktioniert, wurde ein kleiner Teil der Grammatik von Verilog separat für das Analysewerkzeug implementiert. Hierauf basierend konnten die gleichen Berechnungen wie bei der Verwendung von VHDL durchgeführt werden. Es besteht jedoch ein erheblicher Aufwand darin, die Grammatik zu implementieren und diese mit den entsprechenden Berechnungen für die Entwurfsentropie zu annotieren. Daher wurde der Schwerpunkt auf die Erweiterung des VHDL-Parsers gelegt, um auch größere Entwürfe mit verschiedenen Sprachkonstruktionen aus VHDL analysieren zu können. Der Verilog-Parser diente somit nur zu Testzwecken und wurde nicht weiter ausgebaut.

2.1 VHDL

Die Hardwarebeschreibungssprache VHDL (Very High Speed Integrated Circuit Hardware Description Language – VHDL) ist seit 1987 in dem IEEE-Standard 1076 [9] definiert, wobei die aktuelle Überarbeitung aus dem Jahr 2008 stammt [12]. VHDL wurde im Rahmen der VHSIC-Initiative

(Very High Speed Integrated Circuits – VHIC) der US-Regierung mit dem Verteidigungsministerium als Auftraggeber entwickelt [15, S. 65] und zunächst für die Dokumentation von Schaltungen eingesetzt [13, S. 22]. Die erste kommerzielle Version von IBM, Intermetrics und Texas Instruments wurde 1984 veröffentlicht [15, S. 65].

Bausteine werden in VHDL als Entitäten (entity oder design entity) modelliert, welche zum einen die Entitätsdeklaration sowie eine oder mehrere Architekturen enthalten [15, S. 66]. Die Anschlüsse (Ports) ermöglichen es, dass eine Komponente innerhalb anderer Komponenten verschaltet werden kann [13, S. 25]. Die Anschlüsse sind entweder Eingänge, Ausgänge oder bidirektionale Verbindungen [13, S. 25]. Bei den Beispielcodes 2 und 3 enthalten jeweils die Zeilen 1–4 die Entitätsdeklaration mit den Eingabe- und Ausgabeports.

Die Architektur beschreibt die innere Funktion (Verhaltensbeschreibung) oder die Struktur (Strukturbeschreibung) einer Komponente [13, S. 27]. Sind zu einer Komponente mehrere Architekturen vorhanden, wird über die Konfiguration mitgeteilt, welche Architektur verwendet werden soll [13, S. 27]. Bei dem Halbaddierer in Beispielcode 2 bilden die Zeilen 6–10 dessen Architektur. Für den Volladdierer in Beispielcode 3 besteht die Architektur aus den Zeilen 6–17. Innerhalb des Architekturkopfes wird der Halbaddierer mit seinen Anschlüssen durch eine Vorwärtsdeklaration innerhalb der Zeilen 7–10 als Komponente deklariert. Die Vorwärtsdeklaration stellt genügend Informationen über die Komponente zur Verfügung, um diese verwenden zu können, ohne dass die Komponente bereits implementiert sein muss [15, S. 67].

Die Halbaddiererkomponente wird durch sogenannte Portmaps in den Zeilen 14–15 verwendet („instanziiert“) und die Verbindungen zwischen der verwendenden Entität (Volladdierer) und den Ports der verwendeten Entität (Halbaddierer) hergestellt. Die Signale, welche zur Verbindung verwendet werden, entsprechen dabei physikalischen Verbindungen in Hardware, wie Leitungen oder Drähten [15, S. 65]. Bei digitaler Hardware zeichnen sich diese Signale dadurch aus, dass sie grundsätzlich zwei Zustände haben, welche regelmäßig mit „0“ und „1“ bezeichnet werden [28, S. 90]. Jedoch verwendet man zur Modellierung eines Systems oftmals eine mehrwertige Logik mit mehr als zwei Zuständen, auf welche im folgenden Abschnitt eingegangen wird.

2.2 Mehrwertige Logik

Oftmals ist es bei dem Entwurf notwendig, ein Signal nicht nur durch die Zustände „0“ oder „1“ modellieren zu können, sondern es ist auch ratsam, beispielsweise die Stärke eines Signals zu berücksichtigen [15, S. 68]. Teilweise werden auch unbestimmte Zustände oder hochohmige Zustände zur Modellierung benötigt [26, S. 73 ff.]. Daher werden solche Signale als mehrwertige Logik deklariert, welche bis zu sechsundvierzig Zustände haben können [15, S. 68].

In VHDL wird die mehrwertige Logik regelmäßig durch den IEEE-Standard 1164 [11] ausgedrückt, welcher mittlerweile in die aktuelle Überarbeitung des VHDL-IEEE-Standards 1076-2008 eingeflossen ist [12, S. 1]. Die Standardlogik (`std_(u)logic`) umfasst dabei neun Zustände [12, S. 514], wie in Tabelle 1 dargestellt. Abbildung 2 zeigt dabei die partielle Ordnung der Zustände ohne den

nicht initialisierten Zustand (U) und die beliebige Eingabe (-), welche nicht eingeordnet werden können [12, S. 277 ff.]. Die `std_ulogic` (u für unresolved) ist dabei der Grundtyp für skalare Signale und `std_ulogic_vector` für Vektoren [12, S. 517]. Existieren für ein Signal mehrere Treiber, so muss der Typ `std_logic(_vector)` verwendet werden [12, S. 517].

X	Unbekannt
0	Logische 0
1	Logische 1
Z	Hochohmig
W	Schwaches Signal
L	Schwache 0
H	Schwache 1
U	Nicht initialisiert
-	Beliebige Eingabe

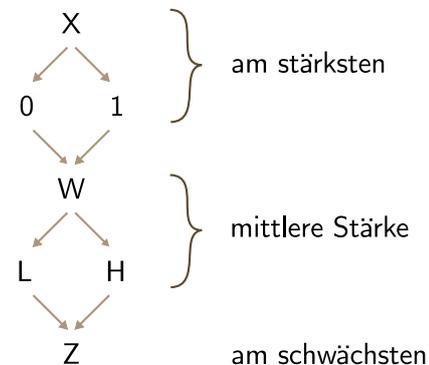


Tabelle 1: Standardlogik [12, S. 514]

Abbildung 2: Partielle Ordnung der Standardlogik [nach 15, S. 72]

Bei dem Entwurf eines Systems macht es für die Komplexität einen Unterschied, ob nur eine binäre Logik mit zwei Zuständen verwendet wird oder, ob zusätzliche Zustände berücksichtigt werden müssen. Dabei kommt es jedoch auch regelmäßig vor, dass innerhalb eines Entwurfes eine Person durchgängig eine Logik mit neun Zuständen für mehrere Treiber (`std_logic`) verwendet, obwohl nur mit binären Zuständen gearbeitet wird. Dies ist für die Entwurfsentropie dahin gehend unproblematisch, da sich durch die logarithmische Funktion die höhere Zustandsanzahl nur als „Skalierungsfaktor“ auswirkt. Somit muss bei dem Vergleich verschiedener Implementierungen gegebenenfalls nur ein Skalierungsfaktor berücksichtigt werden.

