technische universität
dortmund

Bachelorarbeit

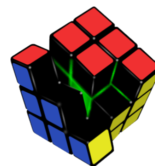
AspectVHDL:
Entwurf und Implementierung eines Aspektwebers für VHDL

Karl Stelzner

24. September 2013

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Matthias Meier

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de



Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 24. September 2013

Karl Stelzner

Zusammenfassung

Hardwarebeschreibungssprachen wie VHDL, mit denen digitale Systeme textuell beschrieben werden können, erlauben die Entwicklung von Hardware auf einem hohen, der Softwareentwicklung ähnlichen Abstaktionsniveau. Daher stellen sie ein vielversprechendes Gebiet für die Anwendung von Technologien aus dem Softwarebereich dar. Mit der Sprache AspectVHDL ist das von dort bekannte Konzept der aspektorientierten Programmierung auf die Hardwareentwicklung übertragen worden. In dieser Arbeit wird der zum praktischen Einsatz der Sprache notwendige Aspektweber entwickelt. Dazu wird zunächst ein bestehender VHDL-Parser gewählt und zur Verarbeitung von AspectVHDL erweitert. Auf Basis dessen wird der Aspektweber implementiert, der die Codemanipulationen am VHDL-Code durchführt, die durch AspectVHDL vorgesehenen sind. Die erarbeitete Lösung wird anschließend mit Hinblick auf ihre Praxistauglichkeit evaluiert.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung . . . 2
1.2 Motivation . . . 2
1.3 Übersicht . . . 4

2 Anforderungen **5**
2.1 Join Points . . . 5
2.1.1 Unterprogramme . . . 6
2.1.2 Introductions . . . 7
2.2 Pointcuts . . . 8
2.3 Aspekte . . . 9

3 Entwurf **11**
3.1 Parsen des AspectVHDL Codes . . . 11
3.1.1 GHDL . . . 12
3.1.2 VAUL . . . 13
3.1.3 vMAGIC . . . 13
3.2 Entwurf des Webers . . . 15

4 Implementierung **17**
4.1 Metaklassen . . . 17
4.1.1 Aspekte . . . 17
4.1.2 Advices . . . 19
4.1.3 Slices . . . 19
4.1.4 Pointcuts . . . 20
4.2 Parser . . . 21
4.2.1 Erweiterung des VHDL-Parsers . . . 21
4.2.2 Anpassung des Baum-Parsers . . . 23
4.3 Weber . . . 25
4.3.1 Analyse des VHDL-Codes . . . 25
4.3.2 Interpretation der Pointcuts . . . 28
4.3.3 Einweben des Aspectcodes . . . 30

5 Evaluation **35**
5.1 Laufzeit . . . 35
5.1.1 Szenario . . . 35
5.1.2 Ergebnisse . . . 36

5.2 MB-Lite Implementierung 37
5.3 Mögliche Erweiterungen 38
5.3.1 Bindung von Prozedur-Argumenten durch Pointcuts 39
5.3.2 Verbindung von Prozeduraufruf und Prozedur 39

6 Zusammenfassung und Ausblick **41**

Quellenverzeichnis **43**

Abbildungsverzeichnis **45**

1 Einleitung

VHDL ist eine der am weitesten verbreiteten Hardwarebeschreibungssprachen [1], also eine Sprache, mit der digitale Systeme textuell beschrieben werden können. Darüber hinaus erlaubt es VHDL, beschriebene Systeme zu simulieren und sogar automatisch zugehörige Schaltungsdesigns zu generieren (*Synthese*). Der Entwurf von komplexen Systemen wird dadurch stark vereinfacht – sie müssen nicht mehr Bauteil für Bauteil entworfen werden, sondern können auf einer höheren Abstraktionsebene für den Menschen verständlich beschrieben werden.

Da der Entwurf von Hardware somit einen ähnlichen Charakter wie das Programmieren von Software gewonnen hat, bietet es sich an, Designphilosophien aus dem Softwarebereich zu untersuchen und auf ihre Übertragbarkeit auf die Hardwareentwicklung zu prüfen. Eine davon ist die Idee der Modularisierung: Für die Struktur von Software ist es von Vorteil, wenn logisch zusammenhängende Teile eines Programms auch im Quelltext zusammenhängen. Dies erlaubt es, Projekte besser zu strukturieren und konfigurierbarer zu machen: Im Idealfall kann die gewünschte Funktionalität nach dem Baukastenprinzip maßgeschneidert zusammengestellt werden. VHDL bietet hierfür bereits einige Konzepte, wie etwa die Möglichkeit, Komponenten zu parametrisieren und so in verschiedenen Varianten wiederzuverwenden (*generics*).

Ein Paradigma, welches darauf abzielt, die Modularisierbarkeit von Programmiersprachen zu verbessern, ist die aspektorientierte Programmierung (*AOP*) [2]. Hierzu bietet sie Sprachkonstrukte, mit denen Programmcode, der normalerweise über mehrere Elemente des Projekts (z. B. mehrere Klassen) verteilt ist, in sogenannten Aspekten zusammengefasst werden kann. Damit können gemeinhin schwer kapselbare Angelegenheiten wie z. B. Logging modularisiert werden. Um diese Vorteile auf VHDL zu übertragen, wurde 2012 von der Arbeitsgruppe *Eingebettete Systemsoftware* ein Prototyp der Sprache *AspectVHDL* erstellt [3]. Durch diese werden bekannte, grundlegende Konzepte der aspektorientierten Programmierung in VHDL eingeführt. Zwar ist die Funktionsweise der Sprache bereits spezifiziert, es fehlt allerdings ein *Aspektweber*, um sie in der Praxis verwenden zu können. Dabei handelt es sich um ein Programm, das den in Aspekten gekapselten Code wieder in das restliche Projekt einwebt und so aus einem AspectVHDL-Projekt äquivalenten VHDL-Code erzeugt. Dieser kann dann durch bereits vorhandene Tools simuliert oder synthetisiert werden. Abbildung 1.1 stellt diese Funktionalität dar.

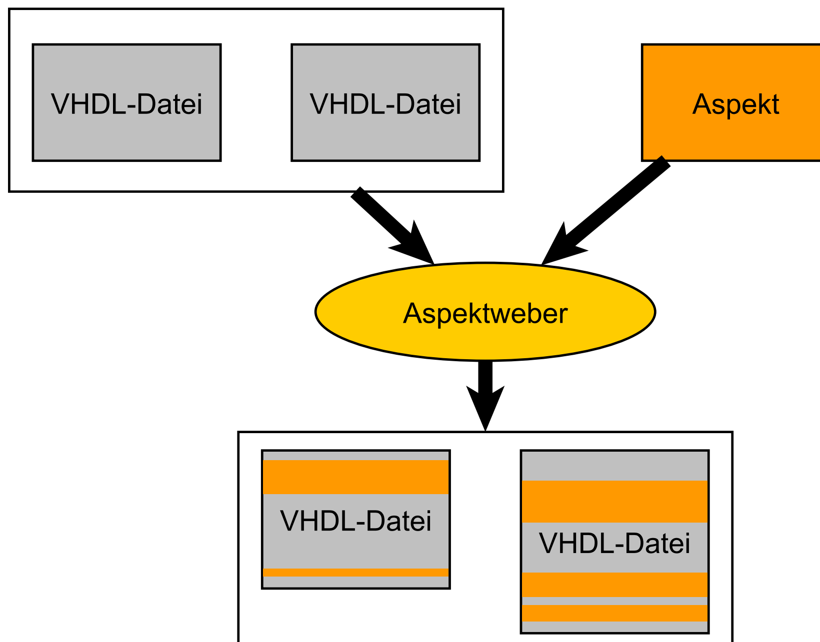


Abbildung 1.1: Visualisierung der Aufgabe des Aspektwebers

1.1 Zielsetzung

Ziel dieser Arbeit ist es, einen solchen Aspektweber zu entwerfen und zu implementieren. Dieser soll in der Lage sein, AspectVHDL-Code einzulesen, also sowohl die neuen AOP-Elemente als auch herkömmlichen VHDL-Code zu verarbeiten. Hierzu muss ein *Parser* für AspectVHDL erstellt werden. Anschließend soll der Aspektweber den eingelesenen Aspektcode gemäß der in [3] gegebenen Spezifikationen in den normalen VHDL-Code einweben. Hierzu ist es insbesondere notwendig, den VHDL-Code zu analysieren, um die Stellen zu finden, die zum Einweben verwendet werden können (*Join Points*). Zuletzt muss der Aspektweber das Ergebnis dieses Vorgangs in Form von VHDL-Dateien ausgeben.

Ein weiteres Ziel der Arbeit besteht in der Evaluation der gewonnenen Lösung und der Formulierung von möglichen Erweiterungen. Dabei gilt es vor allem, die Praxistauglichkeit des Aspektwebers zu untersuchen und mit den durch seine Implementierung gewonnenen Erfahrungen Ansätze für eine zukünftige Weiterentwicklung zu erarbeiten.

1.2 Motivation

Wie oben bereits erwähnt liegt der grundlegende Zweck von AOP darin, Elemente eines Projekts zu modularisieren, die normalerweise über mehrere logische Schichten des Pro-

gramms verteilt wären. Um zu untersuchen, ob solche sogenannten querschneidenden Belange (*Cross-Cutting Concerns*) auch in VHDL vorkommen, wurde im Rahmen des Artikels zu AspectVHDL die MB-Lite CPU, eine quelloffene VHDL-Implementierung von Xilinx' Microblaze Prozessor, analysiert. Diese bietet die Möglichkeit, optionale Erweiterungen des Prozessors hinzuzuschalten. Für drei von diesen Erweiterungen, nämlich Hardwareunterstützung für Multiplikations-, Shift- und Gleitkommaoperationen, wurde die Verteilung des für sie zuständigen Codes im Projekt untersucht. Abbildung 1.2 fasst die Ergebnisse der Untersuchung zusammen: Die Spalten stellen Quelldateien des Projektes dar, die eingefärbten Flächen markieren Codeabschnitte, die zu den optionalen Erweiterungen gehören. Es stellt sich heraus, dass der Quelltext für alle drei Features auf mehrere Standorte aufgeteilt ist. Insbesondere die optionale Gleitkommaeinheit ist über viele verschiedene Stellen im Projekt verteilt.

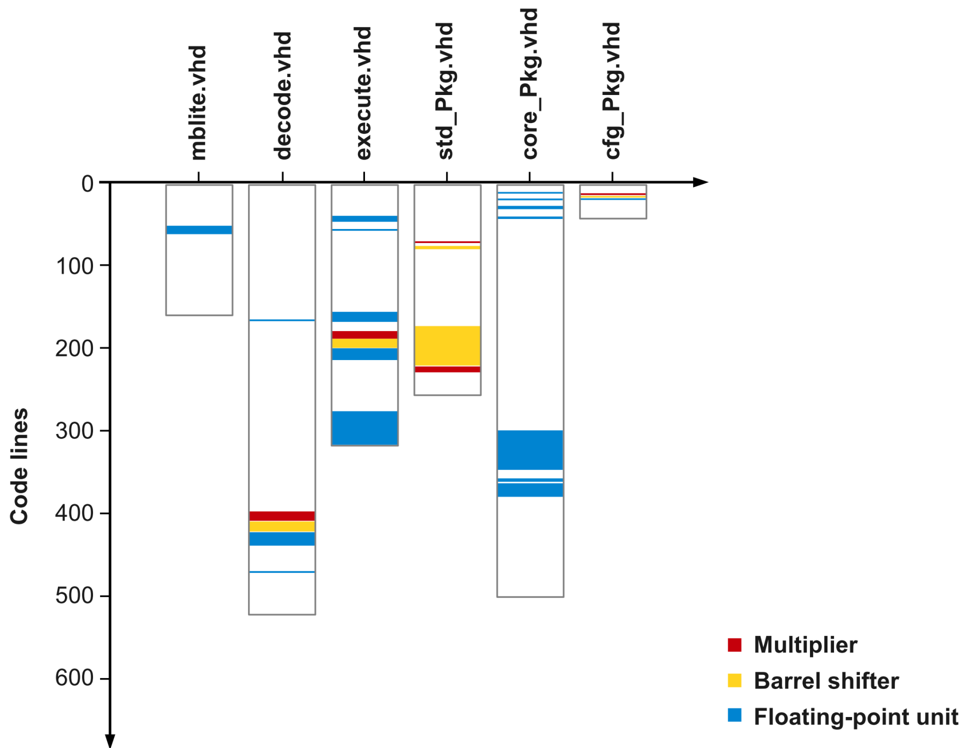


Abbildung 1.2: Darstellung der Verteilung des Codes von drei optionalen Erweiterungen der MB-Lite CPU (aus [3])

Es existiert somit auch in VHDL-Projekten eine Art von Cross-Cutting Concerns, die durch die Konzepte von AOP besser gekapselt werden könnten. Durch AspectVHDL könnten die drei betrachteten Elemente in Aspekten zusammengefasst und so die Verständlichkeit und Wartbarkeit des Projektes verbessert werden. Allgemein könnte AOP dabei helfen, die Modularisierbarkeit von VHDL zu verbessern und somit die Entwicklung von maßgeschneiderter Hardware zu vereinfachen.

Die Entwicklung des Aspektwebers im Rahmen dieser Arbeit ist ein notwendiger Schritt dorthin: Erst mit einer funktionsfähigen Implementierung kann AspectVHDL praktisch eingesetzt sowie untersucht und verbessert werden.

1.3 Übersicht

Im folgenden Kapitel werden zunächst die Anforderungen erläutert, die an den Aspektweber gestellt werden. Hierzu wird ein Überblick über die grundlegenden Elemente von AOP und den Aufbau von VHDL-Projekten gegeben. Anschließend werden die speziellen Sprachkonstrukte von AspectVHDL vorgestellt, die vom Aspektweber verarbeitet werden müssen. In Kapitel 3 folgt eine Darstellung der Analyse- und Entwurfsarbeiten, die vor der Implementierung des Aspektwebers durchgeführt wurden. Hierzu zählt die Wahl einer bestehenden Software zum Einlesen des Programmcodes und der Entwurf eines Algorithmus' für den Aspektweber.

Kapitel 4 befasst sich mit der Implementierung des Aspektwebers. Die Umsetzung der zuvor entworfenen Komponenten des Programms wird hier im Detail erläutert. In Kapitel 5 folgt anschließend eine Evaluation dieser Implementierung: Es wird untersucht, inwieweit die zuvor formulierten Anforderungen erfüllt sind und wie sich die Laufzeit des Aspektwebers verhält. Außerdem werden einige mögliche Erweiterungen von AspectVHDL vorgestellt. In Kapitel 6 werden die erlangten Erkenntnisse zusammengefasst und ein Ausblick auf die mögliche Zukunft des Projekts gegeben.

