Generierung eines VHDL-Parsers

Marius Vollmer

Studienarbeit S13-94 — AG SIV, UniDO

Inhaltsverzeichnis

Kapitel 1

Einleitung

> ***Ada**: n. A Pascal-descended language that has been made mandatory for Department of Defense software projects by the Pentagon. Hackers are nearly unanimous in observing that, technically, it is precisely what one might expect given that kind of endorsement by fiat; designed by committee, crockish, difficult to use, and overall a disastrous, multi-billion-dollar boondoggle (one common description is "The PL/I of the 1980s"). Hackers find Ada's exception-handling and inter-process communication features particularly hilarious. Ada Lovelace (the daughter of Lord Byron who became the world's first programmer while cooperating with Charles Babbage on the design of his mechanical computing engines in the mid-1800s) would almost certainly blanch at the use to which her name has latterly been put; the kindest thing that has been said about it is that there is probably a good small language screaming to get out from inside its vast, elephantine bulk.*
>
> — *The Jargon File 3.0.0*

Das Ziel dieser Arbeit war es, den ersten Schritt für eine allgemeine Einbindung von VHDL in MSIPC zu implementieren: die Umsetzung eines VHDL Programms in einen computergerechten abstrakten Syntaxgraphen. Der folgende Text gibt anhand von Beispielen einen Überblick über die wichtigsten Konzepte und Methoden des entstandenen Softwarepakets. Er soll vor allem einen Einstieg zum groben Verständnis der Quelltexte geben. Eine detailierte Beschreibung, wie sie zum eigentlichen Benutzen des Pakets nötig ist, findet sich in der "`libvaul` Referenz".

1.1 MSIPC und VHDL

MSIPC steht für "**M**ixed mode system **S**imulation for **I**mage **P**rocessing **C**ircuits" [1]. Es ist ein Rahmen für die Simulation von Bildverarbeitungshardware mit verschiedenen Beschreibungssprachen und Simulationstechniken. Das Hauptaugenmerk liegt dabei auf der automatischen Erzeugung eines C-Programms aus den unterschiedlichen Quellen, die die Schaltung beschreiben. Dieses C-Programm führt dann zur Laufzeit die Simulation durch.

VHDL (**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage) ist eine von Ada abgeleitete sehr allgemeine Beschreibungssprache, die in allen Phasen des Hardwaredesigns eingesetzt werden kann und hat sich in Europa als *der* Standard auf diesem Gebiet etabliert. Mit ihr kann sowohl die strukturelle Beschreibung einer Designhierarchie als auch die algorithmische Beschreibung des Verhaltens einzelner Komponenten erfolgen. Als Definition für VHDL diente der ANSI/IEEE Std 1076-1993 ("IEEE Standard VHDL Language Reference Manual" [2], kurz *LRM*). Begriffe aus dem *LRM* werden im folgenden *in diesem Zeichensatz* gesetzt.

Eine typische Vorgehensweise mit MSIPC ist z.B. die graphische Erstellung des Blockschaltbildes einer Schaltung aus der dann automatisch eine Netzliste erzeugt wird. Aus dieser Netzliste wird dann automatisch ein Rahmenprogramm generiert, das die Ablaufsteuerung der Simulation übernimmt und mit den passenden C-Funktionen zusammengebunden wird, die das Verhalten der grundlegenden Blöcke verwirklichen. Diese verhaltensbeschreibenden C-Funktionen könnten z.B. aus der Crosscompilierung eines Assemblerprogramms für einen Signalprozessor hervorgegangen sein oder auch direkt von Hand in C geschrieben werden.

Es besteht auch die Möglichkeit, die Simulation selbst heterogen ablaufen zu lassen: Das automatisch erzeugte C-Programm kann zur Laufzeit Daten mit anderen Simulatoren austauschen, so daß Teile der Simulation mit anderen Techniken durchgeführt werden können. Z.B. ist es möglich aus einer Hardwarebeschreibung in VHDL den strukturellen Teil herauszulösen und — über den Umweg einer Netzliste oder direkt — in das C-Rahmenprogramm zu übersetzen und die Verhaltensbeschreibung der anderen Blöcke von einem handelsüblichen VHDL Simulator durchführen zu lassen.

