technische universität
dortmund

Bachelorarbeit

Erweiterung von AspectC++ um ein Annotationskonzept

Uriel Elias Wiebelitz
12. Mai 2016

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M.Sc. Ulrich Gabor

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

Zusammenfassung

In dieser Arbeit wird untersucht, wie die aspektorientierte Sprache AspectC++ um die Verwendung von Attributen zur Identifizierung von Join Points erweitert werden kann. Es wird ein Konzept erarbeitet, in dem die Eigenschaften von Attributen gewahrt bleiben und ihnen zugleich die für den Einsatz in AspectC++ nötige Flexibilität verliehen wird. AspectC++ ist eine Erweiterung von C++, mit deren Hilfe Aspekte in den Quellcode eingewoben werden können. Dabei werden reine Source-to-Source-Transformationen durchgeführt, d.h. die Ausgabe von AspectC++ ist wiederum C++-Code, der mit einem entsprechenden Compiler übersetzt werden kann. Zur Realisierung von Parametern für Attribute wird im umgesetzten Konzept von Templates Gebrauch gemacht.

Abschließend wird das erarbeitete Konzept anhand eines in AspectC++ implementierten Mechanismus zur Fehlerkorrektur (GOP) auf seine Einsetzbarkeit hin untesucht. Außerdem wird eine kurze Betrachtung der Speicher- und Laufzeiteffizienz des von AspectC++ ausgegeben Codes bzw. des letztlich binären Maschinencodes vorgenommen.

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel sollen zunächst die Motivation, diesen Themenbereich zu beforschen, sowie die damit verbundenen Ziele, dargelegt werden. Abschließend wird die Struktur dieser Arbeit erläutert.

1.1 Motivation

In seiner aktuellen Fassung unterstützt AspectC++ bereits verschiedenste Pointcut-Ausdrücke, um Join Points zu identifizieren. Jedoch ermöglichen sie lediglich die Identifizierung anhand der Typstruktur oder der Bezeichner der Codeelemente. Möchte man nun bereits beim Entwickeln der Programmlogik Join Points schaffen, die später gemeinsam über einen Pointcut erfasst werden können, sollten die Bezeichner einer gemeinsamen Struktur folgen (bspw. ein gemeinsames Prefix haben). Zwar kann man die betroffenen Stellen auch alle einzeln disjunktiv auflisten, dies entspricht jedoch nicht einer generischen und gut wartbaren Lösung. Eleganter ist es, in solchen Fällen die entsprechenden Elemente mittels Metainformationen zu gruppieren und sie daran später als Join Points zu identifizieren.

Auch in anderen Sprachen, wie etwa AspectJ, hat die Identifizierung von Join Points anhand von Metainformationen bereits Einzug genommen (siehe [18, Kapitel 2]). Da mit C++11 Attribute in den C++-Standard aufgenommen wurden, bietet sich an, diese auch in AspectC++ zur Vermittlung und Verarbeitung von Metainformationen zu nutzen, da man somit auch unabhängig von AspectC++ übersetzbaren Code erzeugen kann.

Im Folgenden wird anhand einiger Beispiele illustriert, wozu Attribute bei der Identifizierung von Join Points in AspectC++ eingesetzt werden können.

Singleton

Ein häufig in der Softwareentwicklung verwendetes Muster ist das des *Singleton*, dessen Hauptmerkmal ist, dass im gesamten Projekt nur eine Instanz einer bestimmten Klasse existieren darf. Da nicht alle Klassen, auf die diese Eigenschaft zutrifft, korrekt als Singleton implementiert werden, bietet sich für solche Fälle eine aspektbasierte Überwachung der Einhaltung wie in Beispiel 1.1 an.

```
1   //One counter for each class
2   template<typename T> struct Counter {
3       static int val;
4   };
5   template<typename T> int Counter<T>::val = 0;
6
7   namespace myN { attribute SingletonClass(); } //declare attribute
8
9   aspect SingletonGuard {
10      // count constructions and destructions of annotated classes
11      advice construction("[[myN::SingletonClass]]") : before() {
12          Counter<JoinPoint::That>::val++;
13          if(Counter<JoinPoint::That>::val > 1) throw
                "SingletonViolated";
14      }
15
16      advice destruction("[[myN::SingletonClass]]") : after() {
17          Counter<JoinPoint::That>::val--;
18      }
19  };
```

Beispiel 1.1: SingletonGuard (in Anlehnung an [21, S. 26]).

In dem Beispiel wird pro Klasse, die mit `[[myN::SingletonClass]]` annotiert wurde, ein Zähler angelegt, der bei jeder Konstruktion eines Objekts inkrementiert und bei jeder Destruktion dekrementiert wird. Sobald dieser Wert „1“ übersteigt wird eine entsprechende Ausnahme geworfen.

Parallelisierung

Eine weitere interessante Metainformation kann sein, dass bestimmte Codeabschnitte parallelisiert – also auf mehreren CPU-Kernen gleichzeitig – ausgeführt werden können. Es gibt bereits Frameworks, die sich mit der Herausforderung der Parallelisierung befassen, wie etwa das weit verbreitete OpenMP. Dieses sieht für C++ zur Markierung von parallelisierbaren Stellen eine recht komplizierte Schreibweise mittels `#pragma` vor (siehe Beispiel 1.2). Mit Hilfe von Attributen wäre hier eine einfachere und besser lesbare Schreibweise denkbar, wie in Beispiel 1.3 zu sehen ist.

Weitere Einsatzbeispiele werden später noch bei der näheren Betrachtung von Join Points, die mittels Attributen identifizierbar sein sollen, angeführt.

Da Attribute bisher ausschließlich dazu vorgesehen sind, dem Compiler Metainformationen zur Verfügung zu stellen, sie allerdings nicht vom Entwickler selbst verwertet werden könnten, kann eine entsprechende Umsetzung in AspectC++ die

```
1  //parallel
2  #pragma omp parallel
3  {
4      //for-loop
5      #pragma omp for
6      for(int n=0; n<10; ++n) {
7          //parallel code
8          #pragma omp barrier
9      }
10 }
```

Beispiel 1.2: Parallelisierung mittels OpenMP.

```
1  //parallel
2  using [[omp::parallel()]]
3  {
4      //for-loop
5      for [[omp::for()]] (i=0; i<10; i++) {
6          //parallel code
7          //barrier (at end of block)
8          using [[omp::barrier]] {
9              //parallel code
10         }
11     }
12 }
```

Beispiel 1.3: Parallelisierung mit Attributen, in Anlehnung an Beispiel aus [13].

erste Möglichkeit sein, diese Attribute auch für den Entwickler verfügbar zu machen und ihren Einsatz damit bedeutend flexibler zu gestalten.

1.2 Ziele

Schon aus den ersten Überlegungen und der bisherigen Praxis in der Entwicklung von AspectC++ ergeben sich folgende Ziele:

Spracherweiterung um Annotationskonzept

AspectC++ soll um ein Annotationskonzept erweitert werden, das es ermöglicht, Aspekten Metainformationen zu Join Points verfügbar zu machen und außerdem auch das Matching auf der Grundlage von Attributen durchzuführen.

Dabei soll der Entwickler nicht an bereits existente Attribute gebunden sein, sondern auch selbst Attribute anlegen können.

Performance des generierten Codes

Grundsätzlich besteht der Anspruch an Entwicklungswerkzeuge, möglichst performanten Code zu produzieren. Dies gilt besonders für AspectC++, da es auch zur Entwicklung und Ergänzung von Betriebssystemen eingesetzt wird (siehe bspw. [12]). Daher sollte die Erweiterung von AspectC++ auch weiterhin zu performanten Codegenerierungen in Hinsicht auf Speicherbedarf und Ausführungszeit führen.

Größtmögliche Kompatibilität

Um auch für ältere Projekte einsetzbar zu sein, die noch nicht den C++11-Standard oder neuer verwenden, sollte AspektC++ auch weiterhin zu älteren Standards sowie allen bisher unterstützten Plattformen kompatibel bleiben. Nach Möglichkeit sollte die Erweiterung von AspectC++ auch bei solchen älteren Projekten einsetzbar sein, auch wenn bei Attributen auf Sprachkonstrukte zurückgegriffen wird, die noch nicht in diesen älteren Standards vorgesehen sind.

1.3 Struktur der Arbeit

Eingangs wird näher auf die Grundlagen eingegangen. Dazu wird kurz erläutert, was aspektorientierte Programmierung ausmacht, und die zugehörige Terminologie kurz vermittelt. Anschließend werden Annotationen zunächst allgemein, schließlich

auch in verschiedenen Sprachen, betrachtet, womit die notwendigen Grundlagen dargelegt sind.

Darauf folgend wird untersucht, welche Anforderungen die in dieser Arbeit betrachtete Erweiterung erfüllen muss, sowie ein Entwurf der benötigten Sprachelemente erarbeitet. Im Kapitel 5 „Umsetzungskonzept“ werden verschiedene Wege zur Umsetzung der Erweiterung abgewogen und auf Ihre Realisierbarkeit hin untersucht. In Kapitel 6 „Implementierung“ wird genauer beschrieben, welche Transformationen zum erzeugen des Codes erforderlich sind, sowie welche Teile von AspectC^{++} in welcher Form modifiziert werden müssen. Abschließend wird die erarbeitete Lösung anhand von Anwendungsbeispielen im Hinblick auf Funktionalität und Performanz untersucht, worauf eine Zusammenfassung und Überlegungen zu weiteren Verbesserungen folgen.

2. Aspektorientierte Programmierung

Das Paradigma der aspektorientierten Programmierung, das 1997 von XEROX-PARC entwickelt wurde [4, Kapitel 5.1], stellt eine Erweiterung der objektorientierten Programmierung dar. Auch schon in der Objektorientierung besteht das Ziel der Modularisierung und Vermeidung von Codewiederholungen. Dieser Gedanke wird in der aspektorientierten Programmierung noch weiter fortgeführt, was in diesem Kapitel näher beleuchtet wird.

2.1 Ziele

Mit immer größeren Softwareprodukten wächst auch ständig ihre Komplexität. Um dennoch den Überblick zu behalten und effizient entwickeln zu können, wurden Paradigmen entwickelt, die dem Entwickler einen strukturierten Umgang mit der Komplexität ermöglichen sollen.

2.1.1 Ziele der Objektorientierung

Das Paradigma der Objektorientierung verfolgt zur Strukturierung verschiedene Ansätze, von denen im Folgenden ein paar wesentliche näher erläutert werden:

Modularisierung

Das komplexe Softwareprodukt wird in kleinere Module aufgeteilt, die jeweils nur für einen Teil der Funktionalität zuständig sind. Solche Module können weiter in Module zerlegt werden, sodass man schließlich kleine Teile erhält, die leicht zu erfassen, zu warten und zu testen sind. [4, Kapitel 2], [10, Kapitel 3]

Kapselung

Um Module unabhängig voneinander entwickeln zu können, müssen sie vor unübersehbaren Veränderungen von außen geschützt werden. Dazu kann genau festgelegt

werden, welche Module auf Teile des jeweiligen Moduls zugreifen können und welche Module diese verändern dürfen. [5, Kapitel 9]

Nutzung von Gemeinsamkeiten

Insbesondere vor dem Hintergrund der Wartbarkeit sind Wiederholungen im Quelltext unbedingt zu vermeiden. Daher sollten ähnliche Module auch voneinander profitieren können, was durch Vererbung von Eigenschaften und Polymorphie ermöglicht wird. [10, Kapitel 3.3]

2.1.2 Ziele der aspektorientierten Programmierung

Häufig werden in objektorientierten Anwendungen jedoch auch Operationen modulübergreifend angewendet.

Beispielsweise wird in vielen Softwareprodukten Logging eingesetzt, was in verschiedensten Modulen entsprechende Ausgaben erzeugt. Auch die Realisierung der Persistenz von Objekten erfordert meist Wiederholungen von Quelltext etwa für Datenbankzugriffe.

Der Umstand, dass solche Anliegen über verschiedenste Module verteilt sind, wird als *Code-Streuung* (engl. *C*ode Scattering) bezeichnet [10, Kapitel 9.1]. Code-Streuung hat häufig zur Folge, dass ein Modul nicht nur ein Anliegen beinhaltet, was als *Code-Verwirrung* (engl. *Code Tangling*) bezeichnet wird.

Übergreifende Anliegen

Bei Code-Streuung können die modulübergreifenden Anliegen also nicht einzelnen Modulen zugeordnet werden, sodass mehrere Module dasselbe Anliegen erfüllen müssen. Dieses Anliegen bezeichnet man als *übergreifendes Anliegen* (*Crosscutting Concern*). Eine Veranschaulichung bietet die Abbildung 2.1.

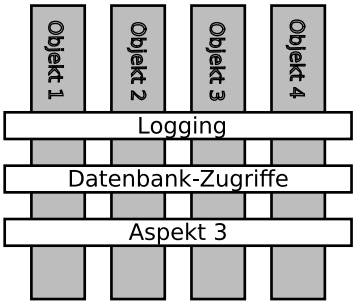

Abbildung 2.1.: In Anlehnung an [5, Abb. 19-2].

Da die Anliegen der Module selbst und die modulübergreifenden Anliegen (bspw. Logging, Persistenz) vertikal zueinander liegen, können sie nicht in einer hierarchischen Struktur angelegt werden.

Problematik Codewiederholungen

Der objektorientierten Programmierung folgend müssen diese *übergreifenden Anliegen* in jedem Modul erneut implementiert werden, was häufig zu Codewiederholungen führt. Zwar besteht die Möglichkeit, durch Vererbung (insbesondere auch der Beerbung mehrerer Klassen) verschiedenste Anliegen in einer Klasse zusammenzuführen, jedoch bedeutet die Mehrfachvererbung häufig, dass der *this*-Pointer zu manipulieren ist, was einen Overhead bedeuten würde (vgl. [8, Abschnitt 2.3.5]). Somit ist auch diese Variante nicht optimal.

Codewiederholungen gilt es aus verschiedenen Gründen zu vermeiden:

- Die **Wartung** von sich wiederholendem Code ist nahezu unmöglich, da bei Feststellung eines Fehlers oder der Notwendigkeit von Anpassungen alle Codestellen gefunden und editiert werden müssen, die davon betroffen sind.
- **Erweiterungen** werden deutlich erschwert, da auch hierzu jede einzelne Stelle editiert werden muss.
- **Wiederverwendbarkeit** ist kaum gegeben, da der Code meist kopiert werden muss und jedes Mal neu an den Kontext angepasst wird.
- Das **Verständnis** des Codes wird massiv erschwert, da der Code an jeder Verwendungsstelle in seinem jeweiligen Umfeld nachvollzogen werden muss.

2.2 Grundlagen der aspektorientierten Programmierung

Die Realisierung von übergreifenden Anliegen soll durch die aspektorientierte Programmierung vereinfacht werden.

Dazu muss die Möglichkeit geschaffen werden, Code an einer Stelle zu definieren und an verschiedenen Punkten in die Software einzubringen. Außerdem muss der einmalig definierte Code an seinen jeweiligen Kontext angepasst und ihm ggf. Zugriff auf selbigen gewährt werden können. Eine Herausforderung der aspektorientierten Programmierung stellt die eindeutige Identifizierung solcher Einbindungspunkte im Softwaresystem dar.