2 Anforderungen

Die Anforderungen, die an den Aspektweber gestellt werden, ergeben sich direkt aus den Spezifikationen von AspectVHDL [3]. Die dort vorgesehenen Sprachelemente müssen eingelesen, analysiert und ihre Semantik umgesetzt werden. Bevor diese Funktionalität erläutert wird, folgt zunächst ein Überblick über die Grundkonstrukte, die AspectVHDL von anderen aspektorientierten Sprachen wie *AspectJ* [2] oder *AspectC++* [4] übernommen hat:

- *Join Points*: Als Join Points werden die Punkte im VHDL Code bezeichnet, an denen durch Aspekte Veränderungen vorgenommen werden können. Welche das genau sind, wird in Abschnitt 2.1 erläutert.
- *Advice*: Ein Advice ist eine Anweisung an den Aspektweber, die dafür sorgt, dass ein bestimmter Codeabschnitt an einem oder mehreren Join Points eingewoben wird.
- *Slice*: In manchen Fällen ist ein solcher Codeabschnitt unabhängig von den Advices definiert, die ihn nutzen. In diesem Fall spricht man von einem Slice.
- *Pointcut*: Um zu bestimmen, an welchen Join Points eingewoben wird, verfügt jeder Advice über einen sogenannten Pointcut-Ausdruck. Hierbei handelt es sich um einen Ausdruck, der mit Hilfe spezieller Funktionen (siehe Abschnitt 2.2) eine Menge von Join Points spezifiziert.
- *Aspekt*: Aspekte bilden die grundlegenden Einheiten von aspektorientierten Programmiersprachen. In AspectVHDL können sie Advices, Slices und einige weitere Deklarationen (siehe Abschnitt 2.3) enthalten.

2.1 Join Points

Die erste Frage, die sich zu einer aspektorientierten Sprache stellt, ist, an welchen Stellen über Aspekte Code eingewoben werden kann, sprich, wo die *Join Points* liegen. Dies ist für den Aspektweber insofern relevant, als dass er die Join Points im VHDL-Code erkennen und in der Lage sein muss, an diesen Stellen Modifikationen vorzunehmen. Zum besseren Verständnis folgt zunächst ein Überblick über den Aufbau von VHDL-Code.

Die grundlegenden Elemente von VHDL-Projekten sind sogenannte *Entitäten* (*Entities*), die die Schnittstelle eines Hardwarebausteins darstellen [1]. Dies umfasst vor allem dessen Ein- und Ausgangssignale. Das eigentliche Verhalten der Hardware wird

durch *Architekturen* beschrieben, die jeweils eine Entität implementieren. Auf diese Weise können unterschiedliche Umsetzungen derselben Schnittstelle formuliert werden. Die Beschreibung des Hardwareverhaltens durch die Architekturen erfolgt durch parallele Anweisungen: Diese werden nicht sequentiell ausgeführt, sondern jede einzelne sorgt für die Erzeugung einer bestimmten Hardwarestruktur. Dementsprechend spielt auch deren Reihenfolge keine Rolle.

VHDL bietet verschiedene Arten von Anweisungen, mit denen ein Hardwareelement beschrieben werden kann: Zum einen können andere Entitäten instanziiert werden, um so eine hierarchische Struktur von Komponenten aufzubauen. Zum anderen können *Prozesse* definiert werden, die Folgen von *sequentiellen* Anweisungen enthalten. Hier können die aus Programmiersprachen bekannten Konstrukte wie Schleifen oder bedingte Anweisungen verwendet werden, um das Verhalten des Hardwareelements durch programmartige Strukturen zu beschreiben. Es ist allerdings zu beachten, dass diese Anweisungen später nicht durch einen Prozessor ausgeführt, sondern auch nur in Hardwareelemente umgewandelt werden. Die Prozesse werden immer dann aufgerufen, wenn sich eines der Signale ändert, die in ihrer Sensitivitätsliste vermerkt sind.

Um redundanten Code zu vermeiden, gibt es auch in VHDL die Möglichkeit, wiederverwendbare Unterprogramme und Datentypen zu spezifizieren. Diese können unter anderem in *Paketen* hinterlegt werden. Hierbei handelt es sich um Container für wiederverwendbare Definitionen, die von den Entitäten und Architekturen genutzt werden können. Um auch hier eine Trennung von Schnittstelle und Implementierung herzustellen, können Pakete in zwei Teile aufgeteilt werden: *Header* und *Body*. In diesem Fall enthält der Kopf des Pakets nach außen sichtbare Deklarationen, während im Körper die zugehörigen Implementierungen folgen.

An welchen dieser Stellen AspectVHDL ansetzt, um Code einzuweben, wird in den folgenden Abschnitten erläutert. Bei den Join Points von AspectVHDL wird, ähnlich wie bei anderen aspektorientierten Sprachen, zwischen der Modifikation von Unterprogrammen und dem Einfügen von zusätzlichen Deklarationen (*Introductions*) unterschieden.

2.1.1 Unterprogramme

In VHDL wird zwischen zwei Arten von Unterprogrammen – Funktionen und Prozeduren – unterschieden: Funktionen haben einen Rückgabewert und werden in der Regel für einfache und vor allem seiteneffektfreie Berechnungen verwendet. Prozeduren hingegen haben keinen Rückgabewert, sondern bekommen Signale übergeben, die in der Prozedur verwendet und modifiziert werden können.

Da Funktionen relativ einfach durch äquivalente Prozeduren ersetzt werden können und die Modifikation von Prozeduren auf Grund des fehlenden Rückgabewertes einfacher ist, beschränkt sich AspectVHDL auf die Nutzung von Prozeduraufrufen als Join Points. Diese können durch Aspekte auf ähnliche Weise modifiziert werden, wie das auch mit

Unterprogrammen in AspectJ oder AspectC++ geschieht: Es können zusätzliche Anweisungen vor (*before*), nach (*after*) oder anstatt (*around*) der Ursprungsprozedur ausgeführt werden. Im letzten Fall kann der ursprüngliche Code über das Schlüsselwort *proceed* an beliebiger Stelle aufgerufen werden.

In den zugehörigen Advices werden dazu eine Parameterliste und die einzuwebenden Anweisungen angegeben. Die Parameter können als lokale Variablen im gegebenen Codeabschnitt genutzt werden, um mit den Kontextinformationen der Ursprungsprozedur zu arbeiten.

Ein wichtiges Detail besteht an dieser Stelle darin, dass in AspectVHDL der Aufruf einer Prozedur als Join Point dient, nicht die Prozedur selbst. Dies ist darin begründet, dass bei der Synthese von VHDL für jeden Prozeduraufruf eine eigene Hardwareeinheit erstellt wird, die diese Prozedur umsetzt. Im Kontrast dazu wird bei Programmiersprachen wie C bei jedem Aufruf einer Methode derselbe Abschnitt Binärcode ausgeführt. Dementsprechend ergibt es Sinn, die einzelnen Hardwarebausteine auch getrennt durch Aspekte modifizieren zu können.

2.1.2 Introductions

Zusätzlich zu der Modifikation von Unterprogrammen können mit AspectVHDL an bestimmten Stellen zusätzliche Codeabschnitte (*slices*) eingefügt werden. Da man es hier nicht mit sequentiellen Anweisungen, sondern nur mit Deklarationen und parallelen Anweisungen zu tun hat, muss nicht spezifiziert werden, an welcher Stelle genau eingewoben wird. Eine Unterscheidung zwischen *before*, *after* oder *around* fällt an dieser Stelle also weg; die zusätzlichen Codeabschnitte werden einfach an beliebiger Stelle hinzugefügt.

2.1.2.1 Architekturen

Wie oben erwähnt können Architekturen Deklarationen und parallele Anweisungen erhalten, um das Verhalten von Hardwareelementen zu beschreiben. In AspectVHDL können hier über Aspekte weitere Elemente, wie Typdeklarationen, Signalzuweisungen und Prozesse hinzugefügt werden.

2.1.2.2 Typen

Zwei wichtige Arten von Datentypen, Enumerations und Records, können in AspectVHDL durch Aspekte erweitert werden. Enumerations, die in VHDL ähnlich wie z. B. in Java funktionieren, lassen sich über Aspekte um weitere Werte erweitern. Records erfüllen in VHDL eine ähnliche Rolle wie Structs in C: Es handelt sich um komplexe Datentypen, die eine Menge von Attributen verwalten. Diese Datentypen können durch Aspekte um zusätzliche Attribute erweitert werden. Da diese Typen auch in den Schnittstellen von Entitäten verwendet werden können, ist es auf diese Weise möglich, strukturelle Erweiterungen an Komponenten vorzunehmen.

2.1.2.3 Prozess Sensitivitätslisten

Wie bereits erwähnt verfügen Prozesse über eine Liste von Signalen, die bestimmt, wann diese aufgerufen werden. AspectVHDL erlaubt es, diese Sensitivitätslisten um zusätzliche Signale zu erweitern, um den Prozess so von weiteren Signalen abhängig zu machen.

2.2 Pointcuts

Pointcut-Ausdrücke werden benutzt, um die Menge der Join Points zu spezifizieren, die durch einen Advice beeinflusst werden. Hierzu wird eine Reihe von Funktionen zur Verfügung gestellt, die auf beliebige Art und Weise durch die booleschen Funktionen *and*, *or* und *not* sowie durch den Einsatz von Klammern kombiniert werden können. Diese Pointcut-Funktionen selektieren, abhängig von den gegebenen Parametern, jeweils eine bestimmte Menge von Join Points:

- *within(package_name)*: Selektiert alle Join Points, die sich in dem Paket mit dem angegebenen Namen befinden.

- *within(arch_name of entity_name)*: Selektiert alle Join Points, die sich innerhalb der über Namen und implementierter Entität spezifizierten Architektur befinden.

- *architecture(arch_name of entity_name)*: Selektiert die angegebene Architektur als Join Point (siehe Abschnitt 2.1.2.1).

- *type(type_name)*: Selektiert die Datentypen mit dem angegebenen Namen (siehe Abschnitt 2.1.2.2).

- *process(process_name)*: Selektiert die Prozesse mit dem angegebenen Namen (siehe Abschnitt 2.1.2.3).

- *call(procedure_name(arg_type1, arg_type2))*: Selektiert alle Prozeduraufrufe mit dem angegebenen Prozedurnamen und den spezifizierten Parametertypen. Die Argumentliste kann auch mit der Wildcard ’*’ ersetzt werden, wenn alle Prozessaufrufe dieses Namens selektiert werden sollen.

- *args(arg_type1, arg_type2)*: Selektiert alle Prozeduraufrufe mit den angegebenen Parametertypen, unabhängig vom Prozedurnamen.

Zusätzlich wird bei allen Bezeichnern die Möglichkeit geboten, ein ’*’ als Wildcard anzugeben, wenn unabhängig vom Namen alle Pointcuts dieser Art ausgewählt werden sollen. Auf diese Weise ist es möglich, eine relativ präzise Auswahl der Join Points vorzunehmen, z.B. alle in einem Paket definierten Datentypen, oder einen bestimmten Prozess in einer bestimmten Architektur auszuwählen.

2.3 Aspekte

Wie in anderen aspektorientierten Sprachen auch werden Aspekte in AspectVHDL benutzt, um Advices und Slices zu gruppieren. Somit befinden sich alle durch AspectVHDL hinzugefügten Sprachkonstrukte auch physisch innerhalb von Aspekten. Zusätzlich dazu können Aspekte Deklarationen enthalten, die sonst in Paketen vorkommen, also z.B. Prozeduren, Funktionen und Typen. Auf diese Weise können VHDL-Elemente erstellt werden, die dann in den Slices verwendet werden können. Aufgabe des Aspektwebers ist es an dieser Stelle, dafür zu sorgen, dass diese an den entsprechenden Stellen auch zur Verfügung stehen.

Die Spezifikationen von AspectVHDL sehen außerdem vor, dass Aspekte (und nur Aspekte) in speziellen Dateien mit der Endung *.avhd* stehen müssen. Dies dient zur Trennung von normalem VHDL-Code und Aspektcode und zum leichteren Auffinden der Aspekte durch den Aspektweber.

3 Entwurf

Um eine erfolgreiche Umsetzung des Aspektwebers zu garantieren, ist es nötig, einige wichtige Entwurfsentscheidungen im Voraus zu treffen. Hierfür ist es hilfreich, dessen Ablauf zunächst in drei wesentliche Schritte zu unterteilen:

1. Parsen: Um den gegebenen AspectVHDL-Code bearbeiten zu können, muss er zunächst durch einen Parser eingelesen und in eine Sammlung von Programmobjekten umgewandelt werden. Durch dieses Modell müssen sowohl die neuen AOP-Konstrukte als auch der herkömmliche VHDL-Code repräsentiert werden.

2. Weben: Anhand der in Abschnitt 2 beschriebenen Anforderungen müssen die gegebenen Aspekte in den restlichen Code eingewoben werden. Das vom Parser erstellte Modell des AspectVHDL-Codes ist also so zu verändern, dass ein äquivalentes Modell von reinem VHDL-Code entsteht.

3. Codegenerierung: Der letzte Schritt besteht darin, aus dem zunächst nur im Aspektweber bestehenden Modell wieder VHDL-Code zu generieren.

Im folgenden Kapitel werden die Vorarbeiten dargestellt, die zur Implementierung dieser Vorgänge nötig sind. Abschnitt 3.1 befasst sich mit dem Entwurf des Parsers, während in Abschnitt 3.2 die Grundidee des Webers vorgestellt wird.

3.1 Parsen des AspectVHDL Codes

Die erste Komponente, die benötigt wird, um den AspectVHDL-Code auf irgendeine Weise zu modifizieren, ist ein Parser. Dessen Aufgabe besteht darin, den Quelltext in einen Objektbaum umzuwandeln, dessen Knoten die einzelnen Sprachelemente repräsentieren. Der Parser muss hierbei nicht nur in der Lage sein, die neuen AOP-Konstrukte von AspectVHDL zu verarbeiten, sondern auch herkömmlichen VHDL-Code einlesen können. Parser basieren in der Regel auf der Grammatik, also dem Regelwerk der Sprache, die sie verarbeiten. Da VHDL an sich bereits eine recht umfangreiche Sprache ist, deren Grammatik über 200 Regeln enthält [5], ist es im Rahmen dieser Arbeit nicht realistisch, den benötigten Parser von Grund auf neu zu schreiben.

Aus diesem Grund muss der Aspektweber auf einem bereits bestehenden VHDL-Parser aufgebaut werden. Dieser bestimmt nicht nur die Programmiersprache, in der der Weber zu erstellen ist, sondern auch die Art und Weise, auf die der Weber mit dem VHDL-Code interagiert. Die wichtigste Entscheidung der Entwurfsphase besteht daher in der

Wahl eines bestehenden VHDL-Parsers, der für die Anforderungen des Aspektwebers möglichst gut geeignet ist. Hierbei sind folgende Kriterien entscheidend:

- Erweiterbarkeit: Der Parser muss die Möglichkeit bieten, die Grammatik, auf der er basiert, zu erweitern. Nur so ist es möglich, die zusätzlichen AspectVHDL-Sprachelemente hinzuzufügen.

- Verständlichkeit: Um den Aspektweber implementieren zu können, ist ein relativ tiefes Verständnis der Funktionsweise des Parser erforderlich. Nur so kann seine Funktionalität erweitert und der von ihm generierte Objektbaum durchschaut werden. Daher ist ein verständlicher Aufbau des Parsers entscheidend.