MSIPC ist für die Simulation von Schaltung der Bildverarbeitung gedacht, bei der für aussagekräftige Ergebnisse riesige Datenmengen verarbeitet werden müssen; ein Vollbild im PAL Format benötigt allein für die Luminanz 400 kByte. Um diese Datenmengen zu bewältigen, ist das Simulationsmodell von MSIPC der "synchrone Datenfluß". Jeder Block der Simulation wird durch ein verzögerungsfreies Schaltnetz mit Schieberegistern an den Ausgängen dargestellt. Das Simulationsmodell von VHDL ist dagegen die "diskrete Ereignissteuerung". Dadurch ist VHDL wesentlich allgemeiner, handelsübliche VHDL Simulatoren können aber unter den speziellen Bedingungen, für die MSIPC optimiert wurde, bei weitem nicht die nötige Effizienz liefern.

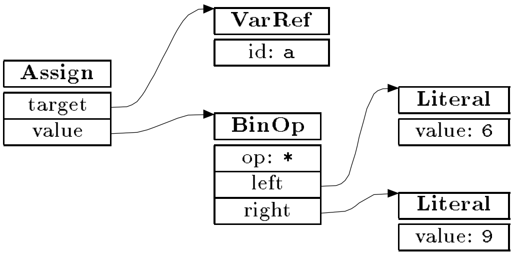
Trotzdem wäre es schön, wenn man sowiel wie möglich von VHDL in ein synchrones Schema pressen könnte und durch eine Crosscompilierung von VHDL nach MSIPC auch die Verhaltensbeschreibungen in VHDL mit in das erzeugte C-Programm aufnehmen könnte. Auf diese Weise würde man sich die zeitaufwendige Simulation von VHDL Beschreibungen mit einem "richtigen" VHDL Simulator sparen.

Als erster Schritt in Richtung eines VHDL nach C Umsetzers sollte mit dieser Studienarbeit ein Werkzeug zur Analyse von VHDL-Code erstellt werden. Das Ergebnis der Analyse sollte als abstrakter Syntaxgraph den nachfolgenden Stufen zur Erzeugung von Netzlisten oder C-Code zur Verfügung stehen. Die VHDL Syntax und Semantik, die von dem entwickelten Werkzeug — dem VHDL Parser — erfaßt werden kann, sollte möglichst komplett sein.

VHDL ist eine sehr umfangreiche und vielschichtige Sprache. Daher war es nicht möglich, im Rahmen dieser Studienarbeit die Vollständigkeit zu erreichen. Die Implementierung der fehlenden Sprachelemente — das sind vor allem *arrays*, *records*, *attributes* und *files* — ist aber vorbereitet und sollte auf keine prinzipiellen Hürden stoßen.

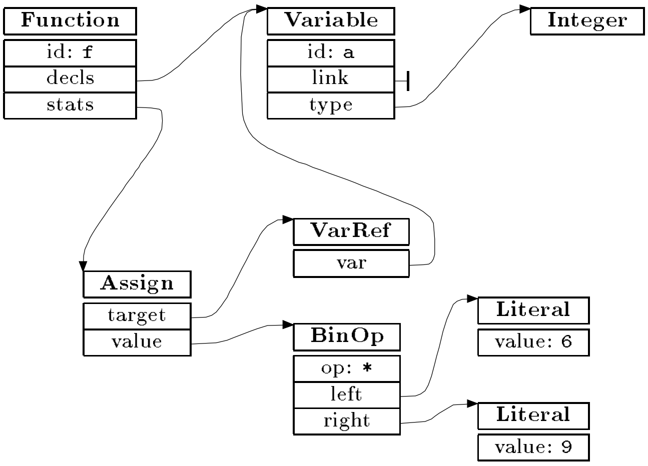
1.2 Syntaxgraphen

Die Hauptaufgabe des Werkzeugs ist die Bewältigung des ersten allgemeinen Schritts einer Compilierung: die Abbildung des menschenlesbaren Programmtextes in einen für Computer besser handhabbaren Graphen. Diese Abbildung kann in der Regel sehr direkt erfolgen. Der Programmtext `a = 6*9;` kann z.B. so dargestellt werden:



Hier wird die Referenz auf die Variable `a` einfach durch ihren Namen ausgedrückt. Es wäre aber sicherlich für die nachfolgenden Phasen des Compilers einfacher, wenn sie nicht erst in der Symboltabelle nach `a` suchen müßten, um z.B. den Typ von `a` zu erfahren. Die Referenz kann auch direkt zu dem Knoten im Graphen zeigen, der die Deklaration von `a` darstellt:

```
f() {
  int a;
  a = 6*9;
}
```



Das Ziel war es daher, soweit wie möglich die Semantik des VHDL Textes ebenfalls in den Graphen zu übertragen und dabei auch schon weitgehende Fehlerüberprüfungen anzustellen.