Eine angedachte Erweiterung hierzu ist die Prüfung der tatsächlich beachteten/verwendeten Signalzustände innerhalb der Implementierung. Dies könnte durch das Analysewerkzeug automatisiert werden. Um den VHDL-Quellcode zu parsen, wird das Analysewerkzeug auf einem Compiler aufgesetzt, dessen Grundzüge im folgenden Abschnitt kurz vorgestellt werden.

3 Compiler

Ein Compiler übersetzt den Quellcode einer Programmiersprache in eine andere Sprache [1, S. 2]. Regelmäßig ist dabei das Ziel, dass der Quellcode in einen (direkt) ausführbaren Code übersetzt wird [1, S. 1 f.] [19, S. 1]. Innerhalb der vorliegenden Arbeit wird das Frontend des Compilers dazu eingesetzt, die in VHDL beschriebene Hardware hinsichtlich ihrer Entwurfskomplexität zu analysieren [18].

Die Struktur eines Compilers kann in mehrere Phasen aufgeteilt werden [34, S. 1 ff.], wie Abbildung 3 zeigt. Dabei werden die ersten drei Analysephasen (lexikalische, syntaktische und

semantische Analyse) als Frontend und die weiteren Synthesephasen als Backend bezeichnet [34, S. 3]. Ziel der Analyse ist die Zerlegung des Quellcodes in seine Bestandteile und die Generierung eines Syntaxbaums [1, S. 6]. Dieser Syntaxbaum wird vorliegend dazu verwendet, um die Entwurfsentropie zu berechnen. Bei der Synthese wird aus der Zwischendarstellung der gewünschte Zielcode erzeugt [1, S. 7]. Dieser Teil eines Compilers wird für die Berechnung der Entwurfsentropie nicht benötigt, sodass auf die Synthese, das heißt, auf das Backend nicht weiter eingegangen wird. Um einen Überblick über die Phasen der Analyse zu geben, werden die typischen Arbeitsweisen im Folgenden kurz beschrieben, ohne dabei auf die unterschiedlichen Möglichkeiten der Umsetzung und die uneinheitlichen Begrifflichkeiten innerhalb der Literatur einzugehen.

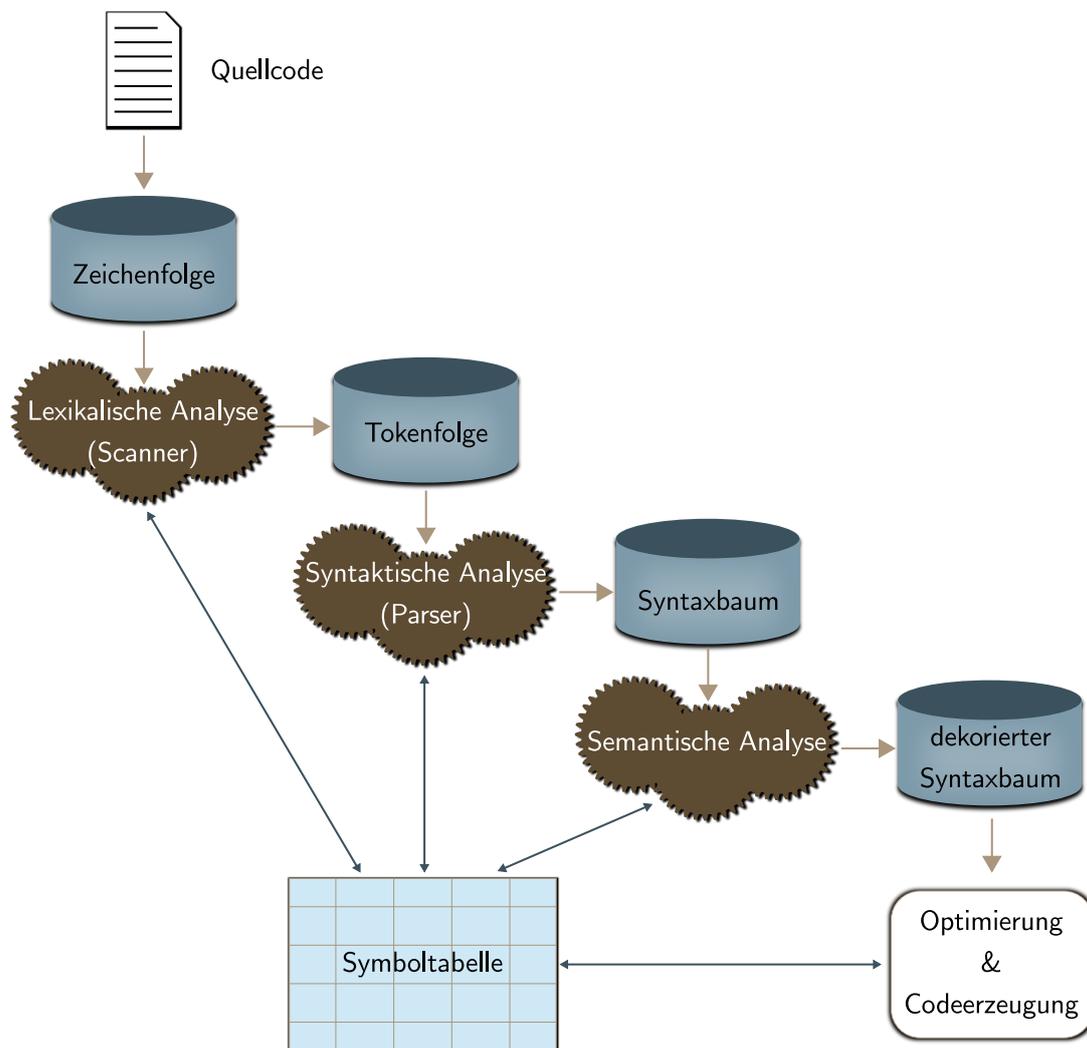


Abbildung 3: Phasen eines Compilers

1. Lexikalische Analyse (Scanner):

Der Quellcode wird als Zeichenfolge eingelesen und in lexikalische Einheiten (Tokens oder Symbole) gruppiert [1, S. 135 f.] [33, S. 3]. Lexikalische Einheiten bestehen aus einem Namen, wie beispielsweise Schlüsselwörter oder eine Folge von Eingabezeichen, und einem optionalen Attribut, welches der Zeiger auf den betreffenden Eintrag in der Symboltabelle ist [1, S. 136 ff.]. Innerhalb dieser Analysephase können auch Kommentare und (überflüssige) Leerzeichen entfernt werden [1, S. 135 f.]. Als Eingabe dient die Zeichenfolge des Quellcodes,

durch welche die Positionen von den in späteren Phasen erkannten Fehlern im Quellcode, insbesondere durch das Mitzählen der Zeilenumbrüche, verknüpft und ausgegeben werden können [1, S. 135].