- Modifizierbarkeit des Objektbaums: Für die spätere Bearbeitung durch den Aspektweber ist es wichtig, dass der durch den Parser erstellte Objektbaum möglichst leicht zu modifizieren ist. Insbesondere muss es möglich sein, die Join Points zu finden und an den entsprechenden Stellen neue Codeabschnitte einzufügen.

- Codegenerierung: Nach dem Abschluss des Webvorgangs muss der Objektbaum wieder in VHDL-Code umgewandelt werden. Idealerweise sollte der Parser hierfür bereits Unterstützung bereitstellen.

- Verlässlichkeit: AspectVHDL kann nur dann eine in der Praxis nützliche Sprache werden, wenn der Aspektweber und der dazugehörige Parser verlässlich arbeiten. Es sind daher solche Parser vorzuziehen, die ausgiebig getestet wurden und den VHDL-93 Standard unterstützen.

Für die Verwendung für AspectVHDL wurden verschiedene Parser in Betracht gezogen, deren Vor- und Nachteile im Folgenden kurz erläutert werden. Die Auswahl ist dadurch eingeschränkt, dass viele der Tools für VHDL kommerziell sind und ihr Quelltext somit nicht offenliegt. Da die erforderlichen Modifikationen so nicht möglich sind, kommt solche Software nicht in Frage.

3.1.1 GHDL

GHDL [6] ist der wohl am weitesten verbreitete quelloffene Simulator für VHDL und umfasst dementsprechend natürlich auch einen Parser. Dies macht das Projekt zu einem offensichtlichen Kandidaten für die Verwendung im Aspektweber – auf Grund der hohen Verbreitung ist eine Zuverlässigkeit gewährleistet, die auch für AspectVHDL wünschenswert wäre.

Problematisch ist jedoch, dass GHDL in ADA geschrieben ist: Eine Verwendung für diese Arbeit würde also bedeuten, dass auch der Aspektweber in ADA geschrieben werden müsste. Dies würde dessen Entwicklung auch auf lange Sicht deutlich erschweren. Des Weiteren müsste der GHDL-Parser vom Rest des Programms, insbesondere vom Simulator getrennt werden. Inwieweit das überhaupt möglich ist, ist nur schwer abzuschätzen. Insgesamt erscheint die Umsetzung des Aspektwebers in einer unbekannten Sprache zu aufwändig und zu risikoreich.

3.1.2 VAUL

Eine weitere Alternative zum Parsen von AspectVHDL besteht in VAUL, einem VHDL-Parser, der bereits 1996 an der Fakultät Elektrotechnik der TU Dortmund geschrieben wurde [7] und heute in dem quelloffenen VHDL Simulator FreeHDL [8] verwendet wird. Da VAUL in C++ geschrieben ist, kommt der Parser für die Verwendung im Aspektweber in Frage. Leider scheint das Projekt nicht aktiv weiterentwickelt zu werden – insbesondere wurde der Parser nicht ausführlich getestet.

Ein weiteres Problem besteht in der Art und Weise, wie der Parser erstellt wurde: Es wurde ein Parsergenerator verwendet, der den C++-Code zum Parsen von VHDL automatisch aus der entsprechenden Grammatik erstellt hat. Dies hat zur Folge, dass der Quelltext des Parsers für Menschen nur sehr schwer lesbar und noch schwerer zu erweitern ist. Da auch der Entwicklungsprozess über den Parsergenerator nicht nachvollzogen werden kann, ist eine effiziente Erweiterung des Parsers kaum möglich.

Auch die Verständlichkeit des restlichen Programms lässt zu wünschen übrig – beispielsweise sind für die Datentypen, aus denen der Objektbaum konstruiert wird, über 200 *structs* in einer einzigen unkommentierten Datei definiert. Eine Verwendung von VAUL ist somit abzulehnen.

3.1.3 vMAGIC

vMAGIC, kurz für *VHDL Manipulation And Generation InterfaCe* ist ein Java Framework, das 2008 an der Universität Paderborn entstanden ist [9]. Es wurde speziell für Anwendungen geschrieben, die VHDL-Code einlesen, manipulieren und generieren. Der Funktionsumfang der Software ist in Abbildung 3.1 dargestellt: VHDL-Code kann nicht nur geparst, sondern auch aus dem internen Objektbaum wieder generiert werden. Dieses Feature kann eingesetzt werden, um nach dem Einweben der Aspekte wieder Quelltext zu generieren. Da sich vMAGIC, wie im Folgenden erläutert, recht leicht erweitern lässt und eine komfortable Interaktion mit dem Objektbaum erlaubt, wurde die Software für die Verwendung im Aspektweber ausgewählt.

Die Verarbeitung des VHDL-Codes durch vMAGIC geschieht in zwei Schritten: Zunächst wird ein Syntaxbaum erstellt, der die einzelnen Elemente des Quelltextes eins zu eins widerspiegelt. Dies ist in etwa das Ergebnis, das die meisten anderen Parser ebenfalls liefern würden; es macht die spätere Bearbeitung durch den Aspektweber allerdings relativ aufwändig. Beispielsweise sind sowohl Name, Deklarationen und Anweisungen einer Architektur allesamt Kinder des entsprechenden Architekturknotens im Syntaxbaum. Bei jedem Zugriff darauf muss manuell unterschieden werden, welcher Knoten welche Bedeutung hat.

Um die Codemanipulation einfacher zu gestalten, führt der vMAGIC-Parser daher einen zweiten Schritt durch: Der Syntaxbaum wird in eine Sammlung von sogenannten *Metaob-*

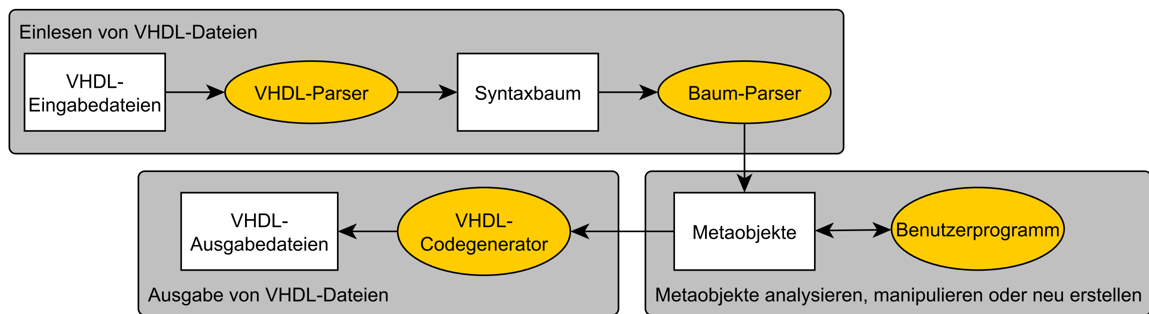


Abbildung 3.1: Visualisierung des Ablaufs von vMAGIC (nach [9])

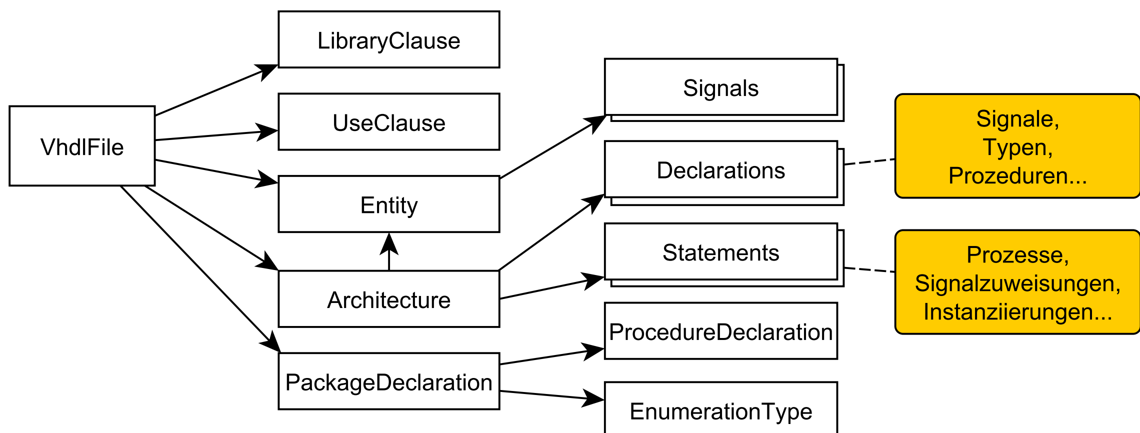


Abbildung 3.2: Ausschnitt aus einem möglichen von vMAGIC erzeugten Objektbaum (nach [10])

jekten umgewandelt. Dabei handelt es sich um Java-Objekte, die den VHDL-Konstrukten entsprechen und alle dazugehörigen Informationen enthalten. vMAGIC liefert hierfür eine umfangreiche Sammlung von Metaklassen, die einen komfortablen, objektorientierten Umgang mit den VHDL-Objekten erlauben. Die Attribute eines *Architecture*-Objekts umfassen beispielsweise den Namen der Architektur, eine verkettete Liste von deren Deklaration, eine Liste von deren Anweisungen und sogar eine Referenz auf das *Entity*-Objekt zu der Entität, die durch die Architektur implementiert wird. Ein Ausschnitt aus einem beispielhafter Objektbaum ist in Abbildung 3.2 dargestellt. Der Zugriff auf die VHDL-Elemente wird so deutlich vereinfacht: Durch einen einfachen Listendurchlauf können z. B. alle Architekturen in einer VHDL-Datei gefunden werden.

Für beide Parserschritte wurde bei vMAGIC, ähnlich wie bei VAUL, ein Parsergenerator verwendet, namentlich *ANTLR*. Anders als bei VAUL kann hier jedoch der Entwicklungsprozess nachvollzogen werden, da die ursprünglichen Grammatiken gegeben sind

und modifiziert werden können. Mit deren Hilfe kann über ANTLR eine veränderte Version des Parsers generiert werden. Auf diese Weise lässt sich der vMAGIC-Parser mit überschaubarem Aufwand erweitern.

Ein Nachteil von vMAGIC ist die Tatsache, dass es sich um keine weit verbreitete Software handelt und ihre Zuverlässigkeit vorab nicht garantiert werden kann. Da der Parser im Vorabtest aber eine Reihe von Projekten korrekt eingelesen hat, ist davon auszugehen, dass er zuverlässig arbeitet.

3.2 Entwurf des Webers

Die eigentliche Arbeit des Aspektwebers besteht natürlich darin, die gegebenen Aspekte in den VHDL-Code einzuweben. Bevor allerdings Code eingewoben werden kann, muss ermittelt werden, an welchen Join Points dies geschehen soll. Dies wird durch die Möglichkeit erschwert, dass die Pointcut-Funktionen zu beliebigen booleschen Ausdrücken kombiniert werden können (vgl. Abschnitt 2.2). Dadurch ist es nicht möglich, für allgemeine Pointcuts Vorhersagen darüber zu treffen, wo sich die von ihnen abgedeckten Join Points befinden. Es ist dagegen nur möglich, für einzelne Join Points zu *entscheiden*, ob sie in einem Pointcut liegen oder nicht. Das hat zur Folge, dass alle Join Points bekannt sein müssen, bevor ermittelt werden kann, welche davon in einem Pointcut liegen und welche nicht. Die Arbeit des Aspektwebers lässt sich somit in drei Teilprobleme aufteilen:

1. VHDL-Analyse: Das Finden aller Join Points im gegebenen VHDL-Code
2. Pointcut Auswertung: Die Entscheidung, ob ein bestimmter Joint Point in einem gegebenen Pointcut liegt
3. Aspektweben: Einweben eines Codeabschnitts in einen bestimmten Join Point

Können diese Teilprobleme gelöst werden, kann damit ein sehr simpler Algorithmus für den Aspektweber erstellt werden:

```
1 Parse Aspekte und VHDL-Code
2 Finde alle Join Points
3 Für alle Advices:
4     Für alle Join Points:
5         Wenn Join Point im Pointcut des Advice:
6             Webe Advice in Join Point
7 Gib VHDL-Code aus
```

Abbildung 3.3: Algorithmus für den Aspektweber in Pseudocode

Dieser Algorithmus stellt die Grundlage für die in Abschnitt 4.3 vorgestellte Implementierung des Aspektwebers dar.

4 Implementierung

Im folgenden Kapitel wird die auf Basis der im vorigen Kapitel angestellten Überlegungen erfolgte Implementierung des Aspektwebers erläutert. Diese ist grob nach den zu Beginn von Kapitel 3 angegebenen Komponenten aufgebaut. Um das Parsen des AspectVHDL-Codes zu ermöglichen, wurde zunächst ein Metamodell erstellt, das den Aspect-Code in Form von Java-Objekten darstellen kann. Dessen Aufbau wird in Abschnitt 4.1 erläutert. Zunächst folgt in Abschnitt 4.1 eine Übersicht über die zur Repräsentation des AspectVHDL-Codes erstellten Metaklassen. Anschließend wird in Abschnitt 4.2 dargestellt, wie der vMAGIC Parser erweitert wurde, um auch AspectVHDL-Code zu verarbeiten.

Zuletzt folgt in Abschnitt 4.3 eine Beschreibung der Funktionsweise des eigentlichen Aspektwebers auf Basis des Algorithmus aus Abschnitt 3.2. Der Codegenerierung wurde keine eigene Sektion gewidmet, da hier die Funktionalität von vMAGIC verwendet wird. Die Software ist in der Lage, aus den Metaobjekten automatisiert VHDL-Code zu erzeugen, der sogar recht gut lesbar ist. Daher waren hier keine Modifikationen nötig.

4.1 Metaklassen

Wie in Abbildung 3.2 dargestellt, erstellt der vMAGIC-Parser aus VHDL-Code Metaobjekte, welche die Elemente des VHDL-Codes repräsentieren. Das bedeutet, dass für die neuen AspectVHDL-Konstrukte Metaklassen implementiert werden müssen, die diese auf entsprechende Art und Weise verkörpern. Ziel ist es hierbei insbesondere, für eine gute Erweiterbarkeit des Metamodells zu sorgen, da die jetzige Version von AspectVHDL noch weiterentwickelt werden soll. Die Umsetzung der Metaklassen muss also z. B. die Möglichkeit bieten, weitere Join Point- oder Pointcut-Typen hinzuzufügen. Die für den Aspektweber gewählte Lösung ist in Abbildung 4.1 als Klassendiagramm dargestellt. Alle Klassen, mit Ausnahme von *Aspect* und den Pointcuts, sind in dem Paket *aspectvhdl.metaclasses* implementiert.