2.2.1 Join Points

Als Join Points werden identifizierbare Stellen im Programm bezeichnet, an denen solche Einfügungen vorgenommen werden können. Dabei wird zwischen den grundlegenden Typen Name und Code Join Point unterschieden:

2.2.1.1 Name Join Points

Name Join Points sind Einbindungspunkte im Programm, die sich auf dessen Struktur beziehen. Dabei handelt es sich um C++-Elemente, wie etwa Funktionen, Variablen, Klassen, Typen, oder Namensräume.

2.2.1.2 Code Join Points

Code Join Points bezeichnen Ausführungspunkte. Dies können bspw. Methodenaufrufe, Methodenausführungen, lesende bzw. schreibende Zugriffe oder Konstruktionen/Zerstörungen von Objekten sein.

2.2.2 Pointcut

Pointcuts sind eine Menge von Join Points und bilden somit eine Sammlung von Punkten, an denen Code eingewoben werden soll.

2.2.3 Pointcut Expression

Zur Identifizierung von Join Points eines Pointcuts werden Muster (sogenannte Pointcut Expressions) verwendet, wobei ggf. auch mehrere Join Points in einem Pointcut erfasst werden können.

2.2.4 Advice

Als Advice wird ein Codefragment bezeichnet, „das beim Erreichen eines Join Points ausgeführt wird“ [4, Kapitel 6.3.2]. Dabei kann der Ausführungszeitpunkt genauer spezifiziert werden (siehe [21, Kapitel 2.3]):

- *around*: Der ursprüngliche Join Point Aufruf wird ersetzt durch den eingefügten Advice. Dadurch kann auch Einfluss darauf genommen werden, ob der Join Point ausgeführt wird. Soll der ursprüngliche Programmcode zur Ausführung kommen, kann dies mit einem speziellen Befehl ausgelöst werden.

- *before*: Der Advice wird unmittelbar vor dem genannten Pointcut ausgeführt.
- *after*: Der Advice wird unmittelbar nach dem genannten Pointcut ausgeführt.

2.3 Aspekt

Ein Aspekt ist eine Ansammlung von Einfügungen und Advice-Code, die ein überschneidendes Anliegen implementiert. Er stellt in gewisser Weise eine Erweiterung des Konzepts der Klasse dar. Somit kann ein Aspekt auch Variablen und Methoden beinhalten sowie andere Klassen oder Aspekte beerben. [21, Kapitel 2.4]

3. Überblick über Annotationen

Annotationen stehen in verschiedenen Sprachen in unterschiedlichen Formen zur Verfügung. Dabei muss die Sprache nicht grundsätzlich aspektorientiert sein. Die wesentlichen Grundlagen sowie Beispiele aus verschiedenen Programmiersprachen werden in diesem Kapitel erarbeitet.

3.1 Grundlagen

Annotationen ermöglichen die Vermittlung von Metainformationen an den Compiler. Hierbei können einzelne Sprachelemente, denen bestimmte Informationen hinzugefügt werden sollen, annotiert werden.

Zwar liefert auch der Quellcode an sich schon sehr viel relevante Information, wie beispielsweise Typen, die durch Annotationen jedoch noch entscheidend erweitert werden können.

Um Pointcuts (siehe Abschnitt 2.2.2) zu definieren, die alle Funktionen mit einer bestimmten Eigenschaft erfassen, müssen diese entweder einzeln aufgelistet werden, eindeutig in ihren Typen sein oder aber Ähnlichkeiten in den Bezeichnern aufweisen. Mit Hilfe von Annotationen können Informationen über solche gemeinsamen Eigenschaften unabhängig von den Bezeichnern und der sonstigen Programmstruktur geliefert werden. [10, Kapitel 9.4]

In Programmiersprachen müssen Doppeldeutigkeiten mit anderen Sprachkonstrukten ausgeschlossen werden. Hierzu existieren verschiedene eindeutige Schreibweisen in den jeweiligen Sprachen. Häufig werden Annotationen eingesetzt, um dem Compiler besondere Anhaltspunkte für Optimierungen oder Überprüfung der Korrektheit zur Verfügung zu stellen.

Um die ausgedrückten Metainformationen weiter zu verfeinern, besteht häufig die Möglichkeit, Parameter zu übergeben. Darauf wird im später in diesem Kapitel eingegangen.

Im Folgenden soll näher erläutert werden, wie und in welchem Umfang Annotationen in den Sprachen Java, AspectJ, C# bereits eingesetzt werden können.

3.2 Java

Auch wenn Java an sich nicht aspektorientiert ist, besteht bereits die Möglichkeit, bestimmte Elemente mit Zusatzinformationen zu versehen.

3.2.1 Syntax

Um Annotationen eindeutig zu kennzeichnen, wird dem Annotationsnamen ein `@` vorangestellt. Optional können runde Klammern mit durch Kommata getrennten Parametern folgen: [20, Kapitel 3.10.2]

```
@SuppressWarnings("all")
public class NoWarnings{ }
```

Wenn ein Attribut mehrere Parameter-Werte erhält, muss der jeweilige Parameter, dem der übergebene Wert zugeordnet werden soll, einzeln namentlich genannt werden.

3.2.2 Benutzerdefinierte Annotationstypen

Neben den bereits eingebauten Annotationen ist es in Java möglich, weitere eigene Annotationen selbst zu definieren. Zudem kann gesteuert werden, wie lange sie im Übersetzungsprozess mitgeführt werden sollen: Angelegte Annotationen können selbst mit `@retention` annotiert werden. Dabei wird über den Parameter vermittelt, wo die Annotation überall verfügbar sein soll [19, Kapitel 25.6.8]:

`SOURCE`

Bei diesem Parameterwert ist die jeweilige Annotation nur im Quelltext verfügbar. Somit kann sie in Vorverarbeitungsschritten oder vom Compiler selbst ausgewertet werden, ist jedoch nicht im vom Compiler generierten Bytecode wiederzufinden.

`CLASS`

Wird ein Annotationstyp mit diesem Parameterwert versehen, wird die Annotation zwar in die jeweilige Klassendatei im Bytecode übernommen, jedoch nicht der Laufzeitumgebung zur Verfügung gestellt.

`RUNTIME`

Dieser Parameterwert bewirkt, dass die annotierte Annotation auch zur Laufzeit noch erreichbar ist. Sie wird also genau wie bei `CLASS` im Bytecode abgelegt und darüber hinaus auch der Laufzeitumgebung zugänglich gemacht.

3.2.3 Parametertypen

Als Parametertypen sind folgende Typen zulässig [6, Kapitel 9.6.1]:

- primitive Datentypen (z. B. `bool`, `byte`, `int`),
- der Typ `String`,
- der Typ `Class` (Metainformationen zu einer Klasse),
- Enumerationstypen,
- Annotationstypen und
- Arrays der oben genannten Typen.

3.3 AspectJ

Seit Version 1.5 (AspectJ5) erlaubt AspectJ Join Point Matching auf der Grundlage von Annotationen. Da AspectJ auf Java basiert, entspricht auch die Schreibweise solcher Annotationen der von Java.

Zudem ist die Syntax für Pointcuts wie folgt festgelegt:

`@foo` stimmt mit jedem Element überein, das mit `@foo` annotiert wurde.

`@foo @bar` stimmt mit Elementen überein, die sowohl mit `@foo` also auch mit `@bar` annotiert wurden.

`@(foo || bar)` stimmt mit Elementen überein, die mit `@foo` oder `@bar` annotiert wurden.

In dieser Form sind auch andere logische Verknüpfungen wie etwa die Negation ausdrückbar.

Darüber hinaus können auch Pointcuts definiert werden, deren Syntax stark an die der Pointcuts ohne Annotationen angelehnt ist [18]:

- `@(org.xyz..*)` stimmt mit Elementen überein, die mindestens ein Attribut aus dem Paket `org.xyz` oder aus dessen Subpaketen tragen.

- `* *.*(@foo *,..)` stimmt mit allen Methoden überein, die mindestens einen Parameter haben, dessen Typ mit `@foo` annotiert ist.

Dadurch ist es sogar möglich, Annotationen an bestimmten Teilen einer Funktionssignatur zu identifizieren.

Des Weiteren können mittels Annotationen auch Pointcuts und Advices (siehe Kapitel 2 „Aspektorientierte Programmierung“) deklariert werden, wie in Beispiel 3.1 veranschaulicht.

```
1 @Pointcut("execution(* *.*(...))")
2 void anyExec() {}
```

Beispiel 3.1: Pointcut als Annotation in AspectJ.

3.4 C#

In C# stehen zur Ergänzung von Metainformationen Attribute zur Verfügung.

3.4.1 Syntax

Attribute werden in einfachen eckigen Klammern vor der jeweiligen Deklaration angegeben. Dabei können optional in runden Klammern Parameter hinzugefügt werden, die entweder anhand ihrer Position oder explizit mitgelieferter Bezeichner zugeordnet werden können. Häufig wird letztere Variante bei optionalen Parametern eingesetzt. Das Beispiel 3.2 illustriert eine solche Verwendung.

```
1 [ Obsolete ( " Use method X instead . " ) ]
2 public void Bar ( int param0 ) {}
```

Beispiel 3.2: Attribut mit Parametern in C#, siehe Beispiel aus [15, Kapitel 21.1].

3.4.2 Benutzerdefinierte Attribute

Auch in C# ist die Definition von benutzerdefinierten Attributen möglich. Hierzu wird eine Klasse angelegt, die die im Namensraum „System“ liegende Klasse `Attribute` beerbt.

Ebenso ist es auch hier (wie in Java) möglich, Attribute selbst zu attributieren und somit beispielsweise Elementtypen festzulegen, für die das jeweilige Attribut verwendet werden kann.

3.4.3 Parametertypen

Die für Parameter verwendbaren Typen sind auf die folgenden beschränkt [11, Kapitel 17]:

- primitive Datentypen (z. B. `bool`, `byte`, `int`),
- der Typ `object`,
- der Typ `System.Type`,
- Enumerationstypen (die als `public` zugänglich sind) und
- eindimensionale Arrays der oben genannten Typen.

3.5 AspectC++ Vorarbeiten

Bereits im Jahr 2011 wurde eine um ein Annotationskonzept erweiterte Fassung von AspectC++ erarbeitet und implementiert. Die dazugehörige Ausarbeitung trägt den Titel „Metadata Annotations for Explicit Join Points in AspectC++“ [14]. Die damalige Umsetzung orientierte sich insbesondere syntaktisch stark an Annotationen aus AspectJ.

In der Zwischenzeit haben sich folgende Überarbeitungspunkte ergeben:

Definition der Semantik

Aus der Ausarbeitung geht nicht klar hervor, zu welchem Zeitpunkt an Annotationen übergebene Parameter ausgewertet werden. Daher wird nicht deutlich, ob Argumente bereits zur Übersetzungszeit auswertbar sein müssen oder ob es ausreicht, wenn sie zur Laufzeit ausgewertet werden können.

Nicht-C++-Sprachelemente

Da der C++11-Standard zu diesem Zeitpunkt noch sehr neu war, wurden Möglichkeiten, die sich mit dem seinerzeit neuen Standard boten, noch nicht berücksichtigt. Dies hat zur Folge, dass die verwendete Syntax für Annotationen, welche als Kennzeichnung ein `@` verwendet, nicht im C++-Standard vorgesehen ist und ein standardkonformer Compiler den Quelltext nicht mehr direkt übersetzen kann.

Attribute an Funktionsdeklarationen

Zu den Orten der Annotationen für Funktionen schreibt der Autor im Abschnitt 5.3.2.3 „Functions“, es werde die Funktionsdefinition gewählt. Diese Entscheidung hat zur Folge, dass annotierte Funktionen nur dann zugeordnet werden können, wenn sich die Definition in der gleichen Übersetzungseinheit befindet wie die Verwendung der Annotation. Hier ist eine Lösung erstrebenswert, die sämtliche Annotationen zuordnen kann.

3.6 Attribute in C++

In C++ können zusätzliche Informationen über verschiedene Elemente mit Hilfe von Attributen spezifiziert werden. Seit C++11 gehören sie zum Standard und werden somit auch von den gängigen Compilern bereits unterstützt. Sie sind zur Informationsvermittlung an den Compiler konzipiert. So können Funktionen beispielsweise mit dem ebenfalls im Standard definierten Attribut `[[noreturn]]` versehen werden, wodurch gekennzeichnet wird, dass diese Funktion niemals zurückkehren sollte (z.B. `exit()`). Der Compiler kann auf Grund solcher Zusatzinformationen besondere Optimierungen vornehmen oder zusätzliche Warnungen bei Missachtung generieren.

Diese Art der Informationsergänzung ist auch in C++ nicht gänzlich neu. Sie wird bereits seit langem von verschiedenen Compilern in unterschiedlichen Formen bereitgestellt. Clang und GCC bspw. verfügen hierzu über das Schlüsselwort `__attribute__`.

Der C++11-Standard sieht Attribute der Form `[[`*attribute-specifier* (*parameter*)`]]` vor. Hierbei können auch mehrere Attribute in denselben oder auch getrennten eckigen Klammern aufeinander folgen:

```
[[Attr1, Attr2]] [[Attr2]] void myFunc();
```

Da Attribute bisher noch nicht sehr verbreitet sind, ist die genaue Verwendung von Attributen teils unzureichend dokumentiert. Auch lässt sich unterschiedliches Verhalten zwischen den Compilern G++ und Clang in Bezug auf Attribute, die

nicht bei der ersten Deklaration (etwa einer Funktion) angegeben wurden, feststellen: Clang verlangt eine Attributierung auch an der ersten Deklaration, während G++ den Vorgang ohne Meldung abschließt. Im Standard ist hierfür keine grundsätzliche Regelung vorgegeben, stattdessen wird dies für Attribute wie `noreturn` oder `carries_dependency` explizit gefordert.

Insbesondere sei in diesem Zusammenhang darauf hingewiesen, dass G++ in der Version 5.2.1, unter Verwendung des C++11-Standards, Attribute für Namensräume an der im Standard vorgesehenen Position (direkt hinter `namespace`) [7, Kapitel 7.3.1] nicht korrekt verarbeitet und eine Fehlermeldung ausgibt.

Um Namensraumüberschneidungen vorzubeugen, können Attribute einem bestimmten Namensraum zugeordnet sein. In solchen Fällen ist selbiger stets mit anzugeben:

```
[[clang::noduplicate]]
```

3.6.1 Attributierbare Elemente

Die in C++11 eingeführten Attribute können an zahlreichen C++-Konstrukten eingesetzt werden. Neben *Namespaces*, *Classes*, *Structs* und *Functions* können auch *Returntypes*, *Parameter* und sogar *Statements* oder einzelne Variablendeklarationen attributiert werden.
Im Beispiel 3.3 werden verschiedene Attribute und ihre jeweiligen Bezugselemente verdeutlicht.

```
1 namespace [[Namespace_Attr]] MyNamespace {}
2
3 class [[Class_attr]] MyClass {};
4
5 [[Function_Attr]] int [[ReturnType_Attr]] myFunc [[Function_Attr2]]
6         ([[Param_Attr]] int [[ParamType_Attr]] p1 [[Param_Attr2]] ) {
7
8   if(true) [[Statement_Attr]] {  /*do something*/  }
9 }
```

Beispiel 3.3: Attribute im Quelltext.