Als Beispiel für die lexikalische Analyse soll Zeile 8 aus Beispielcode 2 dienen, welche dem Signal `s` den „Wert“ der Antivalenz-Verknüpfung aus `x` und `y` zuweist:

```

s      <=      x      XOR      y
ident  assignment  ident  relation  ident

```

2. Syntaktische Analyse (Parser):

Der Parser übernimmt die syntaktische Analyse und erhält als Eingabe die Tokenfolge des Scanners beziehungsweise fordert die Tokens nacheinander an [33, S. 49]. Dabei wird überprüft, ob die Tokenfolge von der Grammatik der Quellsprache erzeugt werden kann [1, S. 233]. Parser arbeiten üblicherweise mit einer Top-Down- oder Bottom-Up-Methode, das heißt, dass der Parserbaum entweder von der Wurzel aus oder von den Blättern aus aufgebaut wird [1, S. 234]. Somit erkennt die syntaktische Analyse die hierarchische Struktur eines Programms [33, S. 5 f.]. Die Grammatik einer Sprache wird dabei regelmäßig als (E)BNF beschrieben ((Erweiterte) Backus Naur Form) [34, S. 7 ff.]. Somit lässt sich der Parser aus der (E)BNF ableiten [34, S. 14]. Die syntaktische Struktur wird mithilfe eines Syntaxbaums repräsentiert [33, S. 13].

Für das vorherige Beispiel ist der (vereinfachte) Syntaxbaum als Ergebnis der lexikalischen Analyse in Abbildung 4 dargestellt.

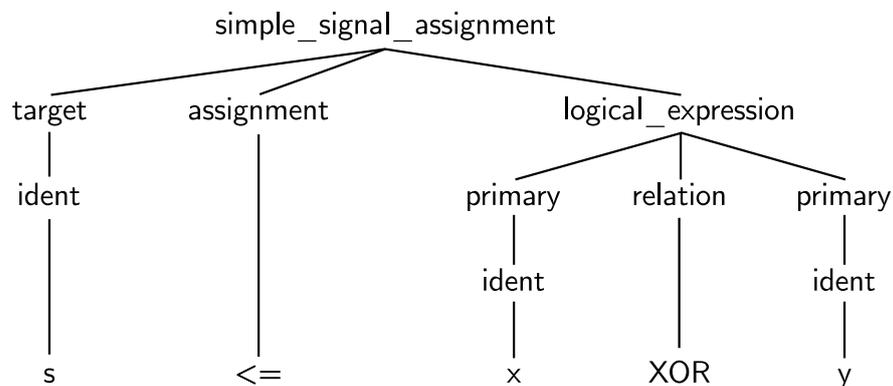


Abbildung 4: Syntaxbaum

3. Semantische Analyse:

Teilweise sind bestimmte Teile eines Programms nur durch den Kontext erkennbar [33, S. 153]. Dieses drückt sich durch die Deklariertheit (Mehrfachbezeichnungen), die Gültigkeit (für welchen Teil hat eine Deklaration die Gültigkeit) und die Sichtbarkeit (versteckte und sichtbare Bezeichner) von Bezeichnern aus [33, S. 153]. Hierzu gehört auch die Prüfung der Typenkonsistenz [1, S. 11]. Um den Syntaxbaum mit Attributen zu versehen, welche Bedeutung die Tokens im jeweiligen Fall haben, muss vorausgeschaut (lookahead) werden und aus dem Kontext des Tokens dessen Zuordnung bestimmt werden [1, S. 312 ff.].

Eine vollständige Implementierung der einzelnen Phasen eines Compilers ist sehr aufwendig. Daher stehen sogenannte Parsergeneratoren zur Verfügung, welche die Erstellung des Frontends eines Parsers in verschiedenen Ausprägungen automatisieren [34, S. 35]. Dabei nutzt das Analyserwerkzeug für die Entwurfsentropie den vom Compiler-Compiler „ANTLR“ generierten Code, welcher im folgenden Abschnitt vorgestellt wird.

4 ANTLR

Ein Compiler-Compiler (Parsergenerator) erleichtert den Aufbau des Compiler-Frontends [1, S. 343]. Dabei wird nicht nur der Parser selbst generiert, sondern meist auch der Scanner und die semantische Analyse [1, S. 15, 1199]. Als Quellen dienen die formellen Beschreibungen der Syntax und der Semantik einer Sprache [27, S. 11]. Die formelle Beschreibung liegt dabei meist als (E)BNF vor.

Vorliegend wird ANTLR (ANother Tool for Language Recognition – ANTLR) als Compiler-Compiler eingesetzt. Mittlerweile ist er in der Version 4 erschienen [24], wobei das hier vorgestellte Tool ursprünglich in Version 3 von ANTLR entwickelt wurde (Version 3.2) [20]. Die Entscheidung für ANTLR beruht auf verschiedenen Gründen [18]:

- ANTLR ist sehr gut dokumentiert [21] [23] [31] [32] und es existieren zahlreiche Beispiele.
- Der von ANTLR generierte Java-Code bleibt weiterhin sehr gut lesbar und strukturiert und damit auch debugbar.
- Die Grammatikentwicklungsumgebungen ANTLRWorks [4] und ANTLRWorks 2 [2] unterstützen die Implementierung der Grammatik aus einer EBNF-Darstellung. Zudem ist ANTLR 4 IDE [6] als Plugin für Entwicklungsplattform Eclipse [29] verfügbar. Ein Bildschirmfoto von dem Programm ist in Abbildung 5 gezeigt.
- Sowohl ANTLR [22] [25] als auch ANTLRWorks [4] sind unter der BSD Lizenz veröffentlicht, ANTLRWorks 2 [2] wurde unter der GNU Lizenz veröffentlicht und ANTLR 4 IDE [6] unter der Eclipse Public License [30].
- ANTLR ist extrem flexibel und automatisiert viele Arbeiten.
- Die Berechnungen der Entwurfsentropie können direkt in die Grammatik eingefügt werden und werden „mitübersetzt“.

ANTLR erzeugt automatisch die lexikalische Analyse und syntaktische Analyse aus einer gegebenen Grammatik [21, S. 4]. Zudem wird auch die semantische Analyse erzeugt, da auch der Kontext durch ein (semantisches und syntaktisches) Vorausschauen betrachtet wird [21, S. 9, 74]. Die Größe des Vorausschauens LL(*) wird dabei dynamisch eingestellt, sodass eine Vielzahl von

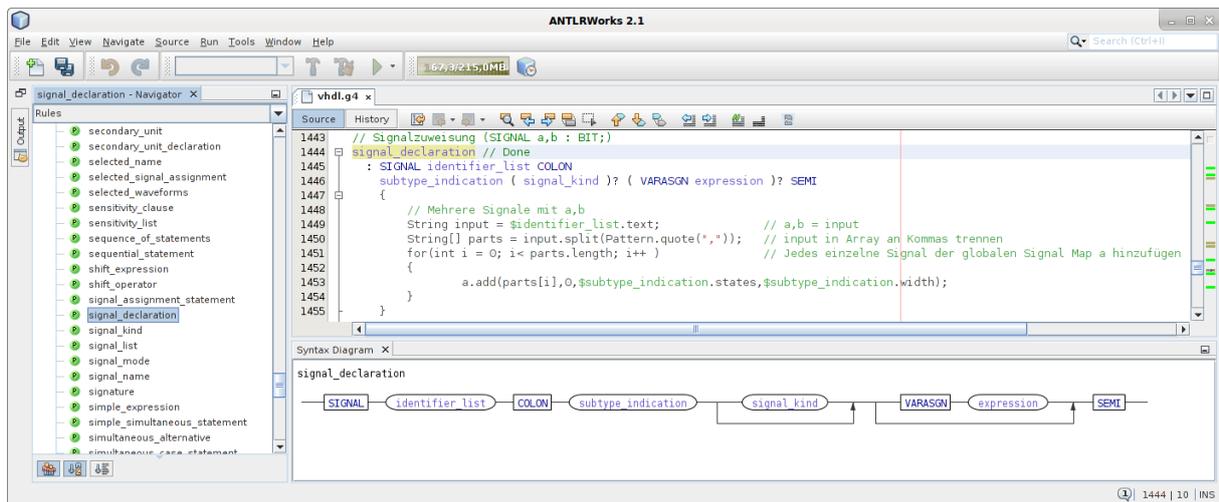


Abbildung 5: ANTLRWorks 2.1

Grammatiken erkannt werden können [21, S. 36 f.]. In die Grammatik können Prozesse (action) eingebracht werden, um einen Übersetzer zu bauen [21, S. 4]. Anstatt den Quelltext zu übersetzen, werden die Prozesse dafür verwendet, die Entwurfentropie zu berechnen.