4.1.1 Aspekte

An oberster Stelle steht die Klasse *Aspect*, die einen Aspekt repräsentiert. Sie erbt von der von vMAGIC zur Verfügung gestellten abstrakten Klasse *LibraryUnit*, die als Oberklasse für alle VHDL-Elemente dient, die im Quelltext auf oberster Ebene stehen können (*Bibliothekselemente*). Damit ist sichergestellt, dass Aspekte analog zu Entitäten oder

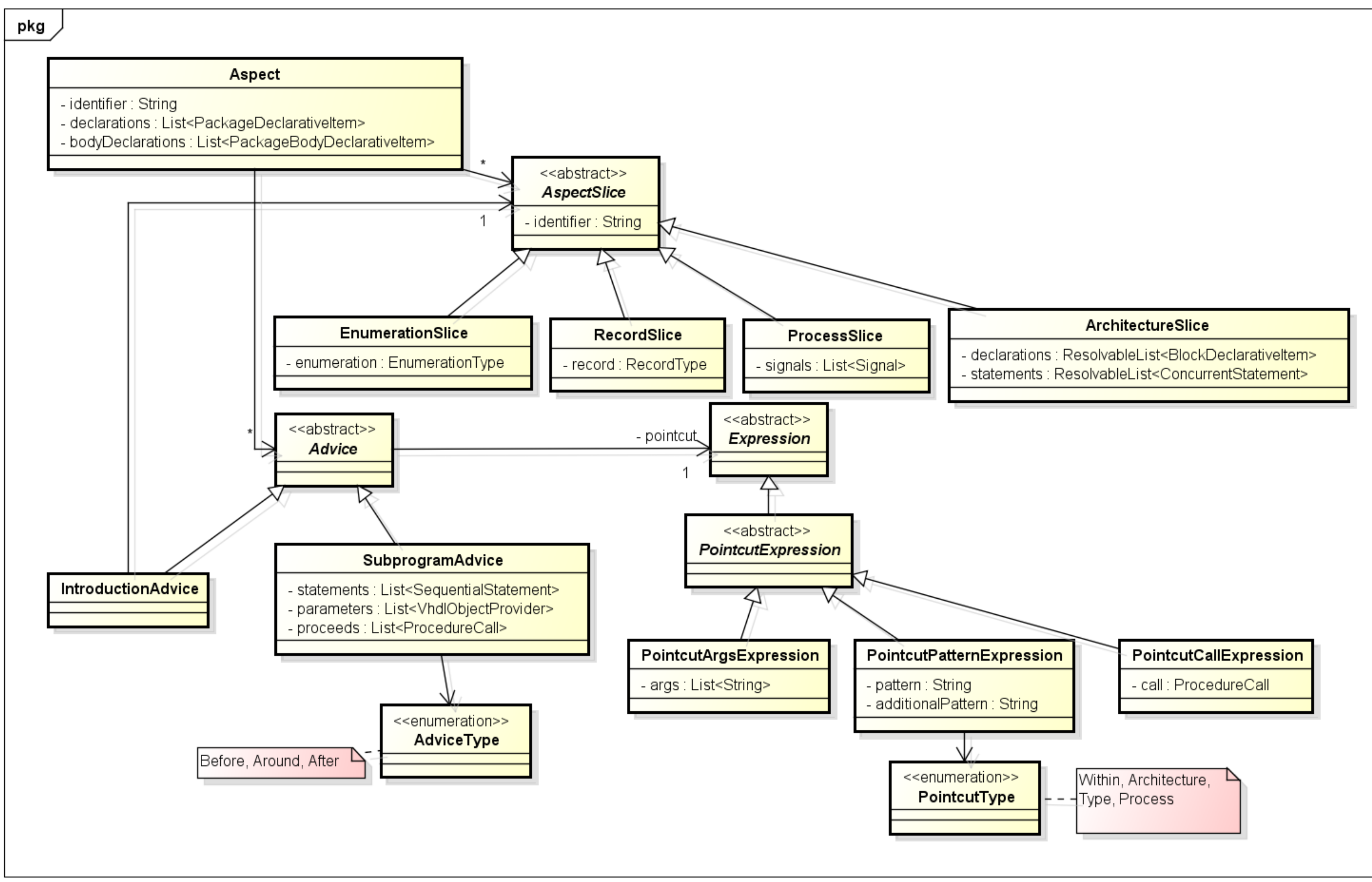


Abbildung 4.1: Klassendiagramm zu den für AspectVHDL erstellten Metaklassen

Architekturen als Hauptelemente einer VHDL-Datei verwaltet werden können (vgl. Abbildung 3.2). Es hat allerdings auch zur Folge, dass die Klasse, anders als die restlichen neu implementierten Metaklassen, in dem Package *de.upb.hni.vmagic.libraryunit* liegen muss, da dort eine mit Sichtbarkeit *Package* definierte Methode zu implementieren ist.

Um die in Abschnitt 2.3 formulierten Anforderungen zu erfüllen, verfügen Aspekte über eine Liste von Advices und Slices, die die im Aspekt deklarierten AspectVHDL-Elemente umfassen. Zusätzlich verwaltet die Aspektklasse die im Aspekt deklarierten herkömmlichen VHDL-Elemente wie Typen, Komponenten, etc.. Da diese in VHDL zum Teil in Paketdeklarationen und zum Teil im Paketkörper stehen müssen, werden sie in zwei Listen gesammelt, die die entsprechenden Typen aufnehmen können. Dies wird durch die beiden Interfaces *PackageDeclarativeItem* und *PackageBodyDeclarativeItem*) ermöglicht, die in vMAGIC von allen Klassen implementiert werden, die VHDL-Elemente repräsentieren, welche im Paketkopf bzw. -körper stehen können.

4.1.2 Advices

Wie in Kapitel 2 erläutert, gibt es in AspectVHDL zwei grundlegende Arten von Advices: Introduction-Advices und Prozedur-Advices beziehen. Dies schlägt sich in der Umsetzung der Metaklassen nieder: Die beiden Typen werden durch die Klassen *IntroductionAdvice* und *SubprogramAdvice* implementiert, welche von der abstrakten Oberklasse *Advice* erben. Auf diese Weise wird es ermöglicht, an dieser Stelle später zusätzliche Typen von Advices zu implementieren. Da alle Advices einen Pointcut verwalten, ist dieser in der Oberklasse als Attribut vom Typ *Expression* (mehr dazu in Abschnitt 4.1.4) umgesetzt.

Introduction-Advices, die sich ja immer auf einen Slice beziehen, enthalten zusätzlich eine Referenz auf einen solchen. Auf der anderen Seite muss von der Klasse *SubprogramAdvice* der einzuwebende Prozedurabschnitt verwaltet werden. Dieser wird, genauso wie das in vMAGIC auch generell bei Prozeduren erfolgt, als Liste sequentieller Ausdrücke repräsentiert. Zusätzlich wird die Liste der verwendeten Parameter und der Typ des Advice gespeichert. Letzteres geschieht über die Enumeration *AdviceType*, die als Werte die drei Varianten *Before*, *After* und *Around* bereitstellt.

4.1.3 Slices

Ähnlich wie die Advices werden auch Slices über eine abstrakte Oberklasse implementiert. Diese hat den Namen *AspectSlice*, da in vMAGIC bereits eine Klasse mit dem Namen *Slice* existiert. Auch hier sind Erweiterungen denkbar, etwa andere Arten von Slices für neue Join Points. Für die aktuelle Version von AspectVHDL werden vier Slice Typen verwendet: *EnumerationSlice*, *RecordSlice*, *ProcessSlice* und *ArchitectureSlice*. Sie alle haben gemeinsam, dass sie den Code verwalten, der zum Einweben in die in Abschnitt 2.1.2 beschriebenen Join Points benötigt wird. Im Falle von Prozessen ist das die

Liste von Signalen, die der Sensitivitätsliste hinzugefügt werden sollen; bei Architekturen die Liste der Deklarationen und Ausdrücke, die eingewoben werden sollen. Dadurch, dass hier die gleichen Typen verwendet werden, die auch vMAGIC selbst benutzt, wird das Parsen deutlich vereinfacht: Die Routinen von vMAGIC, mit denen z. B. Architekturen eingelesen werden, können einfach wiederverwendet werden, um Architektur-Slices zu verarbeiten.

Ein ähnliches Prinzip kommt bei *EnumerationSlice* und *RecordSlice* zum Einsatz: Sie verwalten jeweils ein ganzes Enumeration- bzw. Record-Objekt, in dem die einzuwebenden Werte bzw. Attribute hinterlegt sind. Auch dies verbessert die Wiederverwendbarkeit des vMAGIC Codes; beim Parsen kann schlicht die bestehende Funktion zum Einlesen von Enumerations und Records aufgerufen werden.

4.1.4 Pointcuts

Ein weiteres AspectVHDL-Element, das durch Metaklassen repräsentiert werden muss, sind die Pointcuts. Da deren Interpretation eine der Hauptaufgaben des Aspektwebers ist (siehe Abschnitt 4.3.2), ist es besonders wichtig, dass dies bereits durch die Implementierung der Metaklassen möglichst einfach gehalten wird. Idealerweise sollten sie so verwaltet werden, dass beim Parsen eine Struktur entsteht, aus der die Semantik der booleschen Ausdrücke bereits ersichtlich wird.

An dieser Stelle war es einmal mehr hilfreich, den vMAGIC-Code wiederzuverwenden. In vMAGIC werden alle VHDL-Ausdrücke unter einer Oberklasse zusammengefasst, dem Typ *Expression*. Dazu zählen neben z. B. arithmetischen Ausdrücken und Funktionsaufrufen auch boolesche Ausdrücke. Der Parser erstellt aus diesen Typen eine Baumstruktur, deren Aufbau die Auswertungsreihenfolge des Ausdrucks widerspiegelt. Eine solche Struktur ist auch für die Pointcuts ideal: Der Aspektweber kann durch einen einfachen rekursiven Durchlauf des Baumes ermitteln, ob ein Join Point innerhalb des Pointcuts liegt. Aus diesem Grund sind die Pointcuts als Spezialisierung allgemeiner VHDL Ausdrücke implementiert.

Durch dieses Vorgehen können die Klassen, die in vMAGIC booleschen Ausdrücke repräsentieren, mitverwendet werden. Somit müssen nur noch Klassen für die Pointcut-Funktionen erstellt werden (vgl. Abschnitt 2.2). Hierbei wurden die Funktionen *within*, *architecture*, *type* und *process* in der Klasse *PointcutPatternExpression* zusammengefasst, wobei der Typ durch ein Objekt der Enumeration *PointcutType* festgelegt wird. Dies hat den Vorteil, dass die Anzahl an gleichartigen Klassen reduziert wird: Alle vier Pointcut-Funktionen haben als Parameter ein oder zwei Strings, die durch die Metaklasse verwaltet werden müssen. Wird nur ein Parameter benötigt, kann das zweite Attribut einfach leer gelassen werden. Das Erstellen von vier unterschiedlichen Klassen mit nahezu identischer Funktionalität erschien an dieser Stelle nicht sinnvoll.

Die beiden verbleibenden Pointcut-Funktionen *args* und *call* sind jedoch so unterschiedlich, dass hier zwei zusätzliche Klassen von Nöten sind. *PointcutArgsExpression* verwaltet die Datentypen, nach denen gefiltert werden soll, als Liste von Strings; *PointcutCallExpression* enthält eine Referenz auf ein *PointcutCall*-Objekt, welches die Informationen enthält, nach denen die Prozeduraufrufe ausgewählt werden sollen.

4.2 Parser

Der nächste wichtige Teil der Implementierung ist die Erweiterung des Parsers. Dieser muss in der Lage sein, AspectVHDL-Code zu verarbeiten und daraus Objekte der oben beschriebenen Metaklassen zu erstellen. Wie in Abschnitt 3.1.3 erläutert, arbeitet der vMAGIC Parser in zwei Phasen: Zunächst wird ein Syntaxbaum erstellt, dessen Elemente direkt den Sprachelementen entsprechen, anschließend werden aus diesem die Metaobjekte erzeugt. Für diese beiden Schritte verwendet vMAGIC den Parsergenerator ANTLR.

ANTLR ist ein quelloffener Parsergenerator, der seit 1989 aktiv entwickelt wird [11][12]. Die Software erlaubt es, auf Basis einer gegebenen kontextfreien Grammatik einen Lexer und/oder Parser zu erstellen, der zur Grammatik passenden Text einliest und daraus einen abstrakten Syntaxbaum erzeugt. Zusätzlich dazu ist ANTLR zur Erzeugung von sogenannten *Tree Parsern* in der Lage – Programmen, die wiederum abstrakte Syntaxbäume einlesen und diese weiterverarbeiten. Mit Hilfe eines solchen Baum-Parsers wird in der zweiten Phase des vMAGIC-Parsers der Syntaxbaum in Metaobjekte umgewandelt.

Die Grammatiken, die ANTLR als Eingabe verwendet, müssen in einer festgelegten Syntax vorliegen, die auf der erweiterten Backus-Naur-Form basiert. Für die beiden Phasen des vMAGIC-Parsers werden die Grammatiken *VhdlAntlr.g* und *MetaClassCreator.g* verwendet. Erstere sorgt hierbei für die Generierung von Lexer und Parser zur Erstellung des abstrakten Syntaxbaums, letztere für die Erzeugung des Baum-Parsers. Beide müssen um zusätzliche Regeln erweitert werden, damit die erzeugten Parser die AspectVHDL-Konstrukte verarbeiten können. Dies geschah mit Hilfe von Version 1.4 der grafischen Entwicklungsumgebung ANTLRWorks.

4.2.1 Erweiterung des VHDL-Parsers

Die Regeln, die ANTLR zum Generieren des VHDL-Parsers benötigt, entsprechen im Wesentlichen der Sprachspezifikation von VHDL. Dementsprechend kann auch die AspectVHDL-Grammatik aus [3] größtenteils übernommen werden, um den Parser zu erweitern. Einige Anpassungen müssen aber dennoch vorgenommen werden.

Zum einen gibt es in ANTLR die Möglichkeit zu spezifizieren, wie aus einer Regel ein Knoten im Syntaxbaum erstellt werden soll. Dies ist von entscheidender Bedeutung, da

```
1  subprogram_advice
2      :    ADVICE type=advice_type COLON pc=pointcut_expr IS
3               statements=subprogram_statement_part
4           END (ADVICE)? SEMI
5           -> ^(ADVICE $type $pc $statements)
6      ;
```

Abbildung 4.2: Regel zum Parsen von Prozedur-Advices in der Grammatik des VHDL-Parsers

```
1  advice around( signal r : out decode_out_type) : call(...) is
2      r.ctrl_ex.fpu_op <= FPU_ADD;
3  end advice;
```

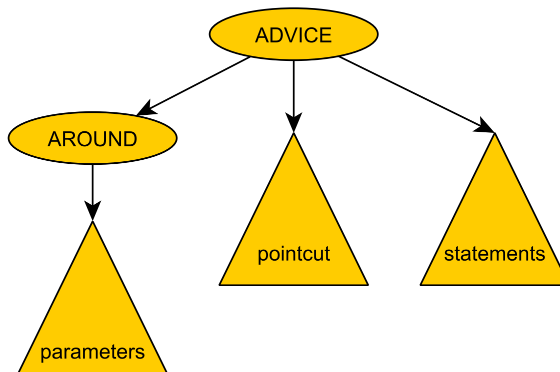


Abbildung 4.3: Codebeispiel für einen Prozedur-Advice und Ausschnitt aus dem daraus erstellten Syntaxbaum

sonst auch für das weitere Vorgehen redundante Sprachelemente wie Semikolons oder Klammern in den Syntaxbaum aufgenommen werden und dessen Weiterverarbeitung erschwert wird.