Bei Funktionen, Variablen und Parametern ist zu beachten, dass Attribute, die sich auf die Funktion, die Variable oder den Parameter selbst beziehen, sowohl ganz am Anfang der Deklaration als auch nach dem Bezeichner stehen können (siehe `[[Function_Attr]]` und `[[Function_Attr2]]`).

Ist ein verwendetes Attribut dem Compiler nicht bekannt, so ist der Umgang mit diesem implementierungsabhängig. Es ist somit dem Compiler überlassen, wie er in solchen Fällen vorgeht. [7, Kapitel 7.6.1]

3.6.2 Parameter

Ebenso wie bisherige Attributkonstrukte der jeweiligen Compiler Parameter unterstützen, sieht auch der C++11-Standard solche Parameter vor. Üblicherweise handelt es sich um Stringliterale, Ganzzahlen oder Enumerationswerte. Da es sich um Informationen für den Compiler handelt, muss selbiger sie auch auswerten können, woraus folgt, dass als Parameter nur Werte in Frage kommen, die zur Übersetzungszeit bereits auswertbar sind.

3.6.3 Benutzerdefinierte Attribute

Der C++-Standard sieht bisher keine vom Entwickler selbst angelegten Attribute vor. Daher ist es nicht möglich, auf spracheigenem Wege wesentliche Informationen wie Parametertypen oder für dieses Attribut geeignete Elementarten im Quellcode zu hinterlegen.

4. Sprachentwurf

Der Sprachentwurf gehört zu den wesentlichen Schritten bei der Erweiterung einer Programmiersprache. Dabei muss besondere Rücksicht auf die Einhaltung der Anforderungen genommen werden. Im Folgenden werden zunächst die Anforderungen dargelegt und schließlich sowohl die Syntax als auch die Semantik der Erweiterung von AspektC++ um Annotationen erörtert.

4.1 Anforderungen

Im folgenden Kapitel werden die verschiedenen Grundanforderungen an die Erweiterung von AspectC++ um Attribute dargestellt und begründet. Dabei werden sowohl der Umgang mit Compilern als auch die Einbettung in die Sprache betrachtet.

4.1.1 Kompatibilität

Da Softwareprojekte häufig über längere Zeiträume entwickelt werden und somit stets auch bereits bestehender Code in selbige einfließt, entstehen bei der Erweiterung von Sprachen einige Anforderungen an die Kompatibilität zu älteren Sprachvarianten und den verwendeten Compilern, auf die in diesem Abschnitt näher eingegangen wird.

4.1.1.1 Compiler

AspektC++ unterstützt bisher die beiden Frontends PUMA und Clang für die Codeanalyse und -transformation. Dabei handelt es sich um Source-to-Source-Transformationen, d.h. die Ausgabe ist wiederum C++-Code, der schließlich mit G++ übersetzt werden kann. Somit müssen all diese drei Compiler mit der Erweiterung umgehen können. Dies stellt insbesondere eine Herausforderung dar, weil sich die Compiler in Bezug auf Attribute unterschiedlich verhalten, wie bereits in Abschnitt 3.6 dargelegt.

Darüber hinaus soll der Quellcode auch ohne Transformationen standardkonform sein und somit von C++-Compilern, die mindestens den C++11-Standard oder neuer unterstützen, direkt übersetzt werden können.

4.1.1.2 C++

Gerade bei der Programmiersprache C++, die 1985 auf der Grundlage von C veröffentlicht wurde, wird bereits seit ihren Anfängen bei der Entwicklung stets auch die Kompatibilität berücksichtigt. Auch wenn es nie Ziel war, 100%ig kompatibel zu C zu sein [17, Kapitel 1.2.3], bietet die Nähe zu C für viele Projekte Möglichkeiten, bisherigen Quellcode zu übernehmen oder zu portieren.

Daher nimmt der Anspruch, kompatible Sprachstandards zu entwerfen, bereits seit der „Geburtsstunde“ von C++ großen Einfluss auf seine Entwicklung.

Ältere C++-Standards

AspectC++ soll nicht als eine Erweiterung des jeweils neuesten C++-Standards verstanden werden, sondern sollte vielmehr als eine Erweiterung von C++ im Allgemeinen betrachtet werden. Daraus ergibt sich, dass Erweiterungen von AspectC++, wie etwa Annotationen, nicht erst durch neueste Standards unterstützt werden sollten, sondern auch in Projekten, die noch mit älteren Standards arbeiten, zur Verfügung stehen sollten.

Die Herausforderung besteht darin, den erzeugten Quellcode möglichst abwärtskompatibel zu gestalten und somit keine Sprachkonstrukte zu verwenden, die erst in neueren Standards berücksichtigt und interpretiert werden können.

Bereits vorhandene Attribute

Gleichzeitig darf natürlich nicht die Semantik von Attributen, die im Standard definiert sind, verfälscht werden, sodass ausschließlich benutzerdefinierte Attribute aus dem Quelltext verändert werden dürfen und bekannte und unabhängig von AspectC++ verwendbare Attribute erhalten bleiben müssen.

Da Fehler- und Warnmeldungen generiert werden, wenn ein unbekanntes Attribut verwendet wird, sind Attribute, die Clang bzw. G++ nicht kennen, aus dem Quelltext zu entfernen.

4.1.1.3 Bisherige Join Point API

Bisher ist die Attributierung von Elementen in der Join Point-API noch nicht vorgesehen. Um Attribute tatsächlich auch zum Matching verwenden zu können,

müssen sie auch in Matchexpressions ausdrückbar sein. Dabei muss besonders auf die Merkmale Eindeutigkeit und intuitiven Gebrauch geachtet werden:

Eindeutigkeit
: Um nicht unvorhergesehen andere Elemente durch Matchexpressions zu referenzieren, müssen sich Pointcut-Ausdrücke zu Attributen von denen bisheriger Sprachkonstrukte unterscheiden.

Entwickler-„Intuition“
: Damit Spracherweiterungen Anklang finden und für den Entwickler verständlich sind, sollten Erweiterungen sich an bereits Bekanntem orientieren. Daher sollten Pointcut-Ausdrücke für Attribute in Struktur und Verwendung in Anlehnung an bisherige entworfen werden.

4.1.2 Statische Parameter

Parameter ermöglichen eine bedeutend flexiblere Nutzung von Annotationen. Attribute sollen zur Identifizierung von Join Points geeignet sein. Demzufolge sollten auch ihre Parameter mit einbezogen werden können. Da Attribute, wie bereits im Abschnitt 3.6 erläutert, für die Informationsvermittlung an den Compiler entworfen wurden, müssen die Parameter in jedem Fall von diesem interpretierbar sein. Daraus ergibt sich, dass Parameter statisch sein müssen und ihre Werte bereits zur Übersetzungszeit auswertbar sind. Diese Eigenschaft lässt sich auch zur genaueren Identifizierung von Join Points nutzen, indem man nicht nur die Existenz oder Typen von Parametern für das Matching heranzieht, sondern auch die konkreten Werte der Parameter selbst.

Als Parameter sollen mindestens Zeichenketten und Ganzzahlen übergeben werden können, um konform zu C++-Attributen zu sein.

Gleichzeitig sollte es auch möglich sein, die Parameter innerhalb des zugehörigen Aspekts zu verwenden, sodass sie von dort aus abrufbar sein müssen.

4.2 Syntax

Die Syntax stellt einen wesentlichen Teil des Sprachentwurfs dar. Dabei muss stets berücksichtigt werden, dass sie eine für den Compiler und den Entwickler verständliche Programmierung ermöglichen sollte. In diesem Abschnitt wird näher betrachtet, mit welchen Sprachkonstrukten die Erweiterung vom Entwickler eingesetzt werden kann.

4.2.1 Attributdeklarationen

Da Anwender auch eigene Attribute verwenden können sollen, müssen dem Compiler zur Transformierung notwendige Informationen geliefert werden. Hierzu zählen insbesondere, welche Attribute überhaupt verfügbar sein sollen, sowie die Typen der Parameter. Dazu sind Deklarationen von Attributen notwendig, für welche das Schlüsselwort `attribute` zur Verfügung steht. Um Namenskonflikte in bereits bestehenden Projekten auszuschließen, wird es, wie andere AspectC++-spezifische Schlüsselwörter, standardmäßig nur in *.ah-Dateien erkannt, kann jedoch mit der Option `-k` auch in anderen Dateien verwendet werden. Um ein Attribut einem bestimmten Namensraum zuzuordnen, muss es in selbigem deklariert werden. Eine Attribut-Deklaration hat somit folgende Struktur:
`attribute` *attribute-specifier* (*attribute-argument-list*$_{opt}$)`;`

Um auch in bereits bestehenden Projekten auf existierende Attribute zurückgreifen zu können, muss die Attributdeklaration nicht zwangsläufig vor der Verwendung auffindbar sein. Mehrfachdeklarationen sollten zwar möglich sein, jedoch ist keine Überladung vorgesehen.

4.2.1.1 Parameter

Um kompatibel zu bereits existierenden Attributen zu sein, müssen mindestens ganzzahlige Parameter und Stringliterale als Typen unterstützt werden.

Diese Parameter erfüllen im Wesentlichen die folgenden zwei Funktionen:

Join Point Matching

Parameter können zur Identifizierung eines Join Points herangezogen werden. Dabei soll nicht nur der Typ des Parameters (wie es bei Funktionen üblich ist), sondern auch der Wert herangezogen werden.

Kontextinformation

Darüber hinaus soll ein Parameter einem Aspekt Informationen über den jeweiligen Kontext liefern, somit müssen die Parameter vom Aspekt aus erreichbar sein. Dies setzt voraus, dass sie im Quelltext abgelegt werden.

4.2.2 Attributverwendungen

Da nicht grundsätzlich vorgeschrieben ist, an welcher Stelle Attribute spezifiziert werden müssen, jedoch bei einzelnen Attributen explizit die Angabe an der ersten

Deklaration gefordert wird (siehe Abschnitt 3.6), ist es sinnvoll, dies grundsätzlich zu fordern. Der C++-Standard sieht für Attribute vor, dass sie teils vor dem zu attributierenden Element genannt werden müssen und dass sie teils nach dem entsprechenden Element eingesetzt werden müssen. Zugunsten der Kompatibilität werden diese Bezüge in dieser Erweiterung gemäß dem Standard übernommen.

4.2.2.1 Deklaration vs. Definition

Für Memberfunktionen und (insbesondere statische) Klassenmembervariablen wird häufig eine Unterteilung von Deklaration und Definition durch eine Aufteilung in Header- und Implementierungsdateien vorgenommen, was ein getrenntes Übersetzen der Übersetzungseinheiten ermöglicht. Darüber hinaus können Typen von Zeigermembern auch im Vorhinein deklariert werden, ohne dass die genaue Struktur des Objekts bekannt sein muss.

Dadurch kann es offensichtlich zu Mehrdeutigkeiten bei der Verwendung von Attributen kommen, da Elemente wie z. B. Klassenmember mehrere Deklarationen haben können, wohingegen nur eine Definition möglich ist. Unter diesen Gesichtspunkten kann es sinnvoll erscheinen, Attribute für Funktionen nur an den Definitionen zu erlauben, um kein Potenzial für Konflikte zu bieten.

Dieses Vorgehen birgt jedoch das Risiko, dass die Attribute einer Funktion, deren Aufruf mit einem Aspekt versehen werden soll, am jeweiligen Aufrufort nicht bekannt sind, da dort nur die Deklaration, jedoch nicht die Definition bekannt sein muss. Dementsprechend könnten solche Aufrufe nur als Join Point dienen, wenn sich aufrufende und aufgerufene Funktion in der gleichen Übersetzungseinheit befinden und somit die Definition bekannt ist.

Um dies auszuschließen, müssen Annotationen an den Deklarationen bekannt sein. Damit Annotationen nicht erst zu spät gefunden werden oder konfliktbehaftet sind, ist erforderlich, dass jede Annotation bereits an der ersten Deklaration einer Funktion vorzufinden sein muss. Optional kann eine (ggf. nicht echte) Teilmenge der Attribute der ersten Deklaration auch an weiteren Deklarationen genannt werden.

4.3 Auswertung von Attributen und Parametern

Um auch die Parameter der Attribute zur Spezifizierung von Join Points hinzuziehen zu können, müssen diese bereits bei der Übersetzung durch den Compiler ausgewertet werden können. Dies hat zur Folge, dass ausschließlich konstante Werte

verwendet werden können.

Neben der Auswertung während der Übersetzung müssen die Werte auch im Quelltext abrufbar hinterlegt werden, um eine Nutzung innerhalb von Aspekten zu ermöglichen.

4.3.1 Attributidentifizierung

Zur eindeutigen Identifizierung sind zwei Voraussetzungen essentiell: Es darf keine Doppeldeutigkeiten zwischen Attributen geben und es darf nicht zu Konflikten zwischen Attributen und anderen Elementen (z. B. Klassen oder Funktionen) kommen.

Eindeutigkeit der Attribute

Da für Attribute keine Überladung vorgesehen ist, können nicht mehrere verschiedene Attribute mit demselben Bezeichner innerhalb desselben *Scopes* existieren. Daher ist das referenzierte Attribut eindeutig anhand des verwendeten Bezeichners identifizierbar.

Bei Verwendungen von Attributen muss immer der komplette Name inklusive des Namensraums angegeben werden. Daher sind auch Überschneidungen zwischen Attributen verschiedener Scopes ausgeschlossen.

Namensüberschneidungen mit anderen Elementen

Prinzipiell ist es denkbar, dass innerhalb desselben Namensraums beispielsweise sowohl eine Funktion mit dem Bezeichner `noreturn` als auch das Attribut mit demselben Bezeichner existieren. Da der Standard diesbezüglich keine Restriktionen vorsieht, darf dadurch kein Konflikt entstehen. Dazu werden Attribute in einen anderen, eigenen Namensraum ausgelagert, welcher intern aufgelöst werden kann (außer bei Reflexion), sodass der Anwender hierdurch keine Beeinträchtigungen erfährt.

4.4 Interpretation als Code- oder Name Join Point

In vielerlei Hinsicht erinnern Attributspezifizierungen an die bereits vorhandenen Pointcuts: Ähnlich wie Pointcuts sind sie geeignet, eine Menge von Codestellen zu identifizieren. Darüber hinaus können sie ebenso Parameter erhalten und sollen

diese auch für Aspekte verfügbar machen. Dementsprechend liegt nahe, ein Attribut ähnlich zu behandeln wie Pointcuts, jedoch wirft dies Schwierigkeiten auf: Im Gegensatz zu Parametern von Pointcuts sollen bei Attributen die Werte und nicht lediglich die Typen zur Spezifizierung von Join Points herangezogen werden können. Daher müssen die Argumente bereits während der Übersetzung ausgewertet werden, sodass Attributspezifizierungen als Name Join Points interpretiert werden müssen.

4.5 Matchexpressions

Aus der Anforderung, Join Points über die Werte der Attributparameter identifizieren zu können, folgt unmittelbar, dass diese auch in die *Matchexpressions* (siehe Pointcut-Ausdrücke) einfließen müssen. Hierbei sind wiederum verschiedene Schreibweisen denkbar, nämlich eine Schreibweise, die der Attribut- bzw. Pointcutdeklaration ähnelt, oder eine, die der Verwendung von Attributen ähnelt. Diese Varianten werden in diesem Abschnitt erörtert.