Zur Beschreibung der Grammatik wird ein Dialekt der Backus Naur Form verwendet, bei welcher Regeln mit einem Kleinbuchstaben und Tokens mit einem Großbuchstaben beginnen [21, S. 73]. Ebenfalls ist die Beschreibung als Erweiterte Backus Naur Form möglich, welche auch Unterregeln enthalten kann (z. B. Alternativen, optionale Elemente) [21, S. 74, 83 f.]. Die Prozesse werden durch geschweifte Klammern direkt in die Grammatikdatei in der Zielsprache, im vorliegenden Java, aufgenommen [21, S. 78]. Dabei kann auch auf die Tokens (Attribute) zugegriffen werden [21, S. 81]. Ein Minimalbeispiel einer Grammatik ist in Beispielcode 5 gegeben.

```

1  grammar Minimalbeispiel;
2
3  r:
4      target = IDENT '<=' '\' value = INT '\';
5      {
6          System.out.println("Signal " + $target.text + " ist " +
7              ($value.text.equals("1") ? "high" : "low"));
8      }
9      ;
10
11  IDENT: 'a'..'z'+;
12  INT: ('0'|'1');
13  WS: (' '\r'\n')+ {$channel = HIDDEN;};

```

Grammatikregel

Matchen von Eingaben wie "out <= '1';"

Anfang eines Codeblocks

Ausgabe von z.B. "Signal out ist high"

Ende des Codeblocks

Ende der Regel

Bezeichnung aus Kleinbuchstaben

Die Zahl aus 0 oder 1

Leerzeichen ignorieren

Code 5: Minimalbeispiel einer Grammatik

In dem Beispiel können Signalzuweisungen mit „Bezeichner <= '1/0'“ als Eingabe dienen, beispielsweise „out <= '1'“ oder „a <= '0'“. Die Grammatikregel r in Zeile 4 besteht dabei aus dem Token IDENT, dessen Wert der Variablen target zugewiesen wird, der Zeichenfolge <= ', dem Token INT, dessen Wert der Variablen value zugewiesen wird und der Zeichenfolge

'; . Der Javacode ist in dem Codeblock in den Zeilen 5–8 enthalten und wird in den Parser mitübernommen. Dabei kann innerhalb von Zeile 6 auf den Inhalt der Variablen `target` und in Zeile 7 auf den Inhalt der Variablen `value` zugegriffen werden, dessen Textinhalt in diesem einfachen Beispiel nur ausgegeben wird. Die Ausgaben für die genannten Beispiele wären `Signal out ist high` und `Signal a ist low`. Die letzten drei Zeilen des Beispielcodes definieren die Regeln für die Tokens `IDENT`, welcher aus Kleinbuchstaben bestehen kann, `INT`, welcher entweder 0 oder 1 sein kann und `WS` für Leerzeichen, welche ignoriert werden. In Beispielcode 6 ist ein Teil des von ANTLR erzeugten Lexers zu sehen, welcher prüft, ob die Eingabefolge dem Token `INT` entspricht und in Beispielcode 7 ist die generierte Grammatikregel `r` zu sehen.

```

1 public final void mINT() throws RecognitionException {
2     try {
3         int _type = INT;
4         int _channel = DEFAULT_TOKEN_CHANNEL;
5         {
6             if ( input.LA(1)=='0' || input.LA(1)=='1' ) {
7                 input.consume();
8             }
9             else {
10                MismatchedSetException mse = new MismatchedSetException(null,input);
11                recover(mse);
12                throw mse;}
13        }
14        state.type = _type;
15        state.channel = _channel;
16    }
17    finally {
18    }
19 }

```

Code 6: Erzeugter Lexer für INT

```

1 public final void r() throws RecognitionException {
2     Token target=null;
3     Token value=null;
4     try {
5         target=(Token)match(input,IDENT,FOLLOW_IDENT_in_r19);
6         match(input,7,FOLLOW_7_in_r21);
7         match(input,8,FOLLOW_8_in_r23);
8         value=(Token)match(input,INT,FOLLOW_INT_in_r29);
9         match(input,9,FOLLOW_9_in_r31);
10        System.out.println("Signal " + (target!=null?target.getText():null) +
11        " ist " + ((value!=null?value.getText():null).equals("1") ? "high" : "low"));
12    }
13    catch (RecognitionException re) {
14        reportError(re);
15        recover(input,re);
16    }
17    finally {

```

```
18 }  
19 return ;  
20 }
```

Code 7: Teil des erzeugten Parsers

Dieses Minimalbeispiel erzeugt bereits einen Scanner (Lexer) mit 350 Zeilen und einen Parser mit 85 Zeilen. Daher konnte hier nur ein Teil der beiden Dateien dargestellt werden. Zur Übersichtlichkeit wurden die in den beiden Codebeispielen enthaltenen Kommentare von ANTLR sowie die aus der Grammatik übernommenen Kommentare entfernt. Anhand der Beispiele ist sehr gut erkennbar, dass der von ANTLR erzeugte Code sehr gut lesbar bleibt und somit auch das Finden von Fehlern wesentlich erleichtert wird. Die normalerweise enthaltenen Kommentare helfen dabei zusätzlich für das Verständnis, ebenso die Möglichkeit, das Programm beispielsweise in dem Eclipse-Framework [29] in der Debugansicht Schritt für Schritt ausführen zu lassen.

Der Einsatz des Compiler-Compilers unterstützt die Entwicklung des Analysewerkzeuges durch ein wesentlich strukturierteres Vorgehen als die direkte Implementierung eines Parsers und vermeidet damit zusätzliche Fehlerquellen [18]. Dabei verlagert sich die Arbeit von der eigentlichen Implementierung hin zu dem Schreiben der Grammatik und die Erweiterung der Grammatikdatei mit den Berechnungen für Entwurfsentropie. Dabei wird im folgenden Abschnitt die konkrete Umsetzung des Analysewerkzeuges beschrieben.