In Abbildung 4.2 ist beispielhaft die Regel zum Parsen von Prozedur-Advices dargestellt. Durch sie wird nicht nur die Syntax für diese Anweisungen festgelegt, sondern auch bestimmt, wie der zugehörige Syntaxbaum aufgebaut ist. Die vorletzte Zeile der Regel bewirkt, dass für einen eingelesenen Prozedur-Advice ein neuer Knoten mit der Beschriftung *ADVICE* angelegt wird. Diesem sind der gelesene Advicetyp, Pointcut und die zugehörigen Anweisungen untergeordnet. Wie genau diese Teilbäume aussehen, wird in den referenzierten Regeln *advice_type*, *pointcut_expr* und *subprogram_statement_part* spezifiziert. Diese Umwandlung wird in Abbildung 4.3 beispielhaft dargestellt.

Auf diese Weise können alle Elemente von AspectVHDL in eindeutige und redundanzfreie Teilbäume des Syntaxbaums umgewandelt werden. Durch diesen Aufbau wird die Weiterverarbeitung durch den Baum-Parser stark vereinfacht.

Neben diesem wichtigen Unterschied weichen die für den Parser erstellten Regeln auch an einigen anderen Stellen leicht von den Spezifikationen aus [3] ab. Dies ist unter anderem dadurch bedingt, dass für ANTLR verwendete Grammatiken nicht linksrekursiv sein dürfen und somit Regeln der Form *pointcut_expr : pointcut_expr AND pointcut_expr* nicht erlaubt sind. Die Regeln für die Pointcuts kommen daher umformuliert in der Grammatik vor und ähneln den Regeln, die dort auch für normale VHDL-Ausdrücke verwendet werden. Dies hat den Vorteil, dass der für beide Ausdruckstypen generierte Syntaxbaum die gleiche Form hat und so im Folgenden Pointcuts nicht mehr explizit von diesen unterschieden werden müssen.

4.2.2 Anpassung des Baum-Parsers

Im nächsten Schritt setzt vMAGIC die Funktion von ANTLR ein, Baum-Parser zu erstellen, um den Syntaxbaum einzulesen und aus ihm einfach zu verarbeitende Java-Objekte zu generieren. Diese Darstellung der Sprachkonstrukte als Metaobjekte ist auch für die AspectVHDL-Elemente wünschenswert, da sie als Objekte der in Abschnitt 4.1 vorgestellten Metaklassen deutlich einfacher zu verwalten sind als als Knoten im Syntaxbaum. Die dafür zuständige Grammatik *MetaClassCreator.g* wurde daher um entsprechende zusätzliche Regeln erweitert.

In Abbildung 4.4 ist zur Verdeutlichung der Funktionsweise des Baum-Parsers einmal mehr die Regel zum Verarbeiten von Prozedur-Advices dargestellt. Wie alle Regeln des Baum-Parsers liest diese einen Teilbaum einer bestimmten Form ein und gibt ein Java-Objekt an die übergeordnete Regel zurück. Hier ist der Rückgabewert vom Typ *SubprogramAdivce*, derjenigen Metaklasse, die zur Repräsentation von Prozedur-Advices vorgesehen ist (vgl. Abschnitt 4.1.2). Um ein Objekt dieser Klasse zu generieren, werden innerhalb der Regel in geschweiften Klammern Java-Anweisungen angegeben, die ausgeführt werden, wenn dieser Teil der Regel zum Tragen kommt.

In diesem Fall wird ein *ADVICE*-Knoten erwartet, wie er durch die Regel in Abbildung 4.2 erzeugt wird. Zusätzlich wird der Subknoten für den Advicetyp gleich mit eingelesen (Zeilen 5-12). Dies hängt damit zusammen, dass sowohl der Advicetyp als auch die zugehörige Parameterliste in dem zurückgegebenen *SubprogramAdvice*-Objekt gespeichert werden sollen. Hierzu wird zu Beginn (Zeilen 2-3) eine Variable vom Typ *AdviceType* und eine Liste für die Parameter angelegt. Beim Einlesen des Knotens für den Advicetyp werden diese dann mit den entsprechenden Werten befüllt.

Anschließend wird der Pointcut eingelesen. Da dieser die gleiche Form wie allgemeine VHDL-Ausdrücke hat (vgl. Abschnitt 4.2.1), kann das allgemeine Symbol *expression* verwendet werden. An geeigneter Stelle in den dazugehörigen Regeln sind die Pointcut-

```
1  subprogram_advice returns [SubprogramAdvice value]
2  :^(ADVICE
3      {AdviceType type = null;
4      LinkedList<VhdlObjectProvider> parameters=new LinkedList();}
5      ^(
6          (
7              BEFORE {type = AdviceType.BEFORE;}
8              | AFTER {type = AdviceType.AFTER;}
9              | AROUND {type = AdviceType.AROUND;}
10         )
11         ( id=interface_declaration { parameters.add($id.value); } )*
12     )
13     expression
14     {$value = new SubprogramAdvice(parameters, null,
15                                         type, $expression.value);}
16     (ss=sequential_statement {$value.getStatements().add($ss.value);})*
17 )
18 ;
```

Abbildung 4.4: Regel zum Parsen von Prozedur-Advices in der Grammatik des Baum-Parsers

Funktionen als Alternative angegeben. Aus den so eingelesenen Teilbäumen wird in den Zeilen 14-15 das *SubprogramAdvice*-Objekt konstruiert und in der Rückgabevariable *value* gespeichert. Danach noch folgende sequenzielle Anweisungen werden an die entsprechende Liste des Objekts angehängt.

Auf diese Art und Weise wurde für alle zuvor im VHDL-Parser neu erstellten Syntaxbaum-Knotentypen eine Regel erstellt, die daraus die passenden Metaobjekte generiert. Eine gewisse Herausforderung stellt an dieser Stelle die Auflösung von Bezeichnern dar – vMAGIC stellt hier ein System zur Verfügung, mit dessen Hilfe Referenzen auf in Beziehung stehende Objekte erzeugt werden können, wie etwa von einer Architektur auf deren Entität. Eingelesene Bezeichner werden hierfür abhängig von ihrem Gültigkeitsbereich in einer Datenstruktur gespeichert, sodass sie anschließend aufgelöst und dem zugehörigen Objekt zugeordnet werden können. Damit das reibungslos funktioniert, müssen Aspekte, Advices und Slices jeweils als in sich geschlossene Geltungsbereiche angegeben werden, genauso wie das bei Paketen oder Prozeduren der Fall ist.

Zusätzlich wird dieses System benutzt werden, um Introduction-Advices ihre Slices zuzuordnen. So kann schon während des Parsens bestimmt werden, welcher Advice sich auf welchen Slice bezieht. Diese Methode hat jedoch den Nachteil, dass der Advice immer nach dem Slice stehen muss, da nur bereits eingelesene Bezeichner aufgelöst werden können. Diese Einschränkung ist jedoch mit den Standards von VHDL konsistent, da

eine Architektur z. B. auch immer hinter der von ihr implementierten Entität stehen muss.

4.3 Weber

Die zentrale Aufgabe des Aspektwebers besteht darin, den Aspekt-Code einzuweben. Durch dessen komfortable Darstellung als Metaobjekte (vgl. Abschnitt 4.1) und mit den Vorüberlegungen aus Abschnitt 3.2 lässt sich diese in drei Teilprobleme aufteilen. Diese Probleme werden durch die drei Klassen *VhdlAnalyser*, *AspectWeaver* und *PointcutInterpreter* gelöst im Paket *aspectvhdl*. Deren Interaktion ist in Abbildung 4.5 als Sequenzdiagramm dargestellt.

Nach dem Einlesen der AspectVHDL-Dateien wird von der Hauptmethode eine Instanz der Klasse *AspectWeaver* erstellt. Diese leitet die übergebenen Dateien an einen neu erstellten *VhdlAnalyser* weiter, der diese nach Join Points durchsucht. Wie genau das funktioniert, wird in Abschnitt 4.3.1 erläutert. Anschließend können über den Aspektweber beliebig Aspekte in die Dateien eingewoben werden. In der Regel sind das diejenigen, die in den eingelesenen *.avhd*-Dateien gefunden wurden; im Sequenzdiagramm ist beispielhaft der Ablauf für einen einzelnen Aspekt dargestellt.

Der Aspektweber durchläuft die im Aspekt deklarierten Advices und ermittelt die zugehörigen Join Points durch einen Aufruf des *VhdlAnalysers*. Dieser iteriert über alle zuvor gefundenen Join Points und gibt diese an den *PointcutInterpreter* weiter, um zu überprüfen, welche von ihnen im gegebenen Pointcut liegen. Hier wird der Pointcut gemäß des in Abschnitt 4.3.2 beschriebenen Vorgehens evaluiert und das Ergebnis in Form eines booleschen Wertes zurückgegeben. Liegt der Join Point im Pointcut, wird er zur Ergebnisliste hinzugefügt. Diese Liste wird an den Aspektweber zurückgegeben und dort benutzt, um in den Advice in die enthaltenen Join Points einzuweben. Dieser Vorgang wird in Abschnitt 4.3.3 genauer vorgestellt.

4.3.1 Analyse des VHDL-Codes

Um entscheiden zu können, welche Join Points von einem Pointcut abgedeckt werden, müssen zuvor alle Join Points bekannt sein. Dadurch, dass die Pointcuts durch den Einsatz von booleschen Ausdrücken recht komplex werden können, ist immer nur die Auswertung eines einzigen Join Points durch den PointcutInterpreter möglich. Vorhersagen über die Position der Join Points können nur auf Basis des Pointcuts im Allgemeinen nicht getroffen werden. Der erste Schritt des Webprozesses besteht daher darin, alle Join Points in den gegebenen VHDL-Dateien zu finden.

Dies wird durch die Klasse *VhdlAnalyser* im Paket *aspectvhdl* erledigt. Sie verwaltet für alle Join Point-Typen eine verkettete Liste und befüllt diese beim Konstruktorauf-

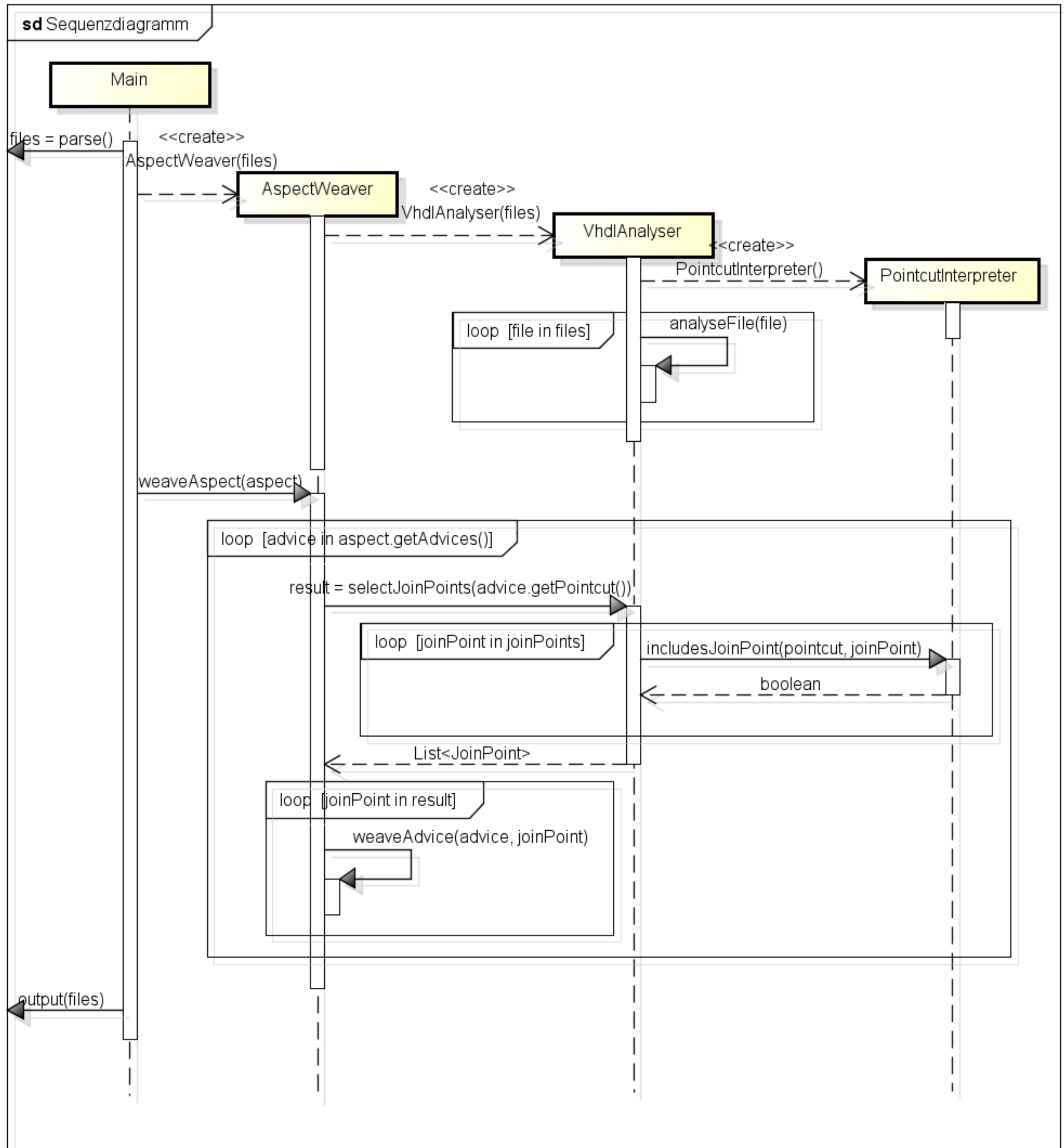


Abbildung 4.5: Sequenzdiagramm zum Ablauf des Aspektwebers

ruf mit den Join Points der übergebenen VHDL-Dateien. Die Art und Weise, wie diese gefunden werden, wird in den folgenden Abschnitten erläutert.