4.5.1 Ähnlich wie Attributdeklarationen

Wie bereits im vorherigen Abschnitt genannt, existieren auf den ersten Blick starke Ähnlichkeiten zwischen Pointcuts und Attributen. Dementsprechend liegt nahe, die Attributdeklarationen, die bereits die Typinformationen liefern (wie auch die Pointcuts), identisch zu Pointcuts zu verwenden und ggf. auch zu verknüpfen. Problematisch ist jedoch die Auswertung von Parameterwerten, da sie für Pointcuts ursprünglich nicht vorgesehen ist und auch bei der Attributdeklaration diese Werte überhaupt nicht genannt werden. Insbesondere wäre mit der Angabe von Parameterwerten keine Umwandlung zu einer Funktion, wie sie bei Pointcuts zur späteren Typerfassung vorgenommen wird, möglich. Daher ist diese Schreibweise für Pointcut-Ausdrücke für Attribute nicht geeignet.

4.5.2 Ähnlich wie Attributverwendung

Eine weitere Möglichkeit stellt die Einführung eines neuen Musters dar. Um deutlich zu machen, dass es sich bei der aktuellen Matchexpression um ein Attribut handelt, wird es auch hier (wie schon bei der Verwendung) von zwei eckigen Klammern eingeschlossen. An dieser Stelle ist es nicht möglich, mehrere Attribute direkt aufeinander folgend zu nennen. Dabei können auch die Parameterwerte angegeben werden, die für das Matching herangezogen werden können. Ein Aufruf eines

execution-Advice, bei dem ein Attribut zur Identifizierung verwendet wird, kann beispielsweise folgendermaßen aussehen:
`execution ("[[acc::myAttr( 5 )]]")`

4.6 Annotierbare Sprachelemente

Um mittels Annotationen Join Points identifizieren zu können, muss zunächst näher untersucht werden, an welchen Join Points ihre Verwendung sinnvoll ist.

Annotationen ermöglichen, unabhängig von den bisher verwendeten Bezeichnern, Informationen zur Join Point-Identifizierung zu liefern. Daher sind sie besonders geeignet, ähnliche Eigenschaften bestimmter Codeelemente hervorzuheben, ohne dabei die Bezeichner dieser Elemente selbst heranziehen zu müssen.

Bisher müssen alle Elemente, die von einem Pointcut erfasst werden sollen, entweder einzeln aufgezählt werden oder aber Ähnlichkeiten in ihren Bezeichnern oder Typen aufweisen, um in Verbindung gebracht werden zu können. Attribute bieten eine Abstraktion von solchen Bezeichnern und ermöglichen somit eine freiere und eventuell lesbarere Wahl von Bezeichnern.

Im Folgenden werden die verschiedenen Annotierungsorte auf mögliche Einsatzfelder hin untersucht.

4.6.1 Namensräume

Auch ohne eine Erweiterung um Annotationen können Namensräume bereits in Matchexpressions zur Einschränkung des betrachteten Bereichs verwendet werden.

Hier ist es durchaus nützlich, solche Join Points auch unabhängig von ihren Bezeichnern erkennen zu können. So kann ein Entwickler etwa alle Namensräume eines Teilprojekts entsprechend annotieren und somit bei der Formulierung der Matchexpressions die Erfassung fremder Elemente ausschließen.

4.6.2 Klassen

Klassen annotierbar zu gestalten, bietet in mehrfacher Hinsicht Vorteile: Zum einen ermöglicht dies eine feinere Identifizierung des betrachteten Bereichs (vgl. Abschnitt „Namensräume“), zum anderen können hierdurch auch klassenübergreifende Mechanismen bereitgestellt werden, wie bspw. beim bereits eingangs (siehe Abschnitt 1.1 „Motivation“) genannten Beispiel eines Singletonguards.

4.6.3 Membervariablen

Die Annotation von Membervariablen bietet die Möglichkeit, einzelne Variablen mit Aspekten versehen zu können oder auch explizit davon auszuschließen. Ein Anwendungsbeispiel sind sogenannte unique IDs (einzigartige Identifikationen), die häufig lediglich zu Persistenzzwecken für Datenbankzugriffe o.Ä. vorgehalten werden. Solche Variablen könnte der Entwickler mittels Attributen explizit von Aspekten ausschließen. Ein weiteres Beispiel findet sich später in Kapitel 7 „Evaluation“.

Bereits in Kapitel 3.6.1 „Attributierbare Elemente“ wurde erwähnt, dass Attribute bei Variablen an mehreren Stellen stehen dürfen. Sie können sowohl vor der Deklaration als auch nach dem Bezeichner angegeben werden.

4.6.4 Funktionen

Für die Annotierung von Funktionen sind verschiedenste Einsatzszenarien denkbar: Es wäre möglich, bestimmte Funktionen für ein Tracing zu Debugzwecken auszuwählen oder auch solche zu markieren, vor und nach deren Ausführungen Daten gesichert werden sollten. Wie bei Membervariablen stehen auch bei Funktionen verschiedene Orte zur Attributierung zur Wahl: Sie können vor der Deklaration oder nach dem Bezeichner verwendet werden. [7, Kapitel 7, Abs. 2]

```
1 [[noreturn]] void f [[noreturn]] (); // OK
```

Beispiel 4.1: Orte für Attribute an Funktionen, aus [7, Kapitel 7, Abs. 2].

4.6.5 Parameter

Auch wenn die bisherige Join Point-API nicht vorsieht, einzelne Parameterwerte untersuchen zu können, ergeben sich immense Vorteile, wenn die Möglichkeit besteht, bestimmte Parameter näher als nur auf ihre Typen hin analysieren zu können. Es ist beispielsweise denkbar, Parameter in einer netzwerkbasierten Softwarelösung gemäß ihrem Synchronisationsbedarf zu markieren. So könnte man Parameter, die in einer Funktion nicht verändert werden, sondern lediglich aktuell vom Server bezogen werden müssen, mit `[[in]]` markieren, und Parameter, bei denen auch das Ergebnis zurück an den Server übermittelt werden muss, mit `[[inout]]`. Mittels Aspekten ließe sich eine automatische Übertragung solcher Pa-

rameter realisieren. Eine entsprechende Funktionsdeklaration könnte beispielsweise so aussehen:

```
void handleShared(int a [[in]], Shared* b [[inout]]);
```

Auch für Parameter sieht der Standard die Verwendung vor der Deklaration oder alternativ nach dem Bezeichner vor.

4.6.6 Rückgabetypen

Bei Rückgabetypen Annotationen zuzulassen bietet die Möglichkeit, beispielsweise zusätzliche Überprüfungen für selbige vorzunehmen. So ist es etwa denkbar, zu prüfen, ob Pointerrückgabetypen `NULL` sind. Gegebenenfalls könnten entweder Fehlermeldungen oder Standardbehandlungen für solche Fälle vorgenommen werden und somit die Produktstabilität erhöht werden.

4.6.7 Statements

Selbst einzelne Statements mit Annotationen versehen zu können, kann nützlich sein, um neue Join Points zu identifizieren.

Ein denkbarer Einsatzzweck könnte zum Beispiel das Festhalten eines Snapshots vor und nach der Ausführung eines Statements sein, wodurch zusätzliche Möglichkeiten der Fehlersuche geschaffen würden.

4.6.8 Compound Statements

Durch solche Annotationen wird es bspw. möglich, die Parallelisierbarkeit bestimmter Codeblöcke aufzuzeigen und dementsprechend an diesen Stellen Nebenläufigkeit und Synchronisation in den Code einzuweben.

Eine Option ist beispielsweise die Annotierung von Schleifenrümpfen, wie in Beispiel 4.2.

```
1 for(int i=0; i<n; i++)
2 [[parallel]] {
3     //do something parallelized
4 }
```

Beispiel 4.2: Attribute für Schleifenparallelisierung.

5. Umsetzungskonzept

Für die Umsetzung von Annotationen auf der Basis von C++11-Attributen kommen verschiedene Konzepte in Betracht, welche im Folgenden im Hinblick Tauglichkeit und Umsetzbarkeit untersucht werden. Abschließend wird das letztlich eingesetzte Konzept näher erläutert.

Wie bereits in Abschnitt 3.6 „Attribute in C++“ dargelegt, können Attribute an den verschiedensten Stellen im Sourcecode verwendet werden. Somit muss ein Umsetzungskonzept ermöglichen, Attribute an all diesen Stellen zu identifizieren und zu verarbeiten. Dies hat zur Folge, dass Parameter stets im jeweiligen Kontext der Attributverwendung ausgewertet werden müssen und im Quelltext verfügbar sein sollen.

AspectC++kann für die Codeanalyse und -manipulation auf verschiedene Front Ends zurückgreifen. So kann sowohl PUMA als auch Clang eingesetzt werden, um den Code zu parsen, zu analysieren und zu transformieren. Da für PUMA immer größerer Wartungsaufwand entstand, um kompatibel zu neuen C++-Standards zu bleiben [9], und zugunsten besserer Fehlerausgaben wurde Clang als Standardfrontend gewählt. Da Clang somit das aktuell häufiger eingesetzt Frontend sein dürfte, wird für diese Erweiterung zunächst Clang betrachtet.

5.1 Auswertung mittels Clang

Es ist naheliegend, diese Auswertung gänzlich vom Compiler übernehmen zu lassen, da er an jeder Stelle des Quelltextes über präzise Informationen zum Kontext und zur Zugehörigkeit von Attributen verfügt. Wie bereits erwähnt (siehe Abschnitt 3.6), ist der Umgang mit nicht im Standard enthaltenen Attributen nicht weiter vorgegeben [7, Kapitel 7.6.1]. Daher kann ein Compiler Attribute, die ihm nicht bekannt sind, schlicht ignorieren.

Clang gibt für solche Attribute zwar Warnungen aus, ignoriert sie allerdings ansonsten vollständig und legt sie auch nicht im Syntaxbaum ab, sodass sie später nicht zur weiteren Verarbeitung verfügbar sind. Daher wäre diese Variante lediglich für bekannte (insbesondere nicht für benutzerdefinierte) Attribute geeignet und genügt somit nicht den Anforderungen.

5.2 Attributparser-Funktionen wrappen

Prinzipiell ist es denkbar, die Erkennung und Behandlung der Attribute durch Clang zu überschreiben, indem man diese Funktionen "wrapt", d.h. eigene Funktionen implementiert, welche vom Linker an Stelle der ursprünglich in der Bibliothek befindlichen Funktionen verwendet werden. Dieses Vorgehen ist jedoch nicht zwangsläufig immer möglich, da nur solche Funktionsaufrufe vom Linker verarbeitet werden, bei welchen dem Compiler nicht schon vorher die Adresse der aufgerufenen Funktion bekannt ist. Daher ist nicht unbedingt sichergestellt, dass alle nötigen Funktionsaufrufe umgeleitet werden können.

Des Weiteren ist diese Form der Ersetzung sehr eng an die jeweils verwendete Bibliothek gebunden, sodass bei Aktualisierungen selbiger oder Verwendung eines anderen Compilers umfangreiche Anpassungen nötig würden. Insbesondere auf Grund der gleichzeitigen Unterstützung von PUMA als Frontend ist die Flexibilität dieser Variante nicht ausreichend.

5.3 Ersetzungen als Kommentar

Ein vom Compiler unabhängiges Vorgehen ist nur durch Hinterlegen von Informationen im Quelltext oder dem selbst verwalteten Join Point Modell möglich. Da bei der ersten Verarbeitung der Quelltext lediglich geparst wird und noch kein Syntaxbaum von Clang zur Zuordnung verfügbar ist, kann die Zuordnung nur auf Grund des eigenen Parsers vollzogen werden, welcher allerdings keine genauen Informationen über Funktionsinhalte erhebt und somit die Verwendungsorte zu stark einschränken würde.

Alternativ könnten die Attributverwendungen zu Kommentaren umgeformt werden, wobei die komplette Verwendung des Attributs durch einen Kommentar (welcher an jeder Stelle des Quelltextes erlaubt ist) ersetzt wird. Dadurch können beim Compiler keine Warnungen oder Fehlermeldungen hervorgerufen werden. Diese Kommentare müssen später, wenn der Syntaxbaum von Clang zur Verfügung steht, wieder auffindbar sein. Um dies zu gewährleisten, sind die Einfügungen direkt bei dem attributierten Element vorzunehmen. Dazu muss während des Parsens bereits ausgewertet werden, ob das Attribut zum vorangegangenen oder zum nachfolgendem Element gehört (siehe 4.2.2 „Attributverwendungen“).

Zur exakten Auswertung der Parameter einer Attributverwendung werden genaue Informationen über Typen und Werte derselben benötigt. Diese können vom Compiler allerdings nur erlangt werden, wenn er sie als gültiges C++-Konstrukt einlesen und auswerten kann. Da dies bei Kommentaren nicht gegeben ist, kommt auch diese Variante nicht in Betracht.

5.4 Ersetzungen als Funktionsaufruf der Deklaration

Um die Parameter bereits während des Übersetzungsvorgangs auswerten zu können, müssen sie in ein gültiges, vom Compiler interpretierbares C++-Konstrukt umgewandelt werden. In der Struktur ähnelt eine Attributverwendung einem Funktionsaufruf, sodass eine Ersetzung durch einen Funktionsaufruf sinnvoll erscheint. Dazu müsste die Deklaration der Attribute zu der entsprechenden Funktionsdeklaration transformiert werden. Problematisch ist, dass Funktionen nicht an jeder Stelle im Code aufgerufen werden können (z. B. in einer Klassendeklaration).

Alternativ wäre denkbar, nicht Ersetzungen zu Funktionsaufrufen vorzunehmen, sondern zu Funktionsdeklarationen. Jedoch auch hier stellen die möglichen Orte ein Problem dar: Innerhalb einer anderen Funktion darf keine Funktion deklariert werden. Möchte man Attribute auch innerhalb von Funktionsrümpfen verwenden können, müssten solche Ersetzungsfunktionen außerhalb der umgebenden Funktion deklariert werden. Dabei ist leider nicht gewährleistet, dass die verwendeten Parameter, welche u.U. aus dem Kontext des Funktionsrumpfs stammen, auch bei der Deklaration außerhalb verfügbar sind. In diesem Fall können sie nicht ausgewertet werden, wodurch auch dieser Ansatz nicht umsetzbar ist.

Zudem ist bei Funktionsparametern nicht unbedingt gewährleistet, dass sie vom Compiler vollständig ausgewertet werden können, sodass dies zusätzlich implementiert werden müsste.

5.5 Ersetzung als Template-Struct

Structs sind in jedem Scope deklarierbar und können somit auch innerhalb einer Funktion deklariert werden. Um zu gewährleisten, dass die Structs exakt zugeordnet werden können, werden ihre genauen Bezeichner, wie zuvor im Abschnitt 5.3 beschrieben, als Kommentar an der entsprechenden Stelle der Attributverwendung hinterlegt. Diese Structs dienen als Container für weitere Structs, die die eigentlichen Attributverwendungen widerspiegeln. Selbige enthalten die bei der Verwendung übergebenen Parameter als Template-Argumente, die im weiteren Vorgehen vom Compiler interpretiert werden und als Member des Structs hinzugefügt werden können und somit auch für die Verwendung innerhalb von Aspekten verfügbar sind.