5 Umsetzung

Die Umsetzung des Analysewerkzeuges basiert auf der durch den IEEE-Standard 1076 [10] gegebenen Grammatik von VHDL. Hierin sind die einzelnen Grammatikregeln als EBNF beschrieben. Basierend auf diesen Regeln kann eine Grammatikdatei erzeugt werden, welche eine in VHDL gegebene Hardwarebeschreibung parsen kann [18]. Die erste Version des Analysewerkzeuges beruhte auf Version 3 von ANTLR und wurde innerhalb einer Bachelorarbeit eines Studierenden entwickelt und gemeinsam publiziert [18]. Dabei wurde nur ein Teil der VHDL-Grammatik implementiert, sodass zunächst die Analyse auf der Gatterebene möglich war [18]. Von dem Autor der vorliegenden Arbeit wurde die Analyse um fehlende Grammatikregeln, Analyseschritte und weitere Funktionalitäten sukzessive erweitert. Die erste Version lief dabei sehr zuverlässig und die Analyse lief auch für größere Projekte innerhalb weniger Sekunden ab. Für einen Ripple-Carry-Addierer, welcher aus Voll- und Halbaddierern aufgebaut ist, ist die Ausgabe der ersten Version des Analysewerkzeuges in Abbildung 6 gezeigt. Dabei wird zunächst die Analyse der Top-Level-Entity (`rca`) vorgenommen. Diese verwendet als Unterkomponenten Volladdierer (`full_adder`). Für die Berechnung der Entwurfsentropie des Ripple-Carry-Addierers wird dabei die Verhaltens- und Strukturentropie des Volladdierers benötigt, sodass diese Komponente zunächst analysiert werden muss. Der Volladdierer selbst verwendet wiederum die Unterkomponente Halbaddierer (`half_adder`), sodass diese zunächst analysiert werden muss. Anhand der Entropien der Halbaddiererkomponente werden die Entropien der Volladdiererkomponente berechnet, mit welchen die Strukturentropie der Top-Level-Entität berechnet wird. Die Verhaltensentropie des Ripple-Carry-Addierers ergibt sich anhand der Schnittstellen und deren möglichen Zuständen. Für

größere Projekte mit mehr Unterkomponenten fällt die Ausgabe dementsprechend umfangreicher aus.



```
$ java source.Run
Name of the program to analyze (w/o .vhd): rca

Top-Level-Entity: rca
=====

Component rca is used.
--> Component not yet analyzed. Starting analysis...

Component full_adder is used.
--> Component not yet analyzed. Starting analysis...

Component halbaddierer is used.
--> Component not yet analyzed. Starting analysis...
<-- Component halbaddierer
  Behavior Entropy: 4 * log(2) = 1.2
  Structure Entropy: 6 * log(2) = 1.81

<-- Component full_adder
  Behavior Entropy: 5 * log(2) = 1.51
  Structure Entropy: 17 * log(2) = 5.12

<-- Component rca
  Behavior Entropy: 14 * log(2) = 4.21
  Structure Entropy: 37 * log(2) = 11.14

=====
Behavior Entropy for Top-Level-Entity rca : 14 * log(2) = 4.21
Structure Entropy for Top-Level-Entity rca : 37 * log(2) = 11.14
```

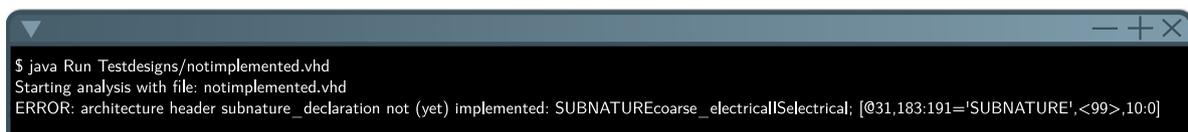
Abbildung 6: Ausgabe der ersten Version des Analysewerkzeuges (ähnlich [18])

Dabei zeigt sich, dass die gewählte Umsetzung mehrere Herausforderungen hatte. Im Kern standen dabei zwei Punkte: Zum einen war die Erweiterung der Grammatik sehr aufwendig, denn neue Grammatikregeln mussten fast immer bis zur Wurzel zurück oder zumindest über viele „Ebenen“ hinweg hinzugefügt werden [18]. Dabei mussten auch regelmäßig bestehende Regeln geändert werden und Berechnungsanweisungen umgeschrieben werden, sodass nach jeder Änderung nicht nur der hinzukommende Teil sondern das ganze Programm ausführlich getestet werden musste. Zum anderen war die Datenstruktur zum Speichern der Daten für die Berechnung der Entwurfsentropie so implementiert worden, dass das Objekt (der Klasse Hashtable) immer von jeder Grammatikregel (Parserregel) an die nächste weitergegeben werden musste [18]. Somit enthielt jede Parserregel zumindest einen Codeteil mit einer Rückgabe (return), auch wenn innerhalb dieser Regeln keine Berechnungen (oder Teilberechnungen) für die Entwurfsentropie stattfanden [18]. Insbesondere das Zusammenspiel der beiden Kernpunkte machte die Erweiterung sehr aufwendig.

Daneben war es noch nachteilig, dass die zu analysierende VHDL-Datei nicht als Argument übergeben werden konnte, Großbuchstaben erst in Kleinbuchstaben umgewandelt werden mussten, da die Grammatik von Kleinbuchstaben ausging, obwohl VHDL keinen Unterschied zwischen Groß- und Kleinschreibung (not case-sensitive) macht [18] und die Ausgabe bei größeren Dateien sehr umfangreich ausfiel. Für die Fehleranalyse fehlten zudem entsprechende Debugausgaben. Wenn noch nicht implementierte Grammatikteile verwendet wurden, brach das Programm ab, und erst ein Debugging mit Eclipse oder ANTLRWorks zeigte, welche Quellcodeaussage noch nicht in

der Grammatik abgebildet war. Hinzu kam, dass bei sehr großen Systemen die Entwurfsentropie so groß wurde, dass die verwendeten Integertypen zur Speicherung der Entropiewerte nicht ausreichten, sodass das System in Teilen analysiert werden musste und die Entwurfsentropie des Gesamtsystems von Hand ausgerechnet werden musste. Wie bereits beschrieben, enthielt jede Grammatikregel seine eigene Rückgabe (return), sodass die Umstellung auf einen anderen Datentyp als Integer nicht durch ein Ersetzen trivial zu lösen war.

Aus den im Vorherigen genannten Gründen wurde eine zweite Version des Analysewerkzeuges komplett neu entwickelt. Dabei wurde auch die aktuelle Version 4 von ANTLR verwendet (Version 4.5.1). Das neue Analysewerkzeug enthält eine (fast) vollständige Grammatik von VHDL, wobei neben den Regeln des VHDL-Standards 1076 von 2008 [12] noch weitere Regeln, insbesondere für die lexikalische Analyse, enthalten sind. Diese ist nun auch, bis auf den Dateinamen, unabhängig von Groß- und Kleinschreibung. Um kenntlich zu machen, an welchen Stellen bis jetzt noch keine Berechnung der Entwurfsentropie erfolgt, wurden Anweisungen hinzugefügt, welche Aussagen des Quellcodes nicht hinsichtlich der Entwurfsentropie analysiert werden können. Eine Beispielausgabe für die noch nicht implementierte Berechnung einer `subnature` Deklaration ist in Abbildung 7 gezeigt. Dabei wird ausgegeben, dass die fehlende Implementierung eine Regel innerhalb des Architekturkopfes ist, es sich um die Anweisung `SUBNATURE coarse_electrical IS electrical;` (Ausgabe erfolgt ohne Leerstellen) handelt und sich diese in Zeile 10 des Quellcodes befindet. Dabei ist die Grammatikregel hierfür bereits schon implementiert, jedoch ohne den Code, welcher für die Berechnung der Entwurfsentropie erforderlich ist, sodass das Programm wie bei der ersten Version des Analysewerkzeuges auch weiterhin abbricht, nur mit einer detaillierten Ausgabe der entsprechenden Stelle. Hierdurch kann nun einfach die Stelle aufgefunden werden, an welcher die Berechnung hinzugefügt werden muss.



```
$ java Run Testdesigns/notimplemented.vhd
Starting analysis with file: notimplemented.vhd
ERROR: architecture header subnature_declaration not (yet) implemented: SUBNATUREcoarse_electricalISElectrical; [031,183:191='SUBNATURE',<99>,10:0]
```