4.3.1.1 Architekturen, Prozesse und Typen

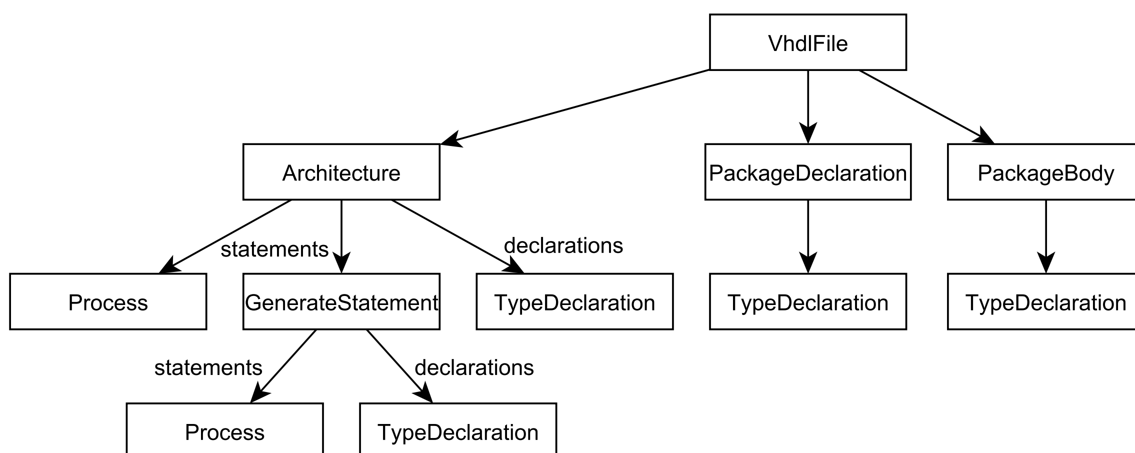


Abbildung 4.6: Visualisierung der Positionen im Metaobjektbaum, an denen Join Points liegen können

Die Architekturen sind der am einfachsten zu findende Join Point-Typ. Da diese nur auf oberster Ebene des Quelltextes stehen können, reicht ein einfacher Durchlauf durch die Elemente der VHDL-Dateien, um alle Architekturen zu erreichen. Auf die gleiche Art und Weise können auch die Pakete gefunden werden. Ausgehend davon werden jeweils die dort vorgenommenen Deklarationen und Anweisungen nach den restlichen Join Points durchsucht. Abbildung 4.6 visualisiert, wo diese sich befinden können: Sowohl in Architekturen als auch in beiden Paketteilen können Typdeklarationen stattfinden. Diese Deklarationslisten werden daher durchsucht und gefundenen Typdeklarationen der Liste von Join Points hinzugefügt.

Etwas komplizierter ist die Analyse der parallelen Anweisungen von Architekturen, da diese nicht nur Prozesse, sondern auch sogenannte *Generate-Anweisungen* enthalten können. Diese werden verwendet, um Anweisungen und Deklarationen zu gruppieren und abhängig von Bedingungen einmal oder mehrfach zu generieren. Auf diese Weise können z. B. array-artige Hardwarestrukturen wie FIFO-Speicher mit geringem Codeaufwand realisiert werden [1]. Da die so spezifizierten Anweisungen selber wieder Generate-Anweisungen enthalten können, werden diese rekursiv von der Methode *analyseConcurrent* durchsucht. Auf diese Weise können parallele Anweisungen beliebiger Schachtelungstiefe analysiert und alle Prozesse und Typdeklarationen gefunden werden. Auf gleiche Art und Weise wird mit etwaigen *Block*-Befehlen umgegangen.

Eine weitere Aufgabe, die der *VhdlAnalyser* an dieser Stelle übernimmt, ist die Erzeugung von Referenzen von den Join Points auf die Architekturen bzw. Pakete, in denen sie sich befinden. Diese werden später zur Auswertung der *Within*-Pointcuts benötigt. Zwar bietet vMAGIC für alle VHDL-Elemente einen *Parent*-Zeiger auf das jeweils übergeordnete Element, über den früher oder später die Architektur bzw. das Paket erreicht werden kann. Diese aber bei der Auswertung eines jeden Pointcuts zu durchlaufen, wäre sehr ineffizient. Stattdessen wurden Prozesse, Typen und Prozeduraufrufe um das Attribut *libraryParent* erweitert, in dem eine direkte Referenz auf das oberste übergeordnete VHDL-Element gespeichert werden kann. Während des Erstellens der Join Point-Listen wird diese Referenz gesetzt, sodass die Auswertung von *Within*-Ausdrücken später in konstanter Laufzeit möglich ist. Dazu werden die Elternzeiger so lange verfolgt, bis eine Architektur oder ein Paket erreicht wird. Damit das in allen Fällen möglich ist, wurde das Interface *Parentable* erstellt, das von allen VHDL-Elementen mit Elternzeiger implementiert wird.

4.3.1.2 Prozeduraufrufe

Die Suche nach Prozeduraufrufen gestaltet sich etwas schwieriger als die nach den restlichen Join Points. Zum einen unterscheidet vMAGIC hier zwei verschiedene Typen: *ProcedureCall* für normale Prozeduraufrufe als sequentielle Anweisungen und *ConcurrentProcedureCall* für Aufrufe aus parallelem Kontext heraus. Daher werden auch zwei Listen für die Join Points benutzt: jeweils eine für die beiden Typen. Um sie später gemeinsam behandeln zu können, wurde das Interface *AbstractProcedureCall* erstellt, das die gemeinsame Schnittstelle formuliert. Da dieses von beiden Typen implementiert wird, können Operationen auf beiden Aufruftypen künftig über dieses Interface durchgeführt werden.

Zum anderen können sequentielle Prozeduraufrufe stark verschachtelt sein. Sequentielle Anweisungen können in Schleifen, bedingten Anweisungen usw. enthalten sein. Ein rekursiver Durchlauf wie bei den *Generate*-Anweisungen wäre also mit beträchtlichem Codeaufwand verbunden. Um das zu vermeiden, wird die Liste der Prozeduraufrufe bereits beim Parsen erstellt: Immer wenn durch den Baum-Parser ein Prozeduraufruf eingelesen wird, wird das neue Objekt zu einer in der aktuellen VHDL-Datei gespeicherten Liste hinzugefügt. Der *VhdlAnalyser* muss die Listen der einzelnen Dateien dann nur noch zusammenfügen.

4.3.2 Interpretation der Pointcuts

Das zweite große Teilproblem des Aspektwebers ist die Interpretation der Pointcuts. Wie bereits erwähnt, wird sich hierbei darauf beschränkt zu überprüfen, ob *ein* Join Point von *einem* Pointcut abgedeckt wird. Dies geschieht durch die zustandslose Klasse *PointcutInterpreter*. Diese bietet für alle vier Join Point-Typen (Architektur, Typ, Prozess, Prozeduraufruf) eine Methode, welche diese Prüfung vornimmt.

Ein wichtiger Punkt, den es hier zu beachten gilt, besteht darin, dass diese Typen immer nur bestimmte Pointcut-Funktionen erlauben: *Process*, *Architecture* und *Type* sind jeweils nur für Prozesse, Architekturen und Typen erlaubt, *Args* und *Call* nur für Prozeduraufrufe und *Within* für alle Join Points *außer* Architekturen. Stößt der PointcutInterpreter auf eine unerlaubte Pointcut-Funktion, wirft er daher eine *InvalidPointcutException*.

4.3.2.1 Boolesche Ausdrücke

Durch die Verwendung von booleschen Ausdrücken können Pointcut-Ausdrücke beliebig kombiniert und verschachtelt werden. Der Parser erzeugt daraus eine Baumstruktur, die die Auswertungsreihenfolge des Ausdruckes widerspiegelt. Bei den Blättern des Baumes handelt es sich um die Pointcut-Funktionen, die restlichen Knoten stellen boolesche Operatoren dar. Um den Baum zu evaluieren, wird er, beginnend an der Wurzel, durch den PointcutInterpreter rekursiv durchlaufen. Wann immer ein Knoten zu einem booleschen Operator erreicht wird, werden die untergeordneten Teilbäume ausgewertet und die Ergebnisse über den Operator kombiniert. Dadurch muss im Folgenden nur noch die Auswertung von einzelnen Pointcut-Funktionen betrachtet werden.

4.3.2.2 Type, Process, Architecture

Relativ einfach ist hierbei die Evaluation der Funktionen, die Join Points nach ihrem Namen selektieren. Hier wird lediglich der im Pointcut angegebene String mit dem Bezeichner des Join Points verglichen. Sind diese identisch oder kommt im Pointcut eine Wildcard zum Einsatz, wird der Ausdruck zu *true* evaluiert.

4.3.2.3 Within

Etwas anspruchsvoller ist die Auswertung von *Within*-Ausdrücken. Um diese evaluieren zu können, muss der Name der Architektur bzw. des Pakets ermittelt werden können, in dem sich ein Join Point befindet. Hier kommt der in Abschnitt 4.3.1 beschriebene Zeiger *libraryParent* zum Einsatz: Über diesen kann die gesuchte Bibliothekseinheit direkt erreicht und so deren Name ermittelt werden.

4.3.2.4 Call und Args

Ebenfalls nicht ganz trivial ist die Evaluation von *Args*- und *Call*-Ausdrücken. Für beide müssen, zusätzlich zur Auswertung des Prozedurnamens, die Parametertypen überprüft werden. Hierzu müssen die im Pointcut angegebenen Datentypen mit den Typen der Parameter des Prozeduraufrufs verglichen werden. An dieser Stelle ist die Fähigkeit von vMAGIC entscheidend, Referenzen zwischen in Beziehung stehenden VHDL-Objekten herzustellen: Beim Parsen werden die Bezeichner in der Parameterliste aufgelöst und Referenzen auf die zugehörigen Signalobjekte erstellt. Diese Referenzen werden vom Aspektweber genutzt, um Datentypen der übergebenen Objekte mit den im Pointcut geforderten Datentypen zu vergleichen.

Weiterhin gibt es die Möglichkeit, im Pointcut anstatt der eigentlichen Datentypen den Bezeichner von einem der Advice-Parameter anzugeben. In diesem Fall wird dessen Datentyp für die Selektion der Join Points verwendet. Auf diese Weise kann redundanter Code eingespart werden; da in der Regel nach den im Advice vorliegenden Parametertypen gefiltert wird, ist es hilfreich, diese nicht doppelt angeben zu müssen. Um diese Funktionalität umzusetzen, vergleicht der Aspektweber die im Pointcut angegebenen Bezeichner mit den Namen der Parameter des Advices und nutzt bei einem Treffer fortan den Datentyp des entsprechenden Parameters zur Evaluation des Pointcuts.

Für diese Vorgänge sind einige recht technische Konversionen zwischen Objekten notwendig. Diese wurden in die Klasse *aspectvhdl.util.OutputUtil* ausgelagert. Zum einen werden an dieser Stelle die Durchläufe durch Parameterdeklarationen (wie etwa den Parameterlisten von Advices) vorgenommen, da diese durch die Nutzung des Aggregationsobjekts *VhdlObjectProvider* recht verschachtelt sein können. Zum anderen ist es nicht ganz einfach, den Namen eines Datentyps aus dem dazugehörigen Metaobjekt zu generieren. Dafür wird in *OutputUtil* die Methode *typeToString* zur Verfügung gestellt, die über einen Aufruf des Codegenerators das Typobjekt in seine textuelle Repräsentation umwandelt.

4.3.3 Einweben des Aspectcodes

Mit Hilfe der beiden Klassen *VhdlAnalyser* und *PointcutInterpreter* lassen sich die Join Points bestimmen, die durch einen bestimmten Advice modifiziert werden. Hierzu stellt der *VhdlAnalyser* Methoden für die einzelnen Join Point-Typen zur Verfügung, die einen Pointcut-Ausdruck erhalten und die Liste der Join Points zurückgeben, die von diesem abgedeckt werden. Das letzte verbleibende Problem des Aspektwebers ist somit das eigentliche Einweben des Aspekt-Codes an den gegebenen Stellen. Dies wird durch die im Folgenden erläuterte Klasse *AspectWeaver* vorgenommen.

4.3.3.1 Introductions

Im Falle von Introductions ist dieses Vorhaben recht einfach, da es lediglich auf die Konkatenation von Listen hinausläuft: Im Falle von Architekturen werden zusätzliche Anweisungen und Deklarationen an die entsprechenden Listen der Architektur angehängt, bei Signalen wird die Sensitivitätsliste um weitere Signale erweitert und auch bei Enumerations und Records werden die hinzukommenden Werte bzw. Attribute schlicht an Listen angefügt.

An dieser Stelle kommen wieder einmal die Vorteile der Metaklassen zum Tragen: Durch das objektorientierte Modell des VHDL-Codes sind derartige Erweiterungen außerordentlich simpel.

4.3.3.2 Prozeduren

Die Verarbeitung von Prozeduren erfordert dagegen deutlich mehr Überlegungen. Die Tatsache, dass nicht ganze Prozeduren, sondern Prozeduraufrufe modifiziert werden, hat einige Konsequenzen: Zum einen muss die Originalprozedur erhalten bleiben, damit sie von anderen, nicht betroffenen Aufrufen noch erreicht werden kann. Zum anderen kann zu einem gegebenen Prozeduraufruf nicht ohne Weiteres die zugehörige Prozedur ermittelt werden. Diese kann sich nicht nur in anderen Sprachelementen, sondern sogar in anderen Dateien befinden.

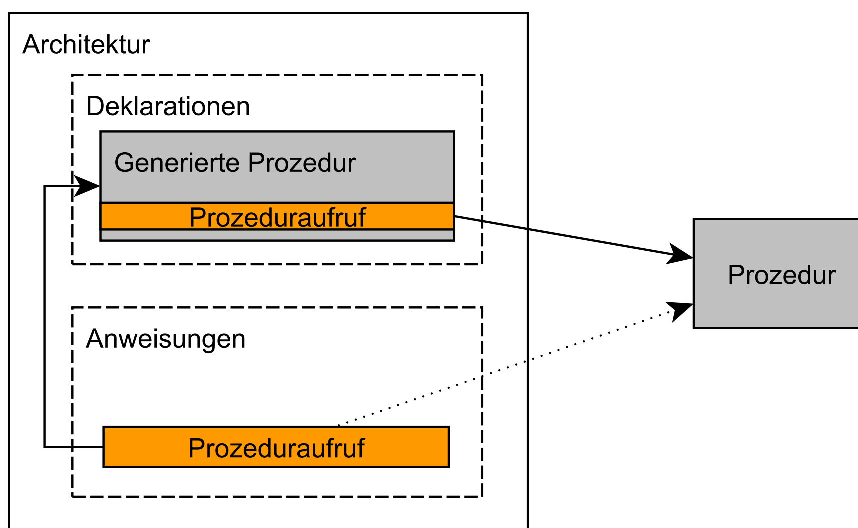


Abbildung 4.7: Schematische Darstellung der Vorgehensweise zum Einweben von Prozeduren

Daher wird für das Einweben von Prozeduren die in Abbildung 4.7 dargestellte Strategie verwendet. Ausgehend von dem Prozeduraufruf wird über den *parent*-Zeiger das nächste überordnete VHDL-Element gesucht, das Prozeduredeklarationen aufnehmen kann[1]. In der Abbildung handelt es sich hierbei um die Architektur. An dieser Stelle wird eine neue Prozedur erzeugt, die den im Advice angegebenen Code beinhaltet. Abhängig vom Advicetyp werden davor, dahinter oder anstelle der *proceed*-Anweisungen Aufrufe der Ursprungsprozedur eingefügt. Anschließend wird der als Join Point dienende Prozeduraufruf auf die neu generierte Prozedur umgeleitet.

Diese Positionierung der neuen Prozedur hat den Vorteil, dass die alte Prozedur an dieser Stelle garantiert sichtbar ist. Dadurch, dass es *zwischen* generierter Prozedur und Prozeduraufruf keinen weiteren Sichtbarkeitsbereich gibt, in dem sich die alte Prozedur

[1] Dazu zählen Architekturen, Pakete, Prozesse, *Generate*- und *Block*-Anweisungen sowie Unterprogramme selbst.

befinden könnte, muss sie für die neu erzeugte Prozedur genauso sichtbar sein wie für den Aufruf. Weiterhin kann so garantiert werden, dass alle am Prozeduraufruf sichtbaren Deklarationen auch im Advice-Code sichtbar sind. Dies ist wichtig, damit der Entwickler abschätzen kann, welche Objekte er hier benutzen kann.