5.5.1 Vorzüge von Templateparametern

Die vergleichsweise komplex erscheinende Umsetzung von Attributparametern mit Hilfe von Templates bietet auch einige Vorteile, welche im Folgenden näher erläutert werden. Neben Integerwerten (Ganzzahlen) können Templateparameter auch Typen, Referenzen oder Pointer repräsentieren. Dadurch wird es möglich, Informationen zu verschiedensten Objekten oder sogar zu verwendende Funktionen auszutauschen. Insbesondere können Templates als Parameter wiederum Templateargumente erhalten, wodurch sogar rekursive Strukturen möglich werden.

Insgesamt bieten Templateparameter deutlich mehr Flexibilität für unsere Ansprüche als gewöhnliche Funktionsparameter. Gleichzeitig ist bei Templateargumenten sichergestellt, dass sie bereits zur Übersetzungszeit auswertbar sind und Attributverwendungen somit keine schwer zu überblickenden Nebenläufigkeiten oder Seiteneffekte hervorrufen können.

5.5.2 Sonderfall Stringliterale

Gemäß dem C++-Standard können Templates keine Stringliterale (Zeichenketten) als Parameter entgegennehmen [7, Kapitel 14.3.2], Attribute hingegen schon. Um auch solche Parameter verwenden zu können (siehe Abschnitt 4.1.2 „Statische Parameter“), müssen sie daher gesondert behandelt werden: Bereits beim ersten Parsen werden die Argumente auf Stringliterale hin untersucht. Wird ein solches gefunden, wird der String durch einen Kommentar ersetzt und an seiner Stelle die Adresse eines `extern char` übergeben, was für den Compiler auswertbar ist. Diese Ersetzung kann ähnlich wie die Kommentare zu den Container-Structs später aufgelöst werden.

Da dieses Umsetzungskonzept alle erforderten Eigenschaften aufweist, wird es in dieser Arbeit eingesetzt, um die Erweiterung zu realisieren.

6. Implementierung

Im Folgenden soll die Implementierung des im Abschnitt 5.5 „Ersetzung als Template-Struct“ erläuterten Umsetzungskonzepts näher beleuchtet werden. Insbesondere wird auf die notwendigen Code- und Modelltransformationen eingegangen.

6.1 Implementierung in AspectC++

In diesem Abschnitt wird grob veranschaulicht, wie AspectC++ den Quelltext verarbeitet und welche Teile von AspectC++ dabei interagieren.

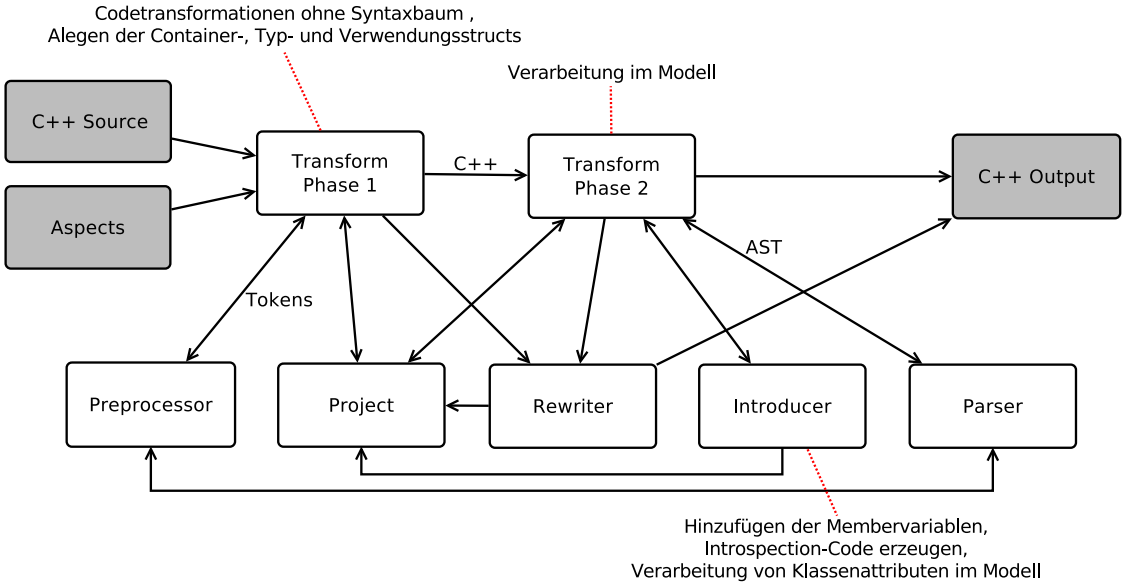

Abbildung 6.1.: Einfache Veranschaulichung der Quelltextverarbeitung [9, In Anlehnung an Abb. 3.1].

Die Abbildung 6.1 veranschaulicht in stark abstrahierter Form die Verarbeitung des ursprünglichen Quellcodes zum Ergebniscode mit eingewobenen Aspekten. Da beim ersten Parsen noch keine Informationen über die Typen und attributierten

Elemente verfügbar sind, werden in *Phase1* lediglich die in den noch folgenden Abschnitten 6.4.1, 6.4.2 sowie 6.4.3 (ohne 6.4.3.1) genannten Codetransformationen durchgeführt.

In *Phase2* kann auf die Informationen aus dem Syntaxbaum zurückgegriffen werden, sodass hier in *Clangintroducer* die Membervariablen, wie in Abschnitt 6.4.3.1 beschrieben, angelegt werden können. Sie werden benötigt, um Attributparameter innerhalb des Programms zur Verfügung zu stellen (Introspektion). Außerdem kann nun die Zuordnung von Attributen zu attributierten Elementen (siehe Abschnitt 6.4.4) in *ClangASTConsumer* vorgenommen werden.

6.2 Erkennung von Attributen im Quelltext

Da Attribute erst vor vergleichsweise kurzer Zeit in den Standard aufgenommen wurden und wie im Abschnitt 3.6 „Attribute in C++“ erläutert, zur Kommunikation mit dem Compiler dienen, werden ihre Kennzeichnungstokens (`[[...]]`) nicht gesondert vom Clang-Parser behandelt, sodass man jede einzelne eckige Klammer als Token erhält.

Um dennoch jede Attributverwendung zu erkennen, wird in Phase1 bei jedem `next_token`, dass abgeholt wird, geprüft, ob es sich dabei um den Anfang einer solchen Attributverwendung handelt. Ist dies der Fall, wird das Attribut komplett eingelesen und das erste Token nach der Attributspezifizierung zurückgegeben. Dies gilt auch für mehrere unmittelbar aufeinander folgende Attributspezifizierungen. Das eingelesene Attribut wird am Ende des Parsens (Phase1) entsprechend transformiert.

Neben den Attributverwendungen müssen auch die Attributdeklarationen geparst werden, um die Attribute korrekt im Modell ablegen zu können. Dazu existieren sowohl für Clang als auch für PUMA je ein neuer Tokentyp, der das Schlüsselwort `atttribute` identifiziert. Wie auch andere AspectC++-Tokens können diese in allen *.ah-Dateien oder unter Verwendung der Option `-k` überall erkannt werden.

6.3 Attributabbildungen im Modell

Wird eine Attributdeklaration identifiziert, wird ein Objekt des neuen Typs `ACM_Attribute` im Modell im aktuellen Scope abgelegt. Befindet sie sich nicht innerhalb eines *Klassen-* oder *Namespace*-Kontexts, wird eine entsprechende Fehlermeldung ausgegeben.

`ACM_Attribute` beerbt `ACM_Name` und jedes Objekt vom Typ `ACM_Any` verfügt über eine Liste von `ACM_Attribute`, in die Attributverwendungen später eingetragen werden können.

6.4 Codetransformationen

Wie bereits in Kapitel 5 „Umsetzungskonzept“ erörtert, müssen Attribute zunächst transformiert werden, um von AspectC++ in Phase2 ausgewertet werden zu können. Diese Transformationen betreffen sowohl die Attributdeklarationen als auch die Attributverwendungen. Die zugehörigen Codeveränderungen werden im Folgenden näher betrachtet.

Im Anhang findet sich ein vollständiges Listing, in dem die in diesem Abschnitt angeführten Beispiele umgesetzt sind.

6.4.1 Attributdeklarationen

Die Attributdeklarationen werden in Vorabdeklarationen (engl. predeclaration) von Structs umgewandelt. Sie dienen später ausschließlich zur Identifizierung des verwendeten Attributs. Da sie am Anfang der Übersetzungseinheit in einem eigenen Namensraum unterhalb des AC-Namensraum eingefügt werden, müssen Attribute nicht vor der Verwendung deklariert werden. Um Namenskonflikte auszuschließen, werden die Structs unter `AC::Attr::` deklariert. Dies wird in Beispiel 6.1 verdeutlicht. Die bei der Attributdeklaration angegebenen Parameter werden bei den Attributverwendungen aufgegriffen und daher später erläutert.

6.4.2 Attributverwendungen

Zu jedem Attributspezifizierungsblock (unmittelbar aufeinander folgende Verwendungen) wird ein Container-Struct erstellt, in dem die in Abschnitt 5.5 „Ersetzung als Template-Struct“ genannten Structs für jede einzelne Attributverwendung abgelegt werden. Darüber hinaus wird, wie ebenfalls bereits erklärt, ein Kommentar zur Identifizierung des Structs direkt bei dem attributierten Element im Quelltext eingefügt.

Um Warn- und Fehlermeldungen bei Verwendung eigener Attribute zu vermeiden, werden die ursprünglichen Attributverwendungen bereits beim Parsen aus dem Quellcode entfernt, es sei denn, sie gehören zum Clang- oder GNU-Namensraum oder haben keinen spezifizierten Namensraum (diese könnten im Standard vorgesehen sein).

```
1  namespace acc {
2    attribute myAttr(int i, void (*param)(int));
3  }
4
5  //---wird transformiert zu:---
6
7  namespace AC {
8    //sonstige Einfuegungen durch AspectC++
9    //...
10
11   namespace Attr {
12     namespace acc {
13       struct myAttr;
14     }
15   }
16 }
```

Beispiel 6.1: Transformation Attributdeklaration.

Möchte man Warnungen über unbekannte Attribute vermeiden, empfiehlt sich daher die Verwendung von Attributen innerhalb eines eigenen Namensraums. In Beispiel 6.2 wird diese Transformation gezeigt.

```
1  class [[deprecated]] [[acc::myAttr]] myClass {};
2
3  //---wird transformiert zu:---
4
5  struct __ac_attr__struct_0 { //Containerstruct
6      //Attributverwendung [[deprecated]]
7      //Attributverwendung [[acc::myAttr]]
8  }
9
10 class [[deprecated]] /*__ac_attr__struct_0*/myClass {};
```

Beispiel 6.2: Transformation Attributverwendung.

In diesem Beispiel wird das im Standardnamensraum vorhandene Attribut an der Stelle belassen, das aus dem eigenen Namensraum hingegen aus dem Quelltext entfernt. Beide Attribute werden innerhalb des Container-Structs aufgeführt, worauf im Folgenden eingegangen wird.

6.4.3 Templates und Datentypen

Da die Parameter essentielle Informationsträger sind, dürfen sie selbstverständlich nicht bei einer Transformation verloren gehen. Daher wird innerhalb eines Container-Structs je verwendetem Attributtyp ein Typ-Struct angelegt, das Templateparameter entsprechend der jeweiligen Attributdeklaration hat. Ein solches Struct wird in Beispiel 6.3 angelegt.

```
1 attribute myAttr(int x, int b);
2
3 //---wird beim jeweiligen Typ-Struct umgesetzt zu:---
4 template < int x, int b >
5 struct Type {};
```

Beispiel 6.3: Typ-struct in Abhängigkeit von Attributdeklaration.

Diese Typ-Structs werden bei jeder Verwendung eines Attributs beerbt und erhalten als Templateargumente die bei der Verwendung angegebenen Parameter. Somit sind sämtliche Parameter später im Syntaxbaum enthalten und können auch in Bezug auf ihre Typen untersucht und ausgewertet werden.

Zu der Attributverwendung `[[acc::myAttr(4, 5)]]` wird somit der Code aus Beispiel 6.4 beim Durchlaufen des ClangIntroduders generiert.

```
1 struct __ac_attr__struct_0 { //Containerstruct
2     template < int x, int b >
3     struct Type {};
4
5     struct AttrUsage : public Type<4, 5> {};
6 };
```

Beispiel 6.4: Typ-struct in Abhängigkeit von Attributdeklaration.

Um auch die transformierte Attributdeklaration (siehe Abschnitt 6.4.1) referenzieren und mehrfache Attributierungen eines Elements mit demselben Attribut erfassen zu können, wird in jedem Container-Struct folgendes Struct angelegt:

```
template <typename __ac_attr__ATTR, int I, int DUMMY = 0>
   struct __ac_attr__Attr { static const VALID = 0}; };
```

`__ac_attr__ATTR` kann das Struct übergeben werden, das aus der Attributdeklaration transformiert wird (vgl. Beispiel 6.1).

`I` zählt die Verwendungen eines Attributs für ein Element.

`VALID` dient dazu, Attribut-Structs, die von AspectC++ angelegt werden, von denen zu unterscheiden, die ggf. vom Entwickler (bspw. bei der Prüfung auf Existenz des Attributs) angelegt werden. Dazu wird der Wert standardmäßig auf `0` gesetzt und ggf. bei der Erzeugung eines Attributstructs durch AspectC++ auf `1` gesetzt.

`DUMMY` ist notwendig, da innerhalb einer Klasse keine vollständigen Templatespezialisierungen vorgenommen werden dürfen, sondern lediglich teilweise Templatespezialisierungen. Durch die Verwendung von `DUMMY` ist auch eine Spezialisierung mit dem entsprechenden Attribut und dem Zähler nur eine Teilspezialisierung, da DUMMY immer noch nicht festgelegt wird.

6.4.3.1 Abbildung der Datentypen

Um die Parameter auch für Aspekte zugänglich zu machen, müssen sie in den jeweiligen Verwendungs-Structs als Member hinterlegt werden.

Für die verschiedenen Typen von Templateargumenten muss hierbei unterschiedlich vorgegangen werden:

Typen

Eine Besonderheit von Templateparametern gegenüber anderen Parametern, wie etwa Funktionsparametern, ist, dass sie die Übergabe von Typen erlauben. Sie erfreuen sich insbesondere bei Containerklassen großer Beliebtheit, da so der Typ des Inhalts nicht schon bei der Entwicklung feststehen muss.

Um Typen innerhalb eines Structs Bezeichner geben zu können, kann auf `typedef` zurück gegriffen werden (wie in Beispiel 6.5 verdeutlicht). Dadurch wird dem übergebenen Typen ein weiterer Name zugewiesen.

```
1 template < typename X >  struct Type{};
2
3 class AttrUsage : public Type < Core >{
4     typedef Core X;
5 };
```

Beispiel 6.5: Typedef für Typen.

Sie können später sehr ähnlich wie Membervariablen angesprochen werden:

```
AttrUsage::X myVar;
```

Referenzen

Templates können als Argument auch sogenannte Nichttyp-Parameter erhalten, sofern diese zur Übersetzungszeit bereits auswertbar sind. Hierzu zählen auch Referenzen.