Abbildung 7: Ausgabe bei einer noch nicht berechenbaren Grammatikregel

Zudem wurde eine „globale“ Tabelle (HashMap) verwendet, um die Daten zur Berechnung der Entwurfsentropie zu speichern. Somit müssen nur die Stellen der Grammatik mit Berechnungsregeln erweitert werden, welche tatsächlich für die Entwurfsentropieberechnung (oder Teile der Berechnung) für die noch fehlenden Regeln erforderlich sind. Die Tabelle wurde als eigene Klasse mit der Unterklasse Tabelleneinträge umgesetzt. Der Parser instanziiert zu Beginn der Analyse ein neues Objekt der Tabelle für jede zu analysierende Datei. Die Klasse der Tabelle enthält dabei auch eine Methode zur Berechnung der Entwurfsentropie sowie die Möglichkeit, alle Signale einer Komponente mit den Verwendungen auszugeben. Zusätzlich kann über eine Auswahl angegeben werden, ob nur die Entwurfsentropie des Systems ausgegeben werden soll, wie in Abbildung 8 gezeigt, oder auch die Strukturentropie der einzelnen Komponenten, wie in der etwas später folgenden Abbildung 10 gezeigt.

```

$ java Run Testdesigns/Addierer/rca.vhd
Starting analysis with file: rca.vhd
=====
--> Design Entropy for input file rca.vhd: 11.14

```

Abbildung 8: Ausgabe der zweiten Version des Analysewerkzeuges

Die Verwendung einer „globalen“ Tabelle für jede zu analysierende Entität wird durch das folgende Beispiel der Signaldefinition `SIGNAL a,b : BIT_VECTOR(1 downto 0);` verdeutlicht. Diese Signaldefinition hat den in Abbildung 9 dargestellten Syntaxbaum (parse tree). Für diese Anweisung müssen in der Tabelle die Anzahl der Zustände für die Signale `a`, `b` gespeichert werden. Die Grammatikregeln einiger Teile hierzu sind in Codebeispiel 8 (verkürzt und ohne Kommentare) wiedergegeben.

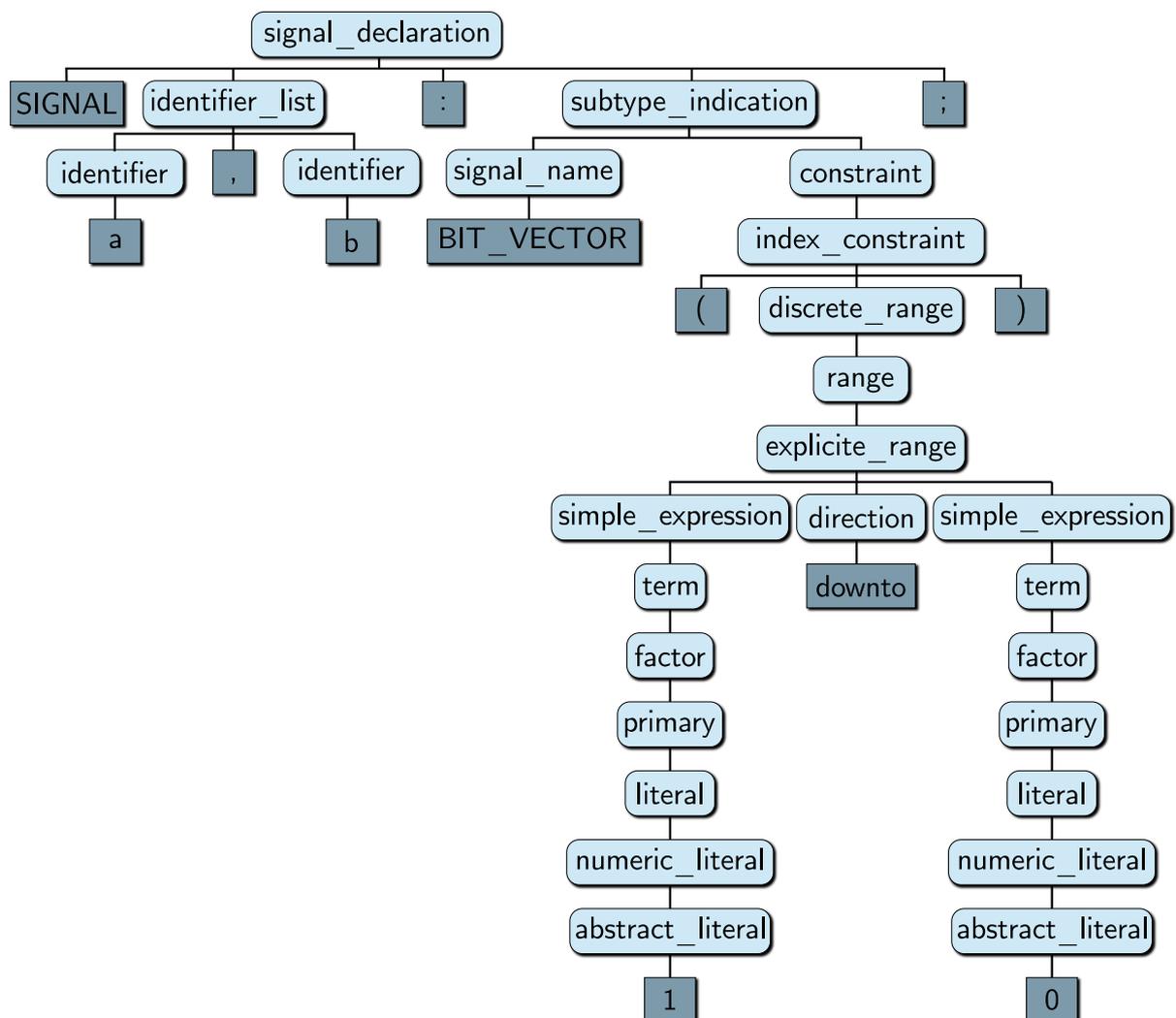


Abbildung 9: Syntaxbaum (Parse Tree)