Um Namenskonflikte zu verhindern, setzt sich der Name der neuen Prozedur aus dem Präfix "gen_", dem Namen der ursprünglichen Prozedur, dem Hashcode des Prozeduraufrufs und dem Hashcode des Advices zusammen. Da jeder Advice an jedem Join Point nur jeweils einmal einweben kann, ist die Einzigartigkeit dieses Namens garantiert.

Aktuell ist für AspectVHDL nicht genau definiert, wie die Zuordnung von den Argumenten des Prozeduraufrufs zu den Parametern des Advice abläuft. Zur Zeit erfolgt diese daher einfach anhand der Deklarationsreihenfolge. Dies hat zur Folge, dass ein Advice immer die gleichen Parametertypen haben muss wie die Prozeduraufrufe in die er einwebt. Diese Einschränkung wirkt sich mit dem aktuell bekannten Anwendungsbeispiel (vgl. Abschnitt 5.2) nicht negativ auf die Aussagekraft der Sprache aus; eine mögliche Verbesserung wird dennoch in Abschnitt 5.3.1 vorgestellt.

Eine letzte Herausforderung stellt die Möglichkeit dar, dass mehrere Advices einen einzigen Prozeduraufruf verändern. In diesem Fall werden die Advices in unbestimmter Reihenfolge abgearbeitet, wobei jeder Advice die von seinem Vorgänger erzeugte Prozedur erweitert. Der Ablauf ist also genauso wie bei einzelnen Advices auch: Neue Prozedur erzeugen, alte Prozedur einbinden, Prozeduraufruf umleiten. Es muss nur sichergestellt werden, dass der Prozeduraufruf, dessen Prozedurname nach der Umleitung durch den ersten Advice ja nicht mehr dem Original entspricht, noch korrekt als Join Point erkannt wird. Prozeduraufrufe enthalten daher das zusätzliche Attribut *originalProcedure*, das den Namen der ursprünglich aufgerufenen Prozedur speichert. Hierüber kann der PointcutInterpreter auch nach einer Modifikation noch erkennen, welcher Advice an einem Prozeduraufruf ansetzt.

Ein augenscheinlicher Nachteil dieses Vorgehens besteht darin, dass eine ganze Menge zusätzlicher Quelltext erzeugt wird. Für jeden von einem Advice manipulierten Prozeduraufruf wird eine neue Prozedur erzeugt. Man muss sich jedoch vergegenwärtigen, dass bei der Hardwaresynthese ohnehin für jeden Prozeduraufruf eine eigene Kopie des zugehörigen Hardwareelements angelegt wird. Für das Endprodukt spielt es also keine Rolle, in wie viele Einzelprozeduren das Projekt zergliedert ist.

4.3.3.3 Deklarationen in Aspekten

Wie in Abschnitt 2.3 erläutert, können Aspekte nicht nur Advices und Slices, sondern auch normale VHDL-Deklarationen enthalten. Diese müssen durch den Aspektweber so im regulären VHDL-Projekt untergebracht werden, dass sie im eingewobenen Code zur Verfügung stehen. Das hierzu verwendete Vorgehen wird in Abbildung 4.8 dargestellt.

Zunächst werden die Deklarationen in ein nach dem Aspekt benanntes Paket in einer neuen VHDL-Datei ausgelagert. Hierbei werden sie den VHDL-Spezifikationen entsprechend auf Paketkopf und -körper aufgeteilt. Damit alle nötigen Pakete zur Verfügung stehen, werden in die neue Datei ebenfalls die *use*-Klauseln der Aspektdatei übernommen.

Umgekehrt muss das neu angelegte Paket überall dort zur Verfügung stehen, wo der Code aus dem Aspekt eingewoben wurde. Daher wird die Menge der durch den Aspekt modifizierten VHDL-Dateien beim Einweben in einem *HashSet* gesammelt. Anschließend werden diese Dateien dann um eine *use*-Klausel erweitert, die das Paket einbindet. Da in dem Aspekt-Code ebenfalls die in der Aspektdatei inkludierten Pakete benutzt werden können, werden hier zusätzlich die dort verwendeten *use*-Klauseln eingewoben.

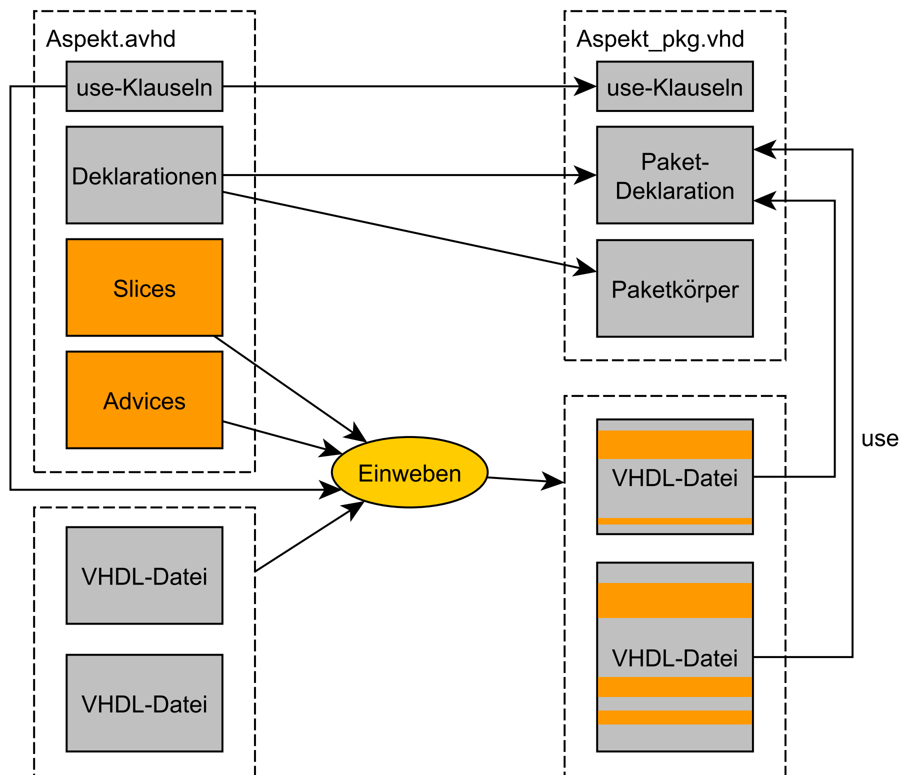


Abbildung 4.8: Übersicht über die Verarbeitung des Aspectcodes

5 Evaluation

Das folgende Kapitel beschäftigt sich damit, die in Kapitel 4 vorgestellte Implementierung des Aspektwebers zu evaluieren. Zunächst einmal lässt sich feststellen, dass die in Kapitel 2 formulierten Anforderungen an den Funktionsumfang des Programms vollständig erfüllt sind: Alle Sprachelemente von AspectVHDL können vom Aspektweber verarbeitet und ihrer Semantik nach eingewoben werden.

Dessen Evaluation ist damit aber natürlich nicht beendet. In Abschnitt 5.1 wird untersucht, inwieweit die Implementierung performant genug ist, um in der Praxis einsetzbar zu sein. Anschließend werden in Abschnitt 5.2 die Erfahrungen vorgestellt, die durch die Arbeit mit einem realistischen AspectVHDL-Projekt gewonnen werden konnten. Zuletzt folgen in Abschnitt 5.3 einige mögliche Erweiterungen für AspectVHDL und den Aspektweber.

5.1 Laufzeit

Eine wichtige Eigenschaft eines jeden Programms ist dessen Laufzeit. Dies gilt insbesondere für den Aspektweber – nur wenn dieser in akzeptabler Zeit terminiert, kann AspectVHDL ein hilfreiches Mittel zum Hardwareentwurf sein. In der Theorie ist die Laufzeit des Aspektwebers, gemäß dem Algorithmus aus Abschnitt 3.2, linear abhängig von der Anzahl der Join Points und der Advices. Es bleibt jedoch zu überprüfen, ob diese Annahme auch in der Praxis zutrifft und ob die absoluten Laufzeiten von Parser, Weber und Codegenerator akzeptabel sind. Dies soll der im Folgenden beschriebene Laufzeitversuch zeigen.

5.1.1 Szenario

Die für den Laufzeitversuch verwendete VHDL-Datei besteht aus einer leeren Entität und einer dazugehörigen Architektur. In dieser ist eine ebenfalls leere Prozedur deklariert, welche im parallelen Anweisungsteil der Architektur aufgerufen wird. Die Anzahl der Aufrufe wird hierbei im Verlauf der Versuchsreihe variiert, um die Menge der Join Points zu modifizieren.

Zusätzlich dazu wird eine Aspektdatei verwendet, die einen Aspekt mit einem oder mehreren Prozedur-Advices enthält. Diese greifen die zuvor erwähnten Join Points an und leiten die dortigen Prozeduraufrufe auf eine ebenfalls leere, eingewobene Prozedur

um. Die Zahl der Advices wurde im Folgenden ebenfalls variiert. Gemessen wurde die Laufzeit, die für die drei wesentlichen Schritte Parsen, Weben und Ausgabe nötig war.

5.1.2 Ergebnisse

In der ersten Testreihe wurde ein Advice und eine variable Anzahl Join Points verwendet. Die Ergebnisse sind in Abbildung 5.1 dargestellt: Die Laufzeiten aller drei Komponenten steigen erwartungsgemäß linear mit der Anzahl der Join Points an. Der Weber ist hierbei deutlich schneller als der Parser und somit für die Gesamtlaufzeit nicht signifikant. Diese bleibt auch bei 6000 Join Points bei unter einer Sekunde und somit absolut im Rahmen.

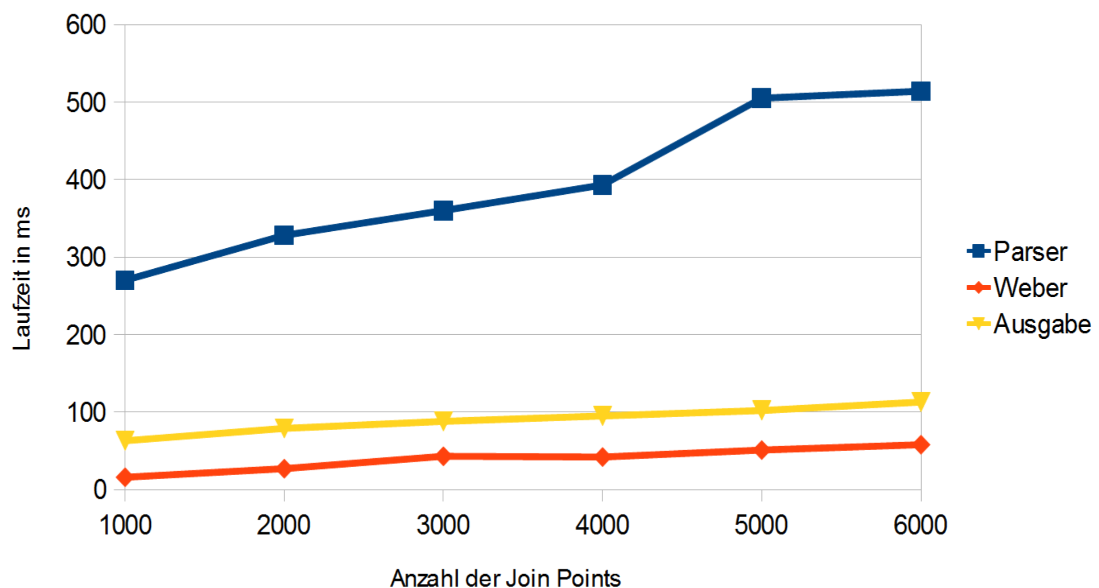


Abbildung 5.1: Darstellung der Laufzeiten von Parser, Weber und Ausgabe bei unterschiedlichen Anzahlen von Join Points

In der zweiten Testreihe wurde eine gleichbleibende Menge von 1000 Join Points verwendet und die Anzahl der Advices erhöht. Dadurch, dass hier nur wenig zusätzlicher Quellcode hinzugefügt wird, bleibt die Laufzeit des Parsers nahezu konstant. Für Weber und Codegenerator ist auch hier ein linearer Anstieg zu verzeichnen, beide Kurven bleiben aber auch bei 25 Einwebevorgängen pro Join Point unter der des Parsers. Die Gesamtlaufzeit fällt durch die geringere Belastung des Parsers dabei sogar noch geringer aus, als beim ersten Testlauf – und das, obwohl zuletzt 25000 neue Prozeduren erstellt werden und die Ausgabedatei über 3 MB groß ist. Es lässt sich also festhalten, dass der Aspektweber für den Praxisgebrauch effizient arbeitet. Höhere Laufzeiten können allenfalls durch den Parser verursacht werden; der Weber verhält sich auch bei unrealistisch großen Mengen von Advices und Join Points performant.

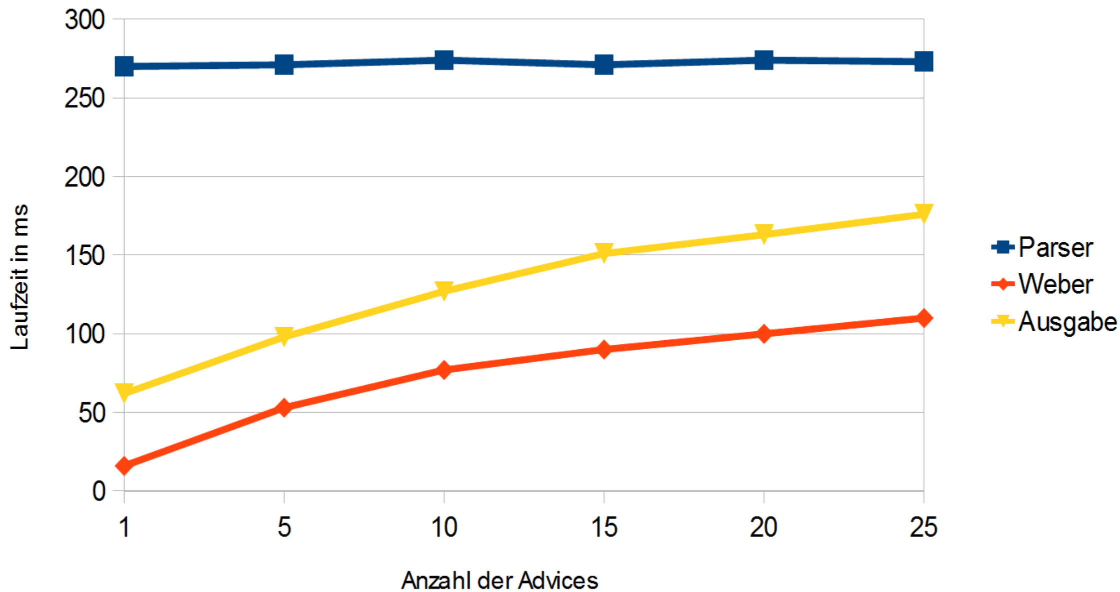


Abbildung 5.2: Darstellung der Laufzeiten von Parser, Weber und Ausgabe bei unterschiedlichen Anzahlen von Advices

Dies wird auch durch die im folgenden Abschnitt beschriebene Arbeit mit dem MB-Lite-Projekt deutlich: Zur Verarbeitung dieses mit 48 VHDL-Dateien und drei Aspekten realistischen und recht umfangreichen Projekts benötigte der Aspektweber ebenfalls nur wenig mehr als eine Sekunde. Mit 934 Millisekunden wird der größte Teil davon durch den Parser verursacht, während Weber und Codegenerator nur jeweils 15 und 129 Millisekunden benötigen. Die Laufzeit des Parsers steigt dabei zwar in merkbare Regionen an, sie steht aber immer noch in keinem Verhältnis zu der Zeit, die für die VHDL-Synthese benötigt wird: Das Synthese-Tool *ISE* von Xilinx benötigt unter gleichen Bedingungen mehr als anderthalb Minuten zur Synthese des MB-Lite Prozessors.