Möchte man sie dem jeweiligen Verwendungs-Struct als Member zuordnen, ist dies unter Verwendung des Merkmals `static` möglich, das aussagt, dass es diese Variable nur einmal in der gesamten Übersetzungseinheit für diese Klasse gibt. Um auch Referenzen übergeben zu können, die innerhalb eines Aspekts manipuliert werden können, erhalten die zugehörigen Structmember nicht zwangsläufig das Merkmal `const`, das eine Veränderung des referenzierten Objekts nach der Initialisierung verbieten würde.

Die entsprechende Transformation ist in Beispiel 6.6, die Initialisierung in Beispiel 6.7 zu finden.

```
1 template < typename X , X &b>  struct Type{};
2
3 class AttrUsage : public Type< Core , co >{
4     typedef Core X;
5     static Core &b;
6 };
```

Beispiel 6.6: Membervariablen für Referenzen.

```
1 Core co;
2 //...
3 Core  &AttrUsage::b = co;
```

Beispiel 6.7: Initialisierung von Referenzen.

Werte

Templates können auch bestimmte Werte, wie `integer`-Werte, die bereits zur Übersetzungszeit auswertbar sind, als Parameter erhalten. Um solche Werte innerhalb des Structs zu speichern, bedarf es Membervariablen, die nicht verändert werden können, wozu sie mit den Merkmalen `static` und `const` versehen werden. `Boolean`- und `integer`-Werte werden aufgelöst und das Ergebnis wird sowohl in der Signatur als auch bei der Initialisierung verwendet.

```
static const int val;
```

Um die Initialisierung dieser Werte nicht gesondert von der der Referenzen behandeln zu müssen, wird auch sie erst in der nächsten Namensraumumgebung vorgenommen.

Funktionszeiger

Eine weitere Form von Templateparametern sind Funktionszeiger. Auch diese werden dem Struct „zugeordnet“, indem ein der Attributdeklaration entsprechender Funktionszeiger als Member angelegt wird, der in der nächsten Namensraumumgebung initialisiert wird. Siehe dazu Beispiel 6.8.

```
1 template < void ( * param ) ( int )>  struct Type{};
2
3 struct AttrUsage : public Type< func > {
4   void ( * param ) ( int );
5 };
```

Beispiel 6.8: Funktionspointer für Funktionsparameter.

Templatetemplateparameter

Templates können als Parameter wiederum Templates erhalten. Um beispielsweise ein Template-Struct im Struct abzubilden, muss ein entsprechender Datentyp innerhalb des Structs angelegt werden.

Bei der Verwendung eines solchen Templatetemplateparameters müssen die Parameter des inneren Templates nicht benannt werden, sodass keine Bezeichner bei der Transformation zur Verfügung stehen. Daher werden diese bei der Umformung schlicht generiert. Dies wird in Beispiel 6.9 veranschaulicht.

```
1 template < template < typename , int > class A>  struct Type{};
2
3 struct AttrUsage : public Type< List >{
4     template <typename T0, int T1> class A : public List<T0, T1> {};
5 };
```

Beispiel 6.9: Klasse für Templatetemplateparameter.

6.4.3.2 Sonderfall Stringliterale

Wie bereits in Abschnitt 5.5.2 „Sonderfall Stringliterale“ genannt, sind Stringliterale nicht als Parameter vorgesehen.

Da sie jedoch als Parameter für Attribute möglich sind und sich auch in anderen Sprachen als durchaus sinnvoll erwiesen haben (siehe Kapitel 3 „Überblick über Annotationen“), müssen auch sie verarbeitet werden können.

Dazu wird dem entsprechenden Verwendungs-Struct nicht das Stringliteral selbst, sondern die Adresse eines `extern char` übergeben. Das eigentliche Stringliteral wird als Kommentar direkt dahinter eingefügt. Somit ist es möglich, den korrekten Typen für die Membervariable zu bestimmen und bei der Initialisierung auf das auskommentierte Stringliteral im Quelltext zurückzugreifen. Zur Veranschaulichung wird dies in Beispiel 6.10 gezeigt.

```
1 extern char ___ac_attr___;
2 //...
3 template < int a , const char * B>  struct Type{};
4
5 struct AttrUsage : public Type< &___ac_attr___/* "alo" "a" */>{
6     static const char * B;
7 };
```

Beispiel 6.10: Stringliterale.

Bei der Initialisierung wird das ursprüngliche, aus dem Quelltext eingelesene Literal angegeben:

```
const char * AttrUsage::B =  "alo" "a" ;
```

6.4.4 Verwendung und Zuordnung von Attributen

Da die Container-Structs lediglich im gleichen Kontext wie das attributierte Element, nicht jedoch an der exakten Stelle der Attributverwendung eingefügt werden können, muss eine genaue Zuordnung auf anderem Wege erfolgen. Hierzu werden die im Abschnitt 6.4.2 genannten Kommentare herangezogen, die den genauen Namen des Structs liefern. Da das Struct bereits in den Syntaxbaum aufgenommen wurde, kann aus diesem auch die für das spätere Matching (siehe Abschnitt 2.2.2 „Pointcut“) benötigte Signatur erzeugt werden und so das passende Attribut zu dem attributierten Element im Modell abgelegt werden.

6.5 Reflexion/Introspektion

AspectC++ bietet dem sich in Ausführung befindlichen Programm die Möglichkeit, Informationen über seine eigene Struktur zu beziehen. Dies können u. a. Klassennamen und Member von Klassen sein. Dieser Mechanismus ist bisher für Klassen und deren Member implementiert. Auch die Attribute und die hinterlegten Informationen (siehe „Templates und Datentypen“) sollen an dieser Stelle erreichbar sein. Dazu wird für die im Quelltext hinterlegten Kommentare mit den Structbezeichnern ein entsprechendes Struct in der Introspektion mit der Bezeichnung `Attribute` hinterlegt. Für den Fall, dass ein zweites Attributstruct zu einem Element existieren kann (bspw. möglich bei Funktionen oder Membervariablen) wird für das zweite Struct die Bezeichnung `Attribute2` festgelegt. Beim Zugriff ist zu beachten, dass Attribute unterhalb des Namensraums `AC::Attr` deklariert werden.

Das Beispiel 6.11 veranschaulicht einen solchen Zugriff auf Attributinformationen.

6.6 Realisierbarkeit weiterer Features

In der folgenden Tabelle sind verschiedene bisher noch nicht implementierte Funktionalitäten aufgeführt. Dazu folgt eine Abschätzung, ob die jeweilige Erweiterung konzeptionell möglich oder mit dem hier verfolgen Ansatz nicht realisierbar ist.

```
1  #include <iostream>
2
3  using namespace std;
4
5  namespace acc {
6      attribute foo(int a);
7      attribute bar(const char* b);
8  }
9
10 struct [[acc::foo(5)]] myStruct {
11     [[acc::bar("attr1")]] int foo [[acc::bar("attr2")]];
12
13     static void attrOutput() {
14         cout << "StructAttr: " <<
15           AC::TypeInfo<myStruct>::Attribute<AC::Attr::acc::foo,
16           0>::signature() << " value: " << AC::TypeInfo<myStruct>
17           ::Attribute<AC::Attr::acc::foo, 0>::a << endl;
18         cout << "StructAttr: " << AC::TypeInfo<myStruct>::Member<0>
19           ::Attribute<AC::Attr::acc::bar, 0>::signature() <<" value: "
20           << AC::TypeInfo<myStruct>::Member<0>::Attribute
21           <AC::Attr::acc::bar, 0>::b << endl;
22         cout << "StructAttr: " <<
23           AC::TypeInfo<myStruct>::Member<0>::Attribute2<
24           AC::Attr::acc::bar, 0>::signature() << " value: " <<
25           AC::TypeInfo<myStruct>::Member<0>::Attribute2<
26           AC::Attr::acc::bar, 0>::b << endl;
27     }
28 };
29
30 int main() {
31     myStruct::attrOutput();
32 }
```

Beispiel 6.11: Zugriff auf Attributeigenschaften.

Feature	umsetzbar	nicht realisierbar	Bemerkung
Attribute an Statements	X		Erweiterung der Join Point API notwendig
Attribute an Compouncstatements	X		Erweiterung der Join Point API notwendig
Attribute an Rückgabetypen	X		Bereits implementiert, Anpassungen beim Matching notwendig
Attribute an Parametern	X		Erweiterung der Join Point API und Anpassungen beim Matching notwendig
Matching auf Basis der Typanalyse, nicht rein string-basiert	X		Überarbeitung des Matchings von Templateparametern notwendig
Matchexpressions mit direkt aufeinander folgenden Attributen	(X)	(X)	Logik müsste innerhalb einer Matchexpression greifen
Variadic Templates / beliebig lange Argumentlisten	X		Codegenerierung müsste erweitert werden
Gleitkommazahlen als Parameter		(X)	Eventuell Sonderbehandlung denkbar?
PUMA als Frontend	X		Anpassungen im Introducer und beim Erstellen des AST notwendig
Attribute über `thisJoinPoint`-Objekt verfügbar machen	X		Erweiterung des `tjp`-Structs notwendig

7. Evaluation

In diesem Kapitel wird untersucht, welche Auswirkungen die in dieser Arbeit implementierte Erweiterung auf damit entwickelte Softwareprodukte hat.

Dazu wird die tatsächliche Einsetzbarkeit der Erweiterung anhand einer aspektorientierten Umsetzung für generische Fehlererkennung und -korrektur exemplarisch betrachtet. Außerdem wird der generierte Code auf seine Performance in Hinsicht auf Speicherbedarf und Ausführungsgeschwindigkeit untersucht.

7.1 GOP (Generischer Objektschutz)

Im Folgenden wird der Einsatz von Attributen am Beispiel des generischen Objektschutzes (engl. *Generic Object Protection (GOP)*) veranschaulicht. Dazu wird zunächst die Grundidee dieser Form der Fehlerkorrektur dargelegt und untersucht, inwiefern sie durch Attribute bereichert werden kann.

7.1.1 Grundlagen generischer Objektschutz

Fehler im so genanntem DRAM (Hauptspeicher der meisten PCs) sind eine häufig auftretende Form von Hardwarefehlern in Computersystemen. Bei einem solchen Speicherfehler wird der Speicher eines oder mehrerer Bits anders ausgelesen, als zuletzt hineingeschrieben wurde. Sie können durch elektrische oder magnetische Interferenzen oder auch durch Übertragungsfehler hervorgerufen werden. Um Fehler erkennen und beheben zu können, werden häufig fehlerkorrigierende Codes (engl. *Error Correcting Code (ECC)*) eingesetzt, die Fehler von einzelnen oder sogar mehreren Bits bereinigen können (vgl. [16]).

Solche Fehlerkorrekturen fest in die Hardware zu integrieren, bringt auf der einen Seite einen höheren Energieverbrauch und Einbußen in der Performanz mit sich, auf der anderen Seite jedoch vor allem deutlich höhere Produktionskosten, weshalb sie meist nur in Serversystemen zum Einsatz kommen [2].

Um auch in günstigen Systemen Speicherfehler in wichtigen Teilen eines Betriebssystems identifizieren zu können, wurde mit Hilfe von AspectC++ ein GOP-Mechanismus ausgearbeitet, der bestimmte Klassen in objektorientierten Projekten gegen solche Fehler absichern und sie ggf. korrigieren kann (siehe [2]).

Die betroffenen Klassen, in denen Fehler verherende Auswirkungen hätten, werden als *kritische Klassen* bezeichnet. Da die Prüfung auf Fehler vor und nach jeder einzelnen Instruktion einen deutlich zu großen Overhead provozieren würde, werden Abschnitte für die Prüfung festgelegt: Die Überprüfungen werden vor dem Betreten einer kritischen Klasse vorgenommen, und nach der Ausführung werden die zur nächsten Überprüfung notwendigen Informationen ermittelt (vgl. [2, Abschnitt 3.1]). Dazu wird Redundanz in den Daten erzeugt, d.h. es werden Informationen hinterlegt, die ursprünglich auch schon in den Daten enthalten waren (vgl. [3, Kapitel 6]). Meist werden dazu zusätzliche Prüfbits ermittelt und bei den Daten abgespeichert. Anhand der Prüfbits kann später festgestellt werden, ob alle Bits den richtigen Wert haben. Im Fehlerfall kann ermittelt werden, welche Position betroffen ist, und das entsprechende Bit korrigiert werden.

7.1.2 Vorzüge von Attributen für GOP

Bisher werden Klassen, die einer solchen speziellen Überprüfung auf Speicherfehler bedürfen, einzeln aufgelistet. In der aktuellen Implementierung von GOP dienen zwei Pointcuts zur Identifizierung solcher Klassen:

```
pointcut criticalClassess = derived("Classname");
```

Dieser Pointcut erfasst die Klasse "Classname", sowie alle von ihr erbenden Klassen.

```
pointcut standAloneCriticalClasses("Foo || Bar");
```

Hier werden die Klassen `Foo` und `Bar` erfasst. Ihre erbenden Klassen sind damit nicht für die Fehlerkorrektur vorgesehen!

Der Nachteil dieses Vorgehens ist, dass die Klassen immer explizit genannt werden müssen. Entweder muss eine explizit genannte Klasse (`Classname`) beerbt werden oder der Pointcut `standAloneCriticalClasses` jedes Mal erweitert werden.

Dadurch ist es nicht möglich, den Mechanismus ohne weitere Anpassungen für ein bestehendes Projekt einzusetzen. Mit Hilfe von Attributen ist es möglich, alle relevanten Informationen im eigenen Quelltext zu hinterlegen und den Mechanismus an sich unangetastet zu lassen.

Die Erweiterung des GOP-Codes aus Beispiel 7.1 ermöglicht nun die Verwendung dieser Attribute, um an beliebigen Stellen im Quelltext Klassen als *kritisch* zu kennzeichnen:

```
class [[GOP::CriticalClass]] XYZ { /*...*/ };

class [[GOP::StandAlone]] Foo { /*...*/ };
```

```
1 namespace GOP {
2   attribute CriticalClass();
3   attribute StandAlone();
4 }
5 //...
6   pointcut criticalClasses() = derived("[[GOP::Critical]]");
7 //...
8   pointcut standAloneCriticalClasses() = "[[GOP::StandAlone]]";
```

Beispiel 7.1: Erweiterung des GOP-Mechanismus um Attribute.

7.1.3 Ausschluss von einzelnen Membervariablen mittels Attributen

Neben der vereinfachten Verwendung des Mechanismus durch Attribute wird mit ihrer Hilfe auch möglich, bestimmte Membervariablen kritischer Klassen zu kennzeichnen, um sie explizit von der Fehlerprüfung auszuschließen. Dies ist in der bisherigen Implementierung nicht möglich.

Im Anhang C befinden sich die am Mechanismus sowie am Beispielcode vorgenommenen Änderungen. In GOP wird zur Identifizierung auszuschließender Variablen ein weiteres Attribut benötigt. Dementsprechend wird das Attribut `[[GOP::uncriticalMember]]` deklariert. Die mit Checksummen zu versehenden Memebervariablen werden zur Laufzeit anhand der Introspektionen ermittelt. Bisher werden dazu Eigenschaften der Variablen wie z. B. *protection*, *static* oder auch *const* ausgewertet. Um die Prüfung für einzelne Variablen zu deaktivieren, muss an dieser Stelle auch die Existenz des entsprechenden Attributs überprüft werden. Dazu wird die in Abschnitt 6.4.3 „Templates und Datentypen“ vorgestellte Variable `VALID` geprüft. Ist der Wert dieser Variable `1`, so wird keine Checksumme für die attributierte Variable gebildet und somit auch keine Fehlerprüfung vorgenommen.