Es ist sinnvoll die Knotentypen in einer Hierarchie anzuordnen. Auf diese Weise können dann nicht nur für die Knoten selbst, sondern auch für die Kanten, die von einem Knoten ausgehen, Typinformationen angegeben werden. So kann z.B. ausgedrückt werden, daß die `left` und `right` Kanten eines `BinOp`-Knoten zu Knoten vom Typ `Expr` (oder eines davon abgeleiteten Typen) zeigen. Die Typinformationen können (wie in C++ auch) während des Kompilierens zusätzlich Fehler im Programm aufdecken und geben den Sinn der einzelnen Kanten auch besser wieder.

Für die Beschreibung und Implementierung der Knotentypen wurde ein extra Werkzeug erstellt. In seiner Notation könnten die oben verwendeten Typen so aussehen:

```
// Der Basistyp. Nichts Besonderes, aber alle Typen müssen von
// einem einzigen Basistypen abgeleitet sein.
//
node Node {
};

// Declarations können über link
// zusammengehängt werden. id ist keine Kante sondern
// ein Attribut. Attribute können beliebige C Typen haben.
//
node Declaration : Node {
  attr char *id;
  Declaration link;
};

// Scopes sind selber Declarations und
// enthalten eine Liste ihrer Declarations.
//
node Scope : Declaration {
  Declaration decls;
};

node Function : Scope {
  Statement stats
};

node Variable : Declaration {
  Type type;
};
```

```
// Typen werden allein durch den verwendeten Knoten
// unterschieden.
//
node Type : Node {
};

node Integer : Type {
};

node Statement : Node {
  Statement next;
};

node Assignment : Statement {
  Expr target;
  Expr value;
};

node Expr : Node {
};

node VarRef : Expr {
  Variable var;
};

node BinOp : Expr {
  attr char op;
  Expr left;
  Expr right;
};

node Literal : Expr {
  attr int value;
};
```

Das Werkzeug (`ctree`) erzeugt daraus eine passende C++-Klassenhierarchie und stellt Methoden zum Lesen und Schreiben von Teilgraphen zur Verfügung.

Aus der Sicht eines C++-Programms stellt sich der Graph als eine stark verzeigerte Sammlung von Strukturen (oder Klassen) dar. Die gerichteten Kanten des Graphen sind Zeiger auf solche Strukturen und die restlichen Attribute sind einfache Strukturelemente. Das Programm kann auch zur Laufzeit eine Typüberprüfung der Knoten vornehmen, mit Berücksichtigung der Typenhierarchie. Eine typische Funktion könnte so aussehen:

```
// gibt e in Postfixnotation aus
//
void postfix(nExpr e)
{
  if(e->is(nkBinOp)) {
    nBinOp b = nBinOp(e);
    postfix(b->left);
    postfix(b->right);
    printf("%c ", b->op);
  } else if(e->is(nkVarRef))
    printf("%s ", nVarRef(e)->var->id);
  else if(e->is(nkLiteral))
    printf("%d ", nLiteral(e)->value);
  else
    Fehler: kaputter Ausdruck, o.ä.
}
```

Die Namen von Knotentypen erscheinen im folgendem `in diesem Zeichensatz`.

1.3 VHDL Design Files, Units und Libraries

Der VHDL-Parser soll den Anwender der erzeugten Syntaxgraphen möglichst gut vom eigentlichen *design file* isolieren. Dazu gehört neben der Analyse des VHDL-Code auch die Verwaltung der *design units* in *design libraries*. Zu diesem Zweck wurde ebenfalls eine — im Vergleich zum Syntaxgraphen leicht überschaubare — C++ Klassensammlung entworfen. Dazu gehören Implementierungen der Basiskonzepte von VHDL, aber auch ein kleiner Rahmen für die Fehler- und Speicherverwaltung.