5.2 MB-Lite Implementierung

Wie bereits in Abschnitt 1.2 erläutert, wurde bereits in der vorherigen Arbeit zu AspectVHDL [3] der MB-Lite Prozessor betrachtet. Drei von dessen optionalen Erweiterungen wurden als querschneidende Belange identifiziert, deren Kapselung in Aspekten vorteilhaft wäre. Um die Zweckmäßigkeit von AspectVHDL zu demonstrieren, wurde das VHDL-Projekt mit dem Ziel, diese Erweiterungen in Aspekten zusammenzufassen, einem Refactoring unterzogen.

Die erste vorgenommene Änderung ist die Auslagerung von Anweisungsabschnitten in Prozeduren. Der ursprüngliche Quellcode des MB-Lite Prozessors enthält einige sehr

lange Prozesse, deren Ablauf auf Grund von fehlenden Join Points nicht durch Aspekte modifiziert werden kann. Daher ist es nötig, die für die Aspekte relevanten Abschnitte in Prozeduren auszulagern. In der Zukunft wäre es daher vorteilhaft, AspectVHDL-Projekte von Anfang an mit Rücksicht auf die Aspekte zu entwerfen, um so die Möglichkeit zu haben, in Schlüsselstellen des Projekts Code einzuweben. Nach dieser Anpassung konnten die Codeabschnitte für die Erweiterungen extrahiert und in Aspekten zusammengefasst werden. Auf Grund des fehlenden Aspektwebers war das Einweben dieser Aspekte jedoch nur gedanklich möglich.

Dies hat sich nun geändert: Mit Hilfe des neu erstellten Aspektwebers können die damaligen Ergebnisse überprüft und weiterentwickelt werden. Gleichzeitig bietet das MB-Lite Projekt ein erstes, realistisches Anwendungsbeispiel, in dem der Aspektweber eingesetzt werden kann. Hierzu wurde der Aspekt-Code und das modifizierte VHDL-Projekt als Eingabe für den Aspektweber verwendet. Bei der dazu nötigen Überarbeitung des Aspect-Codes wurde eine bisher nicht betrachtete Problematik deutlich, die das Extrahieren der Erweiterungen erschwert:

In der *Execute*-Phase des MB-Lite Prozessors werden lokale Signale verwendet, die den Status der Gleitkomma-Einheit speichern und durch die zugehörigen Prozeduren modifiziert werden. Diese gehören natürlich zur FPU-Erweiterung und sollten daher in den entsprechenden Aspekt ausgelagert werden. Hierbei ergibt sich folgendes Problem: Zwar können die Signale selbst durch den *Architecture*-Join Point eingewoben werden, sie können aber nicht in die Parameterliste der Prozeduren aufgenommen werden. Die FPU-Signale können also nicht ohne Weiteres aus dem vom restlichen Projekt separiert werden, da sie sonst in den Prozeduren nicht mehr zugreifbar sind.

An dieser Stelle kommt die in Abschnitt 4.7 erläuterte Garantie zum Tragen, die besagt, dass im Advice-Code alle Deklarationen sichtbar sind, die auch durch den Prozeduraufruf erreicht werden können. Auf diese Weise können die Signale in den neuen Prozedurabschnitten verwendet werden, *auch ohne* dass diese als Argument übergeben werden. Durch das Einweben der Signale in die Architektur ist sichergestellt, dass alle Advices, die in dieser Architektur Prozedur-Code einweben, auf diese Signale zugreifen können.

Mit Hilfe dieser Überlegungen können die drei betrachteten Erweiterung des MB-Lite Prozessors vollständig in Aspekten gekapselt werden. Der Aspektweber ist in der Lage, diese wieder in den verbleibenden Code einzuweben und synthesefähigen VHDL-Code zu generieren.

5.3 Mögliche Erweiterungen

In dem folgenden Abschnitt werden einige mögliche Erweiterungen von AspectVHDL und den Aspektweber vorgestellt, die die Mächtigkeit der Sprache und deren Implementierung durch den Weber verbessern könnten. Diese basieren auf den Erfahrungen, die

während der Implementierung gesammelt wurden und beziehen sich mehr auf die Umsetzung durch den Aspektweber, als das für die in [3] formulierten Erweiterungen der Sprache der Fall ist.

5.3.1 Bindung von Prozedur-Argumenten durch Pointcuts

Wie in Abschnitt 4.3.3.2 erläutert, müssen Prozedur-Advices aktuell die gleichen Parametertypen haben wie die Prozeduraufrufe, die sie betreffen. Die direkte Zuordnung von Advice- zu Prozedurparametern ist für die aktuell bekannten Anwendungen zweckmäßig, sorgt jedoch für einige Einschränkungen. So ist es zum Beispiel nicht möglich, in einem Advice nur einen Teil der Parameter einer Prozedur zu verwenden und so in Prozeduraufrufe mit unterschiedlichen Parametertypen einzuweben.

Andere aspektorientierte Sprachen wie AspectC++ erlauben es, die Parameter über den *Args*-Pointcut aneinander zu binden. Auf diese Weise können Advices dynamischer an ihre Join Points angepasst werden, und die oben beschriebenen Einschränkungen fallen weg. Die Möglichkeit, ein solches System auch für AspectVHDL zu übernehmen, wurde bereits in [3] formuliert. Es bleiben jedoch einige Fragen offen, die eine Implementierung erschweren: Was passiert, wenn mehrere, möglicherweise widersprüchliche *Args*-Funktionen im Pointcut vorkommen? Wie wird mit *Args*-Aufrufen umgegangen, die tief im Pointcut verschachtelt und unter Umständen negiert sind?

Ungeachtet dieser Probleme kann es dennoch sinnvoll sein, einen solchen Ansatz zu verfolgen. Sollte sich in zukünftigen Anwendungen von AspectVHDL zeigen, dass eine solche Funktionalität wünschenswert ist, liefern die aus AspectC++ und AspectJ bekannten Konzepte eine gute Vorlage für eine Umsetzung in AspectVHDL.

5.3.2 Verbindung von Prozeduraufruf und Prozedur

Wie in Abschnitt 4.3.3.2 erläutert, hat der Aspektweber derzeit nicht die Möglichkeit, über einen Prozeduraufruf auf die von ihm aufgerufene Prozedur zuzugreifen. Dies ist durch die vielen verschiedenen Positionen bedingt, an denen sich die Deklaration der Prozedur befinden kann: Um diese durchsuchen zu können, muss genau nachvollzogen werden, welche Deklarationen für den Prozeduraufruf sichtbar sind und welche nicht. Insbesondere ist es nötig, die *use*-Klauseln zu interpretieren und auch in eingebundenen Dateien nach der Prozedurdeklaration zu suchen.

Trotz des Aufwands wäre eine solche Verbindung von Aufruf und Prozedur wünschenswert. Über eine derartige Referenz könnte zum Beispiel sicher ermittelt werden, welche Aufrufe sich auf dieselbe Prozedur beziehen. Dies ist auch für etwaige zukünftige Pointcuts relevant, wie etwa der Selektion aller Aufrufe einer bestimmten Prozedur. Außerdem würde sich so die Ermittlung der Argumenttypen eines Prozeduraufrufs vereinfachen. Auch wenn sie für die jetzige Funktionalität nicht benötigt wird, wäre eine solche Referenz also für die Verarbeitung des Quellcodes hilfreich.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde zu der bestehenden Sprache AspectVHDL erfolgreich ein voll funktionsfähiger Aspektweber erstellt. Dieser ist in der Lage, alle Sprachelemente von AspectVHDL einzulesen und gemäß ihrer Semantik zu verarbeiten. Er erlaubt es somit, AspectVHDL in der Praxis zur aspektorientierten Hardwareentwicklung einzusetzen.

Hierzu wurden zunächst diverse bestehende VHDL-Parser analysiert und bewertet, die als Basis für die Implementierung in Frage kamen. Ausgewählt wurde das Java-Framework vMAGIC, das zusätzlich einen Codegenerator umfasst und eingelesenen VHDL-Code durch ein mächtiges Metamodell repräsentiert. Dieses wurde erweitert, um die neuen Sprachelemente von AspectVHDL darstellen zu können. Entsprechend dazu wurden die Grammatiken des über den Parsergenerator ANTLR erstellten Parsers um die Regeln ergänzt, die zur Verarbeitung von AspectVHDL-Code nötig sind.

Auf Basis dessen wurde der eigentliche Aspektweber erstellt, welcher den eingelesenen VHDL-Code anhand der Spezifikationen von AspectVHDL manipuliert. Hierzu wurden insbesondere Lösungen zum Finden der Join Points im VHDL-Code, zur Auswertung der Pointcut-Ausdrücke und zum Einweben des Aspect-Codes entwickelt. Ein besonderes Augenmerk auf die Erweiterbarkeit der Lösung gerichtet, um eine künftige Weiterentwicklung von AspectVHDL vorzubereiten.

Die erarbeitete Lösung wurde anschließend mit Hinblick auf ihre Praxistauglichkeit evaluiert. Es konnte festgestellt werden, dass sie sich performant verhält und in realistischen Projekten erfolgreich eingesetzt werden kann. Dies wurde durch den Einsatz des Aspektwebers zum Einweben der optionalen Erweiterungen des MB-Lite Prozessors demonstriert. Zuletzt wurden zwei mögliche Erweiterungen vorgestellt, die die Möglichkeiten von AspectVHDL weiter verbessern könnten.

In der Zukunft ist es zunächst wichtig, durch den nun möglichen praktischen Einsatz von AspectVHDL weitere Erfahrungen über die spezifischen Anforderungen zu sammeln, die Hardwarebeschreibungssprachen an die Konzepte der Aspektorientierung stellen. Es gilt zu ermitteln, inwieweit querschneidende Belange in Hardwareentwürfen durch die aktuellen Elemente von AspectVHDL in Aspekten isoliert werden können.

Auf Basis dieser Erkenntnisse kann der Sprachumfang von AspectVHDL gezielt erweitert werden, um die Mächtigkeit der Sprache zu erhöhen. Dieser Prozess muss Hand in Hand mit der Weiterentwicklung des Aspektwebers erfolgen – nur wenn die neuen

Ideen durch diesen umgesetzt werden können, werden sie praxistauglich. Die in dieser Arbeit vorgestellten Erweiterungsmöglichkeiten stellen hierzu erste Ansätze dar. Langfristig kann AspectVHDL so zu einem mächtigen Mittel zur besseren Modularisierung von Hardwareentwürfen werden und die Entwicklung von komplexer Hardware vereinfachen.

Quellenverzeichnis

[1] Mäder, Andreas: *VHDL Kompakt*. Universität Hamburg, Fachbereich Informatik, 2008 http://tams-www.informatik.uni-hamburg.de/vhdl/doc/ajmMaterial/vhdl.pdf

[2] Kiczales, G. ; Hilsdale, E. ; Hugunin, J. ; Kersten, M. ; Palm, J. ; Griswold, W. G.: An overview of AspectJ. In: *Lecture Notes in Computer Science* Bd. 2072. 2001, S. 327–353

[3] Meier, Matthias ; Hanenberg, Stefan ; Spinczyk, Olaf: AspectVHDL Stage 1: The Prototype of an Aspect-Oriented Hardware Description Language. In: *Proceedings of the 2012 workshop on Modularity in Systems Software*. New York, NY, USA : ACM, 2012 (MISS '12), 3–8

[4] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: AspectC++: an AOP Extension for C++. In: *Software Developer's Journal* 05 (2005), S. 68–76

[5] The Design Automation Standards Committee of the IEEE: *IEEE Standard 1076-1993: VHDL*. 1993

[6] Gingold, Tristan: *GHDL*. http://ghdl.free.fr/

[7] Vollmer, Marius ; Schwoerer, Ludwig: *VAUL*. http://www.vhdl.org/misc/vaul.readme.txt

[8] *FreeHDL*. http://freehdl.seul.org/

[9] Pohl, Christopher ; Paiz, Carlos ; Porrmann, Mario: vMAGIC - Automatic Code Generation for VHDL. In: *International Journal of Reconfigurable Computing* 2009 (2009)

[10] Pohl, Christopher ; Paiz, Carlos ; Porrmann, Mario: vMAGIC - Automatic Code Generation for VHDL. In: *edacentrum newsletter* 02 (2010), S. 7–10

[11] Parr, T. J. ; Quong, R. W.: ANTLR: a predicated-LL(k) parser generator. In: *Software - Practice and Experience* 25 (1995), Nr. 7, S. 789–810

[12] *Webseite von ANTLRv3*. http://www.antlr3.org/

Abbildungsverzeichnis

1.1 Visualisierung der Aufgabe des Aspektwebers 2
1.2 Darstellung der Verteilung des Codes von drei optionalen Erweiterungen der MB-Lite CPU (aus [3]) 3

3.1 Visualisierung des Ablaufs von vMAGIC (nach [9]) 14
3.2 Ausschnitt aus einem möglichen von vMAGIC erzeugten Objektbaum (nach [10]) 14
3.3 Algorithmus für den Aspektweber in Pseudocode 15

4.1 Klassendiagramm zu den für AspectVHDL erstellten Metaklassen 18
4.2 Regel zum Parsen von Prozedur-Advices in der Grammatik des VHDL-Parsers 22
4.3 Codebeispiel für einen Prozedur-Advice und Ausschnitt aus dem daraus erstellten Syntaxbaum 22
4.4 Regel zum Parsen von Prozedur-Advices in der Grammatik des Baum-Parsers 24
4.5 Sequenzdiagramm zum Ablauf des Aspektwebers 26
4.6 Visualisierung der Positionen im Metaobjektbaum, an denen Join Points liegen können 27
4.7 Schematische Darstellung der Vorgehensweise zum Einweben von Prozeduren 31
4.8 Übersicht über die Verarbeitung des Aspectcodes 33

5.1 Darstellung der Laufzeiten von Parser, Weber und Ausgabe bei unterschiedlichen Anzahlen von Join Points 36
5.2 Darstellung der Laufzeiten von Parser, Weber und Ausgabe bei unterschiedlichen Anzahlen von Advices 37