Im Beispielcode sind einzelne Variablen, die verfälscht werden, von der Fehlerprüfung ausgeschlossen worden, indem sie mit dem Attribut `[[GOP::uncriticalMember]]` versehen wurden. Anhand der Ausgaben 7.2 (ohne unkritische Variablen) und 7.3 (mit unkritischen Variablen) wird deutlich, dass verursachte Fehler in den entsprechend gekennzeichneten Membervariablen nicht mehr erkannt und behoben werden (siehe Variable `a`).

Beispiel 7.2: Ausgabe ohne unkritische Membervariablen.

```
1 errors_corrected: 1
2 a: 4711 b: 12
3 errors_corrected: 2
4 errors_corrected: 3
5 errors_corrected: 4
```

Beispiel 7.3: Ausgabe mit unkritischen Membervariablen.

```
1 errors_corrected: 1
2 a: 4199 b: 12
3 errors_corrected: 2
4 errors_corrected: 3
```

7.2 Performance des generierten Codes

Neben der Handhabbarkeit der Erweiterung ist die Performanz ein wichtiges Merkmal. Daher wird in diesem Abschnitt näher untersucht, wie performant der von AspectC++ mit der Erweiterung erzeugte Code ist. Bei der Entwicklung von AspectC++ wird darauf geachtet, dass der ausgegebene Code mindestens kompatibel zum GNU G++-Compiler ist [1]. Daher werden in diesem Abschnitt Ausgaben dieses Compilers betrachtet. Die Attribute wirken sich auf zwei wesentliche Bereiche von AspectC++ aus: Zum einen wird das Matching verändert, da Attribute mit einbezogen werden, zum anderen wird der Introspektions-Code, der zur Laufzeit Informationen über die Struktur des Programms liefert, um Attributinformationen erweitert. Performanceauswirkungen in diesen beiden Bereichen werden in den folgenden zwei Abschnitten betrachtet.

7.2.1 Auswirkungen des Matchings

Um einordnen zu können, inwieweit sich die Performance des erzeugten Codes durch den Einsatz von Attributen zum Matching anstelle bisheriger Konstrukte verändert, werden zwei Codebeispiele mit derselben Funktionalität gewoben, wobei bei einem Attribute beim Matching zum Einsatz kommen und bei dem anderen keine Attribute verwendet werden.

Bei der Untersuchung, welche Auswirkungen die Veränderungen von AspectC++ haben, wird zunächst betrachtet, ob bei der Verwendung von Code ohne Attribute Unterschiede zwischen der Ausgabe der AspectC++-Version ohne die Erweiterung und der mit der Erweiterung auftreten, was nicht der Fall sein sollte, da funktional

derselbe Code gewoben werden sollte. Da die ursprüngliche Version von AspectC++ noch keine Attribute unterstützte, muss der Code für diese Version gänzlich ohne Attribute konzipiert sein. Er ist in Beispiel 7.4 zu finden.

Auch der Code, der mit Attributen gewoben wird, sollte die gleiche Funktionalität bieten und sich somit nicht wesentlich von dem anderen unterscheiden. Der betrachtete Quelltext wird in Beispiel 7.5 gezeigt.

```
1  namespace myNamespace {
2      class  myClass {
3          public:
4          int foo(const int a, const char b[], int c) {
5              return c;
6          }
7          bool aTest;
8      };
9  }
10 using namespace myNamespace;
11
12 int main() {
13     myClass mC;
14     mC.foo(5, "as", 0);
15     return 0;
16 }
17
18 #include <iostream>
19 aspect modifierImplicitWithinNSConstr {
20     advice execution("myNamespace") : before () {
21         std::cout << "Contructor␣implicitly␣called␣within␣
               [[acc::NamespaceAttr(8)]]:␣" << tjp->signature() <<
               std::endl;
22     }
23 };
```

Beispiel 7.4: Matching ohne Attribute.

Erwartungsgemäß erzeugt der G++ bei Verwendung desselben Ausgangscodes für AspectC++ ohne die Erweiterung um Attribute exakt denselben Binärcode wie für AspectC++ mit der Erweiterung. Auch bei Verwendung des attributierten Ausgangcodes, mit attributbasiertem Matching für die Pointcuts, wird derselbe Binärcode erzeugt. Dies gilt unabhängig von den getesteten Optimierungsstufen (`-O0`, `-O3`, `-Os`), sofern bei beiden Varianten dieselben Optimierungen vorgenommen werden. Die Identifizierung von JoinPoints anhand von Attributen anstelle von Bezeichnern wirkt sich demzufolge nicht auf den Ergebniscode dieses Beispiels aus, sodass der erzeugte Code ebenso effizient ist wie bisheriger Ergebniscode.

```
1  namespace [[acc::NamespaceAttr(8)]] myNamespace {
2      class  myClass {
3          public:
4          int foo(const int a, const char b[], int c) {
5              return c;
6          }
7          bool aTest;
8      };
9  }
10 using namespace myNamespace;
11
12 int main() {
13     myClass mC;
14     mC.foo(5, "as", 0);
15     return 0;
16 }
17
18 #include <iostream>
19 namespace acc{
20     attribute NamespaceAttr(int a);
21 }
22 aspect modifierImplicitWithinNSConstr {
23     advice execution("[[acc::NamespaceAttr(8)]]") : before () {
24         std::cout << "Contructor implicitly called within
              [[acc::NamespaceAttr(8)]]: " << tjp->signature() <<
              std::endl;
25     }
26 };
```

Beispiel 7.5: Matching mit Attribut.

7.2.2 Auswirkungen von Attributzugriffen

Neben den Auswirkungen des Matchings auf den Ergebniscode ist auch interessant, welchen Aufwand es erfordert, auf Eigenschaften von Attributen zuzugreifen. Je nach Umsetzung kann es große Unterschiede in Bezug auf die Laufzeit geben. Für jeden Zugriff auf Attributeigenschaften muss eindeutig identifiziert werden, um welches Attribut es sich handelt und welchen Wert die entsprechende Eigenschaft hat. Würde dies zur Laufzeit geschehen, würde insbesondere die Identifizierung des Attributs einen nicht unerheblichen Aufwand bedeuten.

In der Umsetzung dieser Erweiterung wird die Identifizierung des jeweiligen Attributs mit Templates realisiert und somit bereits beim Übersetzen erledigt. Dementsprechend sollte der von G++ ausgegebene Binärcode zum Ausgabecode von AspectC++ performant sein und ähnlich wie Zugriffe auf Konstanten sein.

In den Beispielen 7.6 und 7.7 sind die verwendeten Beispielquelltexte zu finden. Bei beiden werden `integer`-Werte zugeordnet (Zeile 17), zwei `const char*`-Zeiger gespeichert (Zeilen 18+19) sowie eine Referenz zugewiesen. Ausgaben von Werten erfolgen in den Zeilen 22 bis 25. Im ersten Fall wurde schlicht der konstante Wert `8` gesetzt, ein konstanter String als Platzhalter referenziert und die Referenz direkt auf das angelegte Objekt gesetzt. Im anderen Fall werden die Werte, die dem Attribut als Parameter übergeben wurden (ebenfalls `8` sowie eine Referenz auf das Objekt `d`), zugewiesen und die Signatur des Attributs als String verwendet. Außerdem wird zur Betrachtung der Stringersetzungen bei Parametern (siehe Abschnitt 5.5.2 „Sonderfall Stringliterale“) eine Zeichenkette als Parameter übergeben und auch diese zugewiesen und ausgegeben.

Objectdumps zu beiden Varianten sind im Anhang B zu finden. Ohne Optimierungen unterscheiden sich die Objectdumps der beiden Varianten. Mit Ausnahme einer Assemblerinstruktion sind die Instruktionen zwar identisch, jedoch wird bei der Variante mit Attribut ein zusätzlicher Block für die Attributeigenschaften aufgeführt und die Adressen unterscheiden sich. Unter Verwendung der Optimierungsoption (`-O3`) wird deutlich, wie ähnlich die beiden Varianten tatsächlich sind: Der Ausgabecode ist absolut identisch. Der Zugriff auf Werte, die als Parameter übergeben werden, sowie auf die Attributsignatur kann auf Grund der Verwendung von Templates gänzlich vom Compiler aufgelöst werden, sodass letztlich direkt die Konstanten bzw. hinterlegten Werte verwendet werden.

Der Zugriff auf Attributeigenschaften bzw. Attributparameter entspricht im Ergebniscode bzgl. des Aufwands zur Ausführungszeit einem direkten Zugriff auf Konstanten oder Werte bzw. im Falle der Stringersetzung dem einer Zeigerzuweisung und ist somit sehr performant.

```
1  #include <iostream>
2
3  namespace acc {
4      //  attribute classAttr(int a, const char* b, RefObj& c);
5  }
6
7  const char *bTest = "hallo";
8
9  struct RefObj{const char* test = "test";} d;
10
11 struct myClass {int b;};
12
13 int main() {
14     myClass mC;
15     mC.b = 0;
16
17     int test = 8;
18     const char *testStr = "[[acc::classAttr(8, \"hallo\" )]]";
19     const char *b = bTest;
20     RefObj &refTest = d;
21
22     std::cout << test << std::endl;
23     std::cout << b << std::endl;
24     std::cout << testStr << std::endl;
25     std::cout << refTest.test << std::endl;
26
27     return 0;
28 }
```

Beispiel 7.6: Ausgangscode ohne Attribute und Zugriffe.

```
1  #include <iostream>
2
3  namespace acc {
4      attribute classAttr(int a, const char* b, RefObj& c);
5  }
6
7  const char *bTest = "hallo";
8
9  struct RefObj{const char* test = "test";} d;
10
11 struct [[acc::classAttr(8, "hallo", d)]] myClass {int b;};
12
13 int main() {
14     myClass mC;
15     mC.b = 0;
16
17     int test =
           AC::TypeInfo<myClass>::Attribute<AC::Attr::acc::classAttr,
           0>::a;
18     const char *testStr =
           AC::TypeInfo<myClass>::Attribute<AC::Attr::acc::classAttr,
           0>::signature();
19     const char *b =
           AC::TypeInfo<myClass>::Attribute<AC::Attr::acc::classAttr,
           0>::b;
20     RefObj &refTest =
           AC::TypeInfo<myClass>::Attribute<AC::Attr::acc::classAttr,
           0>::c;
21
22     std::cout << test << std::endl;
23     std::cout << b << std::endl;
24     std::cout << testStr << std::endl;
25     std::cout << refTest.test << std::endl;
26
27     return 0;
28 }
```

Beispiel 7.7: Ausgangscode mit Attribut und Zugriff.

8. Zusammenfassung und Ausblick

Dieses Kapitel bietet eine Zusammenfassung sowie einen Ausblick auf mögliche zukünftige Forschungen mit Bezug zu dieser Arbeit.

8.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Konzept zur Realisierung von Anntotationen auf der Grundlage von Attributen gemäß dem C++11-Standard erarbeitet. Mit diesen Annotationen sind zwei wesentliche Bereiche von AspectC++ verbunden, die kurz näher genannt werden:

Matching
: Mit Hilfe von Attributen können Join Points eindeutig identifiziert und zugeordnet werden. Daraus ergibt sich die Möglichkeit, Sprachelementen Informationen hinzuzufügen, die insbesondere für die gemeinsame Erfassung ähnlicher Elemente geeignet sind. Die übergebenen Parameter können direkt mit einbezogen werden und ermöglichen somit eine feinere Zuordnung.

Reflexion/Introspektion
: In AspectC++ kann das Programm während der Ausführung Informationen über seine Struktur abfragen. Derzeit ist dieser Mechanismus für genau jene Attribute verfügbar, die an Klassen oder deren Membern verwendet werden. Somit können sowohl Parameterwerte als auch die Signatur eines Attributs zur Laufzeit abgefragt werden.

Beim Entwurf des Annotationskonzepts und dessen Umsetzung wurde besondere Rücksicht darauf genommen, die Erweiterung für Projekte kompatibel zu gestalten, die nicht mit dem C++11-Standard oder neuer arbeiten. Die Bedeutung und Auswirkung von Attributen unter Verwendung eines neueren Standards wird selbstverständlich nicht verfälscht.

Die Erweiterung bietet neue Möglichkeiten zur einheitlicheren Gestaltung von PointCut-Ausdrücken und zur Deklaration sowie Verwendung von zusätzlichen Attributen durch den Entwickler, was in C++ in dieser Form bisher nicht möglich ist.

Ein großer Teil der von dieser Erweiterung erhofften zusätzlichen Funktionalität ist somit bereits gegeben und umgesetzt. Dennoch besteht an vielen Stellen Erweiterungs- und Verbesserungsbedarf, worauf im folgenden Abschnitt eingegangen wird.

8.2 Ausblick

Die in dieser Arbeit umgesetzte Erweiterung von AspectC++ wurde mit Blick auf weiteren möglichen Ausbau konzeptioniert. Die bisherige Implementierung bietet die Grundlage für die Verwendung von Attributen in AspectC++, beschränkt sich in einigen Bereichen jedoch auf beispielhafte Typen von Sprachelementen. Hier werden weitere Ansätze für einen umfangreicheren Einsatz dargelegt. Dazu werden Erweiterungen in den Bereichen Reflexion, attributierbare Elemente, Matchexpressions, Parametertypen sowie Standardwerten für Parameter betrachtet.

Reflexion/Introspektion

Die zuvor genannte Reflexion ist bisher ausschließlich auf Klassen beschränkt. Für diesen Elementtyp bestand bereits zuvor eine Implementierung, sodass Attribute einfach hinzugefügt werden konnten. Beispielsweise für Funktionen und Variablen außerhalb von Klassen steht dieser Mechanismus noch nicht zur Verfügung, sodass zu erwägen ist, ihn auch hierfür bereitzustellen und somit auch auf deren Attribute während der Ausführung zugreifen zu können.

Attributierbare Elemente

Die bisherige Join Point-API sieht unter anderem Namensräume, Klassen und Funktionen als Join Points vor. Mit Hilfe von Attributen könnten auch bedeutend feinere Strukturen identifiziert werden, die jedoch noch nicht im Modell abgebildet werden können. Mögliche Einsatzszenarien für feinere Join Points wurden im Abschnitt 4.6 „Annotierbare Sprachelemente“ aufgeführt. Bestrebenswert ist insbesondere die Möglichkeit, Compoundstatements, Schleifen und Funktionsparameter attributieren zu können.

Matchexpressions

Bei der Identifizierung von Attributen mit Parametern wird auf bereits bestehende Mechanismen zur Untersuchung von Templateparametern zurückgegriffen. Dabei wird verglichen, ob es sich um dieselben Parameter handelt. Insbesondere bei Referenz- und Typparametern ist es denkbar, auch die Vererbungshierarchie mit einzubeziehen. Darüber hinaus könnte es sehr nützlich sein, Werte nicht nur auf Gleichheit bzw. Ungleichheit zu prüfen, sondern im

Falle von Ganzzahlen bspw. auch „kleiner“- oder „größer“-Vergleiche durchführen zu können.