Die Wurzel (dieses Teilbaums) ist die Grammatikregel `signal_declaration` in Zeile 1. Hierin wird direkt der Inhalt der `identifier_list` geparkt (Zeilen 4–10). Für die Gesamtanzahl der Zustände

hat die Regel `subtype_indication` als Rückgabewerte die Anzahl der Zustände (`states`) sowie die Breite des Bitvektors (`width`). Innerhalb dieser Anweisung sieht man in Zeile 19, dass die Verwendung der Anweisung `TOLERANCE...` eine Fehlerausgabe verursachen würde, da hinter dieser Grammatikregel noch keine Berechnungen hinterlegt sind. Die Anzahl der Zustände wird aus der Regel `signal_name` in den Zeilen 23–27 zurückgegeben, während die Breite aus der Regel `explicit_range` in den Zeilen 29–31 zurückgegeben wird. Hierzu ist es erforderlich, dass die Regeln `constraint`, `index_constraint`, `discrete_range` und `range` als Rückgaben ebenfalls den Wert für die Breite „weitergeben“. Mithilfe all dieser Informationen können dann die Signale `a,b` in Zeile 9 innerhalb der Regel `signal_declaration` zu der Tabelle mit Signalen hinzugefügt werden. Die `add`-Funktion wird dabei von der Tabellenkasse bereitgestellt.

```

1  signal_declaration
2  : SIGNAL identifier_list COLON
3  subtype_indication ( signal_kind )? ( VARASGN expression )? SEMI
4  {
5  String input = $identifier_list.text;
6  String[] parts = input.split(Pattern.quote(","));
7  for(int i = 0; i < parts.length; i++)
8  {
9  a.add(parts[i],0,$subtype_indication.states,$subtype_indication.width);
10 }
11 }
12 ;
13
14 subtype_indication returns [int states, int width]
15 : signal_name { $states=$signal_name.states; }
16 ( signal_name { $states+=$signal_name.states; } )?
17 ( constraint { $width += $constraint.width; } )?
18 ( tolerance_aspect
19 { error("signal subtype tolerance_aspect",$tolerance_aspect.text,$tolerance_aspect.start);}
20 )?
21 ;
22
23 signal_name returns [int states] // Vereinfacht
24 : BIT { $states = 2; }
25 | BITVECTOR { $states = 2; }
26 // Gekürzt
27 ;
28
29 explicit_range returns [int width]
30 : value1 = simple_expression direction value2 = simple_expression
31 { $width = Math.abs($value1.value - $value2.value) + 1;}
32 ;

```

Code 8: Ausschnitt der Grammatikdatei

Innerhalb des Beispielcodes 8 ist die neue Datenstruktur zur Berechnung der Entropie erkennbar. Der gesamte Syntaxbaum wird so in Teilbäume zerlegt, welche Rückgabewerte der Kinder benötigen. Jedoch müssen diese Rückgabewerte nur bis zu demjenigen Knoten zurückgegeben

werden, welcher aus allen Daten einen Eintrag in die Tabelle erstellen kann. In dem vorliegenden Fall werden neben dem Namen der Signale dessen Zustandsanzahl und dessen Breite benötigt.

Der in der Grammatik enthaltene Java-Code wird durch ANTLR mitübersetzt und es werden zwei Hauptklassen generiert: der Parser und der Scanner (Lexer). Das eigentliche Java-Programm, welches zur Analyse aufgerufen wird, bindet diese beiden Klassen zur Analyse ein. Das Hauptprogramm `Run` enthält dabei eine Klasse `calculate(String name)`, welcher der Name der VHDL-Datei übergeben wird. Innerhalb dieser Klasse wird zunächst geprüft, ob die Strukturentropie bereits berechnet wurde und somit nicht mehr berücksichtigt werden darf. Ist dies nicht der Fall, wird geprüft, ob die Datei vorhanden ist. Falls die Datei nicht vorhanden ist, wird davon ausgegangen, dass es sich um einen wiederverwendeten (Teil-)Entwurf, insbesondere aus einer Bibliothek, handelt, für welche die Strukturentropie null ist. Ist die Datei vorhanden, so wird innerhalb der Kommentare nach dem Ausdruck „- - Reused“ gesucht. Mit diesem Kommentar kann kenntlich gemacht werden, dass die Entität nicht innerhalb der Entwicklung des zu analysierenden Systems entwickelt wurde, obwohl die Implementierungsdatei innerhalb des Systems vorhanden ist. In diesem Fall wird ebenfalls eine Strukturentropie von null angenommen. Die Ausgabe bei der Analyse des Volladdierers mit einem wiederverwendeten Halbaddierer ist in Abbildung 10 gezeigt, für welche auch die Ausgabe der Strukturentropie der einzelnen Komponenten aktiviert ist.



```
$ java Run Testdesigns/volladdierer.vhd
Starting analysis with file: volladdierer.vhd

Finished analysis: Structure entropy of all files:
=====
haldaddierer.vhd          (Reused)
volladdierer.vhd         3.31
=====
--> Design Entropy for input file volladdierer.vhd: 3.31
```

Abbildung 10: Ausgabe des Analysewerkzeuges bei wiederverwendeten Komponenten

Liegt keine wiederverwendete oder bereits analysierte Komponente vor, wird die Analyse durch den Aufruf `components.put(name, parser.design_file().entropy)` gestartet. Das Ergebnis, die Entwurfsentropie der zu analysierenden Datei, wird nach der Analyse in eine Liste `components` mit den Komponentennamen und der Entwurfsentropie gespeichert. Die Methode zur Berechnung der Strukturentropie `entropy` wird von der Tabellenklasse bereitgestellt und liefert als Ergebnis eine `Double` mit dem Wert der Strukturentropie zurück. Die Verwendung des Datentyps „`Double`“ sowie die Möglichkeit auch Potenzen in der Tabelle speichern zu können, erlaubt nun auch die vollautomatische Analyse von sehr großen Systemen. Wird innerhalb des Parsens der Entität eine Unterkomponente oder eine andere, verwendete Entität gefunden, so wird eine neue Instanz der Berechnungsroutine `Run.calculate($id.text+".vhd")` rekursiv erzeugt, wie der Codeausschnitt 9 zeigt.

```

1 component_declaration
2   : COMPONENT id = identifier ( IS )?
3   {
4     Run.calculate($id.text+".vhd");           Rekursiver Aufruf mit der Unterkomponente
5   }
6   ( generic_clause
7     ( component_port_clause )?
8     END COMPONENT ( identifier )? SEMI
9   ;

```

Code 9: Komponentendeklaration

Das kompilierte Hauptprogramm `Run` kann direkt von der Konsole aus aufgerufen werden. Die zu analysierende Datei kann dabei als Argument übergeben werden. Bei einem Projekt mit mehreren Dateien wird der Name des Systems (top-level entity) übergeben. Wird kein Dateiname übergeben, fragt das Programm nach der zu analysierenden Datei beziehungsweise nach dem zu analysierenden System. Daraufhin startet die Analyse der eingegebenen Komponente durch den Aufruf der Methode `calculate(name)`. Werden Unterkomponenten gefunden, werden diese automatisch nach dem oben beschriebenen Vorgehen analysiert.

6 Zusammenfassung

Zusammenfassend erlaubt die neue Version des Analysewerkzeuges eine wesentlich bessere Weiterentwicklung. So sind (fast) alle Grammatikregeln in der Grammatikdatei bereits enthalten, jedoch wird die Analyse an der Stelle abgebrochen, an welcher noch keine Berechnungen der Entwurfsentropie hinterlegt sind. Dabei wird detailliert ausgegeben, welche Anweisung des Quellcodes zum Abbruch der Analyse geführt hat, sodass an dieser Stelle die Berechnung in die Grammatik eingefügt werden kann. Durch die Überarbeitung der Rekursion und der Datenaufzeichnung innerhalb des Parsers kann das Hinzufügen der Berechnungen für einzelne Teilbäume getrennt erfolgen, ohne dass die berechneten Werte wieder bis zur Wurzel des Parsers zurückgegeben werden müssen.