Die Klasse `vaul_design_unit`,[1] die das Konzept der *design unit* realisiert, beherbergt in erster Linie den aus dem entsprechenden VHDL-Code erzeugten Syntaxgraphen. Es ist die kleinste Einheit, die von einer `vaul_library` verwaltet werden kann. *Design files* werden durch die Klasse `vaul_design_file` vertreten, die den gesamten Apparat zur Analyse von VHDL-Code — den eigentlichen Parser — enthält. Der typische Ablauf z.B. bei der Generierung von C-Code aus einem *package* sieht so aus:

- Zuerst wird das *package* in eine *design library* eingefügt. Die VHDL Datei, die die Beschreibung des *packages* enthält, wird in einem Programm durch ein `vaul_design_file` dargestellt; die VHDL *design library*, in die das Package eingefügt werden soll, durch eine `vaul_library`. Der Aufruf der Funktion `vaul_design_file::insert` fügt alle *design units* aus dem *design file* in die *design library* ein.

---

[1] `vaul` steht für Vhdl Analyser and Utility Library.

- Ein zweites Programm (oder auch dasselbe) findet die dem *package* entsprechende `vaul_design_unit` über ihren Namen in der `vaul_library` wieder und beginnt den erzeugten Syntaxgraphen entlangzulaufen. Alle Querverweise zu anderen *design units* (z.B. dem allgegenwärtigen `package standard`) sind bereits aufgelöst und diese *design units* stellen sich dem Programm einfach als Teil des Syntaxgraphen dar. In den allermeisten Fällen erfährt das Programm gar nicht, aus welcher *design unit* oder *design library* welcher Teil des Graphen stammt.

Dieser Ablauf kann auch verfeinert werden. *Design units* können z.B. einzeln aus *design files* herausgelöst und direkt verarbeitet werden.

1.4 Lexer und Parser

Die Begriffe Lexer (oder Scanner) und Parser bezeichnen die Programmteile eines Compilers, die die lexikalische bzw. syntaktische Analyse des Programmtextes durchführen.

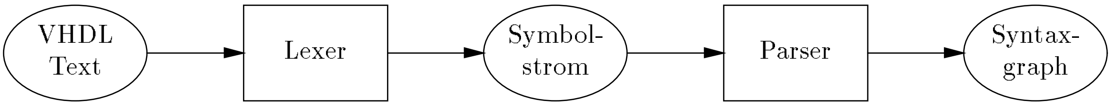


Abbildung 1.1: Lexer und Parser

Die lexikalische Analyse hat zur Aufgabe, die einzelnen Buchstaben des Quelltextes nach den Regeln der untersuchten Sprache zu Symbolen (*token*) zu gruppieren. Diese Symbole sind die Grundbausteine, mit denen die Grammatik einer Sprache beschrieben wird, unter anderem die Schlüsselwörter wie `for`, `return`; Operatoren wie `+`, `<=`; Konstanten wie `2`, `"Hello, World"` und einfache Namen wie `x`, `printf` (Beispiele aus C).

Die syntaktische Analyse versucht den Strom von Symbolen, den der Lexer liefert, auf die Grammatik der Sprache abzubilden. Dazu wird die Grammatik in Form von Regeln beschrieben, die angeben wie sich die Konstrukte der Sprache aus Symbolen und anderen Konstrukten zusammensetzen. Eine Regel könnte zum Beispiel lauten:

> “Ein Ausdruck kann aus einem Ausdruck, gefolgt von einem `+`-Symbol, gefolgt von einem Ausdruck bestehen.”

Um die endlose Rekursion dieser Regel zu beenden, die Ausdrücke durch Ausdrücke definiert, ist noch eine weitere Regel nötig, z.B.:

> “Ein Ausdruck kann auch aus einer Konstanten bestehen.”