Das Analysewerkzeug erlaubt somit, eine in VHDL gegebene Beschreibung von digitaler Hardware automatisch auf ihre Komplexität hin zu analysieren. Dabei ist es in VHDL möglich, Entwürfe in verschiedenen Domänen und auf verschiedenen Ebenen zu beschreiben, sodass die Analyse auf mehreren Ebenen und in mehreren Domänen möglich ist. Mittlerweile enthält die Grammatikdatei knapp 2.000 Zeilen Code inklusive der Java-/Berechnungsanweisungen (ohne Kommentare) und kann alle gängigen Struktur- und Verhaltensbeschreibungen auf der Logikebene und Registertransferebene analysieren. Auch die Analyse abstrakter Beschreibungen auf der Systemebene ist möglich. Auch in diesem Fall können die Komponenten mit Schnittstellen modelliert werden und miteinander verbunden werden.

Literaturverzeichnis

- [1] Aho, Alfred et al., *Compiler – Prinzipien, Techniken und Werkzeuge*, 2. Auflage, Pearson Studium, München, Deutschland, 2008.
- [2] *ANTLRWorks 2*, www.tunnelvisionlabs.com/products/demo/antlrworks (Abgerufen am 25. 11. 2015), Helotes, TX, USA.
- [3] Black, David et al., *SystemC – From the Ground Up*, 2. Auflage, Springer, New York, NY, USA, 2010.
- [4] Bovetn, Jean und Parr, Terence, *ANTLRWorks*, wwwantlr3.org/works (Abgerufen am 20. 11. 2015).
- [5] Bybell, Tony et al., *GTKWave*, gtkwave.sourceforge.net (Abgerufen am 20. 11. 2015).
- [6] Espina, Edgar, *ANTLR 4 IDE*, www.github.com/jknack/antlr4ide (Abgerufen am 25. 11. 2015), Buenos Aires, Argentinien.
- [7] Free Software Foundation (Hrsg.), *GNU Licenses*, www.gnu.org/licenses (Abgerufen am 20. 11. 2015), Boston, MA, USA.
- [8] Gingold, Tristan, *GHDL*, ghdl.free.fr (Abgerufen am 20. 11. 2015).
- [9] IEEE (Hrsg.), *IEEE Std 1076-1987 – IEEE Standard VHDL Language Reference Manual*, New York, NY, USA, 1988.
- [10] IEEE (Hrsg.), *IEEE Std 1076-1993 – IEEE Standard VHDL Language Reference Manual*, New York, NY, USA, 1993.
- [11] IEEE (Hrsg.), *IEEE Std 1164-1993 – IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164)*, New York, NY, USA, 1993.
- [12] IEEE (Hrsg.), *IEEE Std 1076-2008 – IEEE Standard VHDL Language Reference Manual*, New York, NY, USA, 2009.
- [13] Kesel, Frank und Bartholomä, Ruben, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*, 3. Auflage, Oldenbourg Verlag, München, Deutschland, 2013.
- [14] Lange, Walter und Bogdan, Martin, *Entwurf und Synthese von Eingebetteten Systemen*, Oldenbourg Verlag, München, Deutschland, 2013.
- [15] Marwedel, Peter, *Eingebettete Systeme*, Springer, Heidelberg, Deutschland, 2008.
- [16] Menhorn, Benjamin und Slomka, Frank, *Entwurfsentropie – Ein Maß im Schaltungsentwurf*, in: 7th GI/GMM/ITG-Workshop Multi-Nature-Systems, Günzburg, Deutschland, 2009.
- [17] Menhorn, Benjamin und Slomka, Frank, *Design Entropy Concept – A Measurement for Complexity*, in: ESWEEK 2011 Compilation Proceedings, Seiten 285–294, ACM, New York, NY, USA, 2011.
- [18] Menhorn, Benjamin et al., *Digital Hardware Projects – A New Tool for Automated Complexity Analysis*, in: Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems, IEEE, Piscataway, NJ, USA, 2013.
- [19] Muchnick, Steven, *Advanced Compiler Design and Implementation*, Elsevier, Amsterdam, Niederlande, 2012.

- [20] Parr, Terence, *ANTLRv3*, www.antlr.org (Abgerufen am 20. 11. 2015), San Francisco, CA, USA.
- [21] Parr, Terence, *The Definitive Antlr Reference – Building Domain-Specific Languages*, The Pragmatic Bookshelf, Raleigh, NC, USA, 2007.
- [22] Parr, Terence, *ANTLR 3 License*, www.antlr3.org/license.html (Abgerufen am 20. 11. 2015), San Francisco, CA, USA, 2010.
- [23] Parr, Terence, *The Definitive Antlr 4 Reference*, The Pragmatic Bookshelf, Raleigh, NC, USA, 2013.
- [24] Parr, Terence und Harwell, Sam, *ANTLRv4*, www.antlr.org (Abgerufen am 20. 11. 2015), San Francisco, CA, USA.
- [25] Parr, Terence und Harwell, Sam, *ANTLR 4 License*, www.antlr.org/license.html (Abgerufen am 20. 11. 2015), San Francisco, CA, USA, 2012.
- [26] Reichardt, Jürgen und Schwarz, Bernd, *VHDL-Synthese*, 5. Auflage, Oldenbourg Verlag, München, Deutschland, 2009.
- [27] Safonov, Vladimir, *Trustworthy Compilers*, John Wiley & Sons, Hoboken, NJ, USA, 2010.
- [28] Stockmayer, Friedemann und Kreutzer, Hans, *Design using Standard Description Languages*, in: *The Electronic Design Automation Handbook*, Jansen, Dirk (Hrsg.), Seiten 86–145, Springer, Dordrecht, Niederlande, 2007.
- [29] The Eclipse Foundation (Hrsg.), *eclipse*, www.eclipse.org (Abgerufen am 25. 11. 2015), Ottawa, Kanada.
- [30] The Eclipse Foundation (Hrsg.), *Eclipse Public License – v 1.0*, www.eclipse.org/legal/epl-v10.html (Abgerufen am 25. 11. 2015), Ottawa, Kanada.
- [31] Vergnaud, Eric et al., *ANTLR 4 Documentation*, theantlr.guy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation (Abgerufen am 20. 11. 2015), San Francisco, CA, USA.
- [32] Vergnaud, Eric et al., *ANTLR v3 Documentation*, theantlr.guy.atlassian.net/wiki/display/ANTLR3/ANTLR+v3+documentation (Abgerufen am 20. 11. 2015), San Francisco, CA, USA.
- [33] Wilhelm, Reinhard et al., *Übersetzerbau – Syntaktische und semantische Analyse*, Band 2, Springer, Berlin, Deutschland, 2012.
- [34] Wirth, Niklaus, *Grundlagen und Techniken des Compilerbaus*, 2. Auflage, Oldenbourg Verlag, München, Deutschland, 2008.