Für einige Symbole steht genau fest, welche Buchstaben im Quelltext gestanden haben, wenn der Lexer ihr Auftauchen meldet. Sobald der Lexer z.B. den Buchstaben `+` sieht, gibt er das Symbol für den Operator `+` an den Parser weiter; mehr gibt es nicht zu sagen. Bei anderen Symbolen, die für eine ganze Klasse von Eingaben zuständig sind (z.B. Konstanten), ist es dagegen für den Parser wichtig, mehr zu erfahren. Daher gibt der Lexer nicht nur Symbole an den Parser weiter, sondern (Symbol, Wert) Paare. Sieht der Lexer z.B. die Eingabe `2`, so meldet er dem Parser das Paar (Konstante, 2).

Die Aufgabe des Parsers ist es nun, zu entscheiden, für welchen Teil der Eingabe welche Regel zuständig ist, so daß die komplette Eingabe abgedeckt ist.

Sobald der Parser eine der Regeln anwendet, die die Konstrukte der Sprache beschreiben, führt er die mit ihr verknüpften Aktionen aus. Diese Aktionen ermitteln für das Konstrukt, das die Regel beschreibt, einen neuen Wert aus den Werten der "Unterkonstrukte" und Symbole, aus denen es besteht. Eine Aktion, die Ausdrücke direkt berechnet, könnte für die erste Regel z.B. so lauten:

> "Der Wert dieses Ausdrucks ist der Wert des linken Ausdrucks plus dem Wert des rechten Ausdrucks."

Und für die zweite Regel:

> "Der Wert dieses Ausdrucks ist der Wert der Konstanten."

In formaler Schreibweise wird das alles viel übersichtlicher:

```
Ausdruck:
                Ausdruck ´+´ Ausdruck      { $$ = $1 + $3; }
        |       Konstante                  { $$ = $1; }
        ;
```

Der Teil in den geschweiften Klammern sind die Aktionen, die mit den Regeln in der gleichen Zeile verknüpft sind.

Es gibt zahlreiche Tools, die aus solchen formalen Beschreibungen der Grammatik zusammen mit den gewünschten Aktionen C-Funktionen erzeugen, die die syntaktische Analyse abdecken. Ebenso gibt es Tools für die lexikalische Analyse.

Der VHDL-Parser wurde mit solchen Tools (`flex` und `bison`) erstellt und erzeugt mit seinen Aktionen den gewünschten Syntaxgraphen.

Für eine gründliche Behandlung dieses Themas siehe [3].

Kapitel 2

Syntaxgraphen

Die eigentliche Aufgabe des VHDL-Parsers ist es, einen Syntaxgraphen zu erstellen, der die Syntax und auch ein wenig der Semantik des VHDL-Textes in computergerechter Form wiedergibt. Das Ziel war dabei, schon während dieser Phase der Compilierung möglichst viele Fehler zu erkennen und zu melden. Fehler, die im allgemeinen erst durch Ausführung von VHDL Anweisungen oder Auswertung von VHDL Ausdrücken entdeckt werden können, z.B. Verletzung der Einschränkungen eines *subtype*, blieben dabei allerdings unberücksichtigt.

Obwohl jede *design unit* getrennt analysiert und in einen Graphen umgewandelt wird, hat der entstehende Graph in der Regel Querverbindungen zu den Graphen anderer *design units*. Diese Verbindungen sind den Verbindungen innerhalb des Graphen völlig gleichwertig. Man muß sich sowohl während der Konstruktion als auch während der späteren Inspektion des Graphen nicht darum kümmern, zu welcher *design unit* ein Knoten gehört oder aus welcher *design library* er stammt. Das erledigen die grundlegenden Lese- und Schreibroutinen der Graphenverwaltung automatisch.

Die Verarbeitung des VHDL-Textes zu dem endgültigen Graphen geschieht in zwei Stufen: zuerst wird die Struktur des VHDL Programms sehr direkt in einen "Zwischengraphen" umgewandelt. Aus diesem Zwischengraphen wird dann durch Auflösen von Namen und anderen "Optimierungen" der eigentliche Graph produziert, der schließlich abgespeichert oder direkt von nachfolgenden Phasen des Compilers bearbeitet wird. Die "Optimierung" des Zwischengraphen geschieht eng verzahnt mit der syntaktischen Analyse und dem Erstellen des Zwischengraphen und ist im allgemeinen nicht scharf von den anderen Aktivitäten während des Parsens zu trennen.

Es wurde darauf geachtet, daß bei der "Optimierung" möglichst wenig interessante