Parametertypen

Durch die Umsetzung von Parametern mit Hilfe von Templates sind mögliche Parametertypen recht starr festgelegt. Zwar werden bei bisherigen Attributen meist Ganzzahlen oder Stringliterale übergeben, jedoch sind insbesondere für benutzerdefinierte Attribute weitere Typen wünschenswert. Beispielsweise kann es von Vorteil sein, Gleitkommazahlen übergeben zu können. Solche sind jedoch nicht als Templateparameter vorgesehen. Wie am Beispiel der Stringliterale deutlich wird, ist es möglich, mit AspectC++ weitere Typen für Parameter zu ermöglichen.

Standardwerte für Parameter

Viele Attribute erlauben optional, auf die Angabe von Parametern zu verzichten. Da das Konzept dieser Arbeit keine Überladung von Attributen vorsieht, kann die Auslassung von Attributen nicht damit simuliert werden. Eine andere Variante stellen Standardwerte (auch Defaultargumente genannt) dar. Sie ermöglichen, bestimmte Werte anzugeben, die ein Parameter annimmt, wenn kein anderer Wert explizit genannt wird. Für Templates sind solche Standardwerte nach dem C++-Standard bereits möglich, sodass für die Umsetzung der Reflexion kaum Änderungen vorzunehmen wären. Für die Umsetzung des Matchings müssten diese Standardwerte jedoch korrekt berücksichtigt werden.

Zusammengefasst bietet die in dieser Arbeit entwickelte Erweiterung einen vielversprechenden Ansatz zur Erweiterung von AspectC++ um Annotationen. Das erarbeitete Konzept scheint auch für weitere Anpassungen und Erweiterungen eine geeignete Grundlage zu bieten.

Literatur

[1] *AspectC++ Frequently Asked Questions*. zuletzt abgerufen: 2016-04-26. URL: http://www.aspectc.org/FAQ.php.

[2] C. Borchert, H. Schirmeier und O. Spinczyk. "Generic Soft-Error Detection and Correction for Concurrent Data Structures". In: *IEEE Transactions on Dependable and Secure Computing* PP.99 (). To appear. ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2427832.

[3] I. Djordjevic. *Quantum Information Processing and Quantum Error Correction: An Engineering Approach*. Academic Press. Elsevier/Academic Press, 2012. ISBN: 9780123854919.

[4] R. Dornberger, A. Reber und A. Hochuli. "Aspektorientierte Programmierung". In: (März 2005).

[5] J. Goll. *Methoden und Architekturen der Softwaretechnik*. Vieweg+Teubner Verlag, 2012. ISBN: 9783834881649.

[6] J. Gosling u. a. *The Java ® Language Specification*. Englisch. Standard JSR-337. Oracle America, Inc., 13. Feb. 2015. Kap. 9.6. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[7] ISO/IEC. *Programming Languages – C++*. Englisch. ISO N3690. Vorläufige Version! Geneva, Switzerland: International Organization for Standardization, 15. Mai 2013. Kap. 7.6.

[8] ISO/IEC. *Technical Report on C++ Performance*. Englisch. ISO WG21 N1487=03-0070. zuletzt abgerufen: 2016-04-26. Geneva, Switzerland: International Organization for Standardization, 11. Aug. 2003. Kap. 2.3.5. URL: http://www.open-std.org/jtc1/sc22/open/n3646.pdf.

[9] B. Kramer. "AspectClang: Moving AspectC++'s Weaver to the Clang C++ Front End". Bachelor's Thesis. TU Dortmund, 2013.

[10] B. Lahres und G. Raýman. *Praxisbuch Objektorientierung. Von den Grundlagen zur Umsetzung*. Rheinwerk Verlag (vorm. Galileo Press), 2006. Kap. 9. ISBN: 3-89842-624-6.

[11] *C#Language Specification*. Englisch. Standard. Microsoft Corporation, 2012. Kap. 17.1.3. URL: https://www.microsoft.com/en-us/download/confirmation.aspx?id=7029.

[12] D. Lohmann u. a. "A Quantitative Analysis of Aspects in the eCos Kernel". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, S. 191–204. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217954. URL: http://doi.acm.org/10.1145/1217935.1217954.

[13] J. Maurer und M. Wong. *Towards support for attributes in C++ (Revision 6)*. n2761. zuletzt abgerufen: 2016-01-03. 2008. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf.

[14] S. Radetzky. "Metadata Annotations for Explicit Joinpoints in AspectC++". Diploma's Thesis. TU Dortmund, 2011.

[15] G. Roden. *Auf der Fährte von C#: Einführung und Referenz*. Xpert.press. Springer Berlin Heidelberg, 2008. ISBN: 9783540278894.

[16] B. Schroeder, E. Pinheiro und W.-D. Weber. "DRAM Errors in the Wild: A Large-Scale Field Study". In: *SIGMETRICS*. 2009.

[17] B. Stroustrup. *The C++ Programming Language*. Pearson Education, 2013. ISBN: 9780133522853.

[18] the AspectJ Team. *The AspectJTM 5 Development Kit Developer's Notebook*. Dokumentation. zuletzt abgerufen 2016-04-11, 12:00. 2005. Kap. 2. URL: https://eclipse.org/aspectj/doc/next/adk15notebook/index.html (besucht am 08. 04. 2016).

[19] C. Ullenboom. *Java ist auch eine Insel: das umfassende Handbuch ; [Programmieren mit der Java Platform, Standard Edition 6 ; Java von A bis Z: Einführung, Praxis, Referenz ; von Klassen und Objekten zu Datenstrukturen und Algorithmen]*. Galileo computing. Galileo Press, 2010. ISBN: 978-3-8362-1506-0.

[20] C. Ullenboom. *Java ist auch eine Insel: das umfassende Handbuch ; [Programmieren mit der Java Platform, Standard Edition 7 ; Java von A bis Z: Einführung, Praxis, Referenz ; von Klassen und Objekten zu Datenstrukturen und Algorithmen ; aktuell zu Java 7]*. Galileo computing. Galileo Press, 2012. ISBN: 9783836218023.

[21] M. Urban und O. Spinczyk. *AspectC++ Language Reference*. Reference. 2012.

Abbildungsverzeichnis

Listingverzeichnis

A. Vollständinge Transformation von Attributverwendungen

Die Transformation von Attributverwendungen mit Introspektion wird im folgenden Beispiel veranschaulicht:

```
1  //Attributdeklarationen - meist in *.ah-Datei
2  namespace acc {
3      attribute aha(typename X, template<typename, int> class A);
4      attribute myAttr(int i, void (*param)(int));
5      attribute hi();
6  }
7
8  //...
9
10 void func(int ab) {}
11 template<typename A, int B> struct List{ };
12
13 //...
14
15 class [[acc::aha(int, List)]] myClass {
16     public:
17     int myFunc [[acc::myAttr(4, func), acc::hi]] (const int a, const
           char b[], int c) {
18         return c;
19     }
20 };
21
22 //---wird transformiert zu:---
23
24 //...
25 extern char __ac_attr__;
26 namespace AC {
27     //...
28
29     //Attribut-Deklarationen
30     namespace Attr {
31         namespace acc{
32             struct aha;
33         }
```

```
34          namespace acc{
35              struct myAttr;
36          }
37          namespace acc{
38              struct hi;
39          }
40      }
41  }
42
43  void func(int ab) {}
44  template<typename A, int B> struct List{ };
45
46  struct ___ac_attr___struct_0 { //Containerstruct fuer Klassenattribute
47      template <typename ___ac_attr___ATTR, int I, int DUMMY = 0> struct
            ___ac_attr___Attr { static const int VALID = 0; };
48
49      template < typename X , template < typename , int > class A>
            struct ___ac_attr___type_0{};
50      template <int DUMMY> struct ___ac_attr___Attr<AC::Attr::acc::aha,
            0, DUMMY> :
51      public ___ac_attr___type_0< int , List >{
52          static const int VALID = 1;
53          typedef int X;
54          template <typename T0, int T1> class A : public List<T0, T1>
                {};
55          static const char *signature () { return
                "[[acc::aha(int,List)]]";}
56      };
57
58  };
59
60  class  /*//_**___ac_attr___struct_0**_//*/myClass {
61      public:
62      //...
63      struct ___ac_attr___struct_1 { //Containerstruct fuer
            Funktionsattribute
64          template <typename ___ac_attr___ATTR, int I, int DUMMY = 0>
                struct ___ac_attr___Attr { static const int VALID = 0; };
65
66          template < int i , void ( * param ) ( int )>  struct
                ___ac_attr___type_0{};
67          template <int DUMMY> struct
                ___ac_attr___Attr<AC::Attr::acc::myAttr, 0, DUMMY> :
68          public ___ac_attr___type_0< 4 , func >{
69              static const int VALID = 1;
70              static const int i;
71              static void (*param) (int);
```

```
72              static const char *signature () { return
                    "[[acc::myAttr(4,func)]]";}
73          };
74
75          struct __ac_attr__type_1{};
76          template <int DUMMY> struct
                __ac_attr__Attr<AC::Attr::acc::hi, 0, DUMMY> :
77          public __ac_attr__type_1{
78              static const int VALID = 1;
79              static const char *signature () { return "[[acc::hi]]";}
80          };
81
82      };
83
84      int myFunc/*//_**__ac_attr__struct_1**_//*/ (const int a, const
            char b[], int c) {
85          return c;
86      }
87
88      //Introspection / Reflexion
89      public:
90      struct __TI {
91          static const char *signature () { return "myClass"; }
92          enum { HASHCODE = 515203373U };
93          typedef myClass That;
94          template<int I, int __D=0> struct BaseClass {};
95          enum { BASECLASSES = 0 };
96          template<int I, int __D=0> struct Member {};
97          enum { MEMBERS = 0 };
98          template<int I, int __D=0> struct Function {};
99          template<int I, int __D=0> struct Constructor {};
100         template<int I, int __D=0> struct Destructor {};
101         //Eigenschaften von func
102         template <int __D> struct Function<0, __D> { static const
                AC::Protection prot = AC::PROT_PUBLIC; static const
                AC::Specifiers spec = (AC::Specifiers)(AC::SPEC_NONE);
                template<typename ATTR, int I> struct Attribute { static
                const int VALID = 0; }; template<typename ATTR, int I>
                struct Attribute2 :
                __ac_attr__struct_1::__ac_attr__Attr<ATTR, I> {}; };
103         enum { FUNCTIONS = 1 };
104         enum { CONSTRUCTORS = 0 };
105         enum { DESTRUCTORS = 0 };
106         //Attribute der Klasse
107         template<typename ATTR, int I> struct Attribute :
                __ac_attr__struct_0::__ac_attr__Attr<ATTR, I> {};
108     };
109 }
```

```
110 //Initialisierungen
111 ;template<int DUMMY>
112 const int
        myClass::___ac_attr___struct_1::___ac_attr___Attr<AC::Attr::acc::myAttr,
        0, DUMMY>::i = 4;
113 template<int DUMMY>
114 void
        (*myClass::___ac_attr___struct_1::___ac_attr___Attr<AC::Attr::acc::myAttr,
        0, DUMMY>::param) (int) = func;
```

Beispiel A.1: Transformation Attributverwendung ohne Introspection.

In diesem Beispiel wird sowohl die Klasse mit einem Attribut als auch die Funktion mit zwei Attributen versehen. Da die Attribute bei der Transformation in einem dazu eigens kreierten *Namespace* abgelegt werden, wird ihre ursprüngliche Deklaration aus dem Quelltext gelöscht.

B. Objectdumps zu Attributzugriffen

B.1 Ohne Optimierungen (-O0)

Beispiel B.1: Konstantenzugriff ohne Optimierung.

```
1  ac.o:     Dateiformat elf64-x86-64
2
3
4  Disassembly of section .text:
5
6  0000000000000000 <main>:
7     0:   55                      push   %rbp
8     1:   48 89 e5                mov    %rsp,%rbp
9     4:   48 83 ec 20             sub    $0x20,%rsp
10    8:   c7 45 e0 00 00 00 00    movl   $0x0,-0x20(%rbp)
11    f:   c7 45 e4 08 00 00 00    movl   $0x8,-0x1c(%rbp)
12   16:   48 c7 45 e8 00 00 00    movq   $0x0,-0x18(%rbp)
13   1d:   00
14   1e:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 25
        <main+0x25>
15   25:   48 89 45 f0             mov    %rax,-0x10(%rbp)
16   29:   48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
17   30:   00
18   31:   8b 45 e4                mov    -0x1c(%rbp),%eax
19   34:   89 c6                   mov    %eax,%esi
20   36:   bf 00 00 00 00          mov    $0x0,%edi
21   3b:   e8 00 00 00 00          callq  40 <main+0x40>
22   40:   be 00 00 00 00          mov    $0x0,%esi
23   45:   48 89 c7                mov    %rax,%rdi
24   48:   e8 00 00 00 00          callq  4d <main+0x4d>
25   4d:   48 8b 45 f0             mov    -0x10(%rbp),%rax
26   51:   48 89 c6                mov    %rax,%rsi
27   54:   bf 00 00 00 00          mov    $0x0,%edi
28   59:   e8 00 00 00 00          callq  5e <main+0x5e>
29   5e:   be 00 00 00 00          mov    $0x0,%esi
30   63:   48 89 c7                mov    %rax,%rdi
31   66:   e8 00 00 00 00          callq  6b <main+0x6b>
```

```
32    6b:   48 8b 45 e8             mov    -0x18(%rbp),%rax
33    6f:   48 89 c6                mov    %rax,%rsi
34    72:   bf 00 00 00 00          mov    $0x0,%edi
35    77:   e8 00 00 00 00          callq  7c <main+0x7c>
36    7c:   be 00 00 00 00          mov    $0x0,%esi
37    81:   48 89 c7                mov    %rax,%rdi
38    84:   e8 00 00 00 00          callq  89 <main+0x89>
39    89:   48 8b 45 f8             mov    -0x8(%rbp),%rax
40    8d:   48 8b 00                mov    (%rax),%rax
41    90:   48 89 c6                mov    %rax,%rsi
42    93:   bf 00 00 00 00          mov    $0x0,%edi
43    98:   e8 00 00 00 00          callq  9d <main+0x9d>
44    9d:   be 00 00 00 00          mov    $0x0,%esi
45    a2:   48 89 c7                mov    %rax,%rdi
46    a5:   e8 00 00 00 00          callq  aa <main+0xaa>
47    aa:   b8 00 00 00 00          mov    $0x0,%eax
48    af:   c9                      leaveq
49    b0:   c3                      retq
50
51  00000000000000b1 <_Z41__static_initialization_and_destruction_0ii>:
52    b1:   55                      push   %rbp
53    b2:   48 89 e5                mov    %rsp,%rbp
54    b5:   48 83 ec 10             sub    $0x10,%rsp
55    b9:   89 7d fc                mov    %edi,-0x4(%rbp)
56    bc:   89 75 f8                mov    %esi,-0x8(%rbp)
57    bf:   83 7d fc 01             cmpl   $0x1,-0x4(%rbp)
58    c3:   75 31                   jne    f6
          <_Z41__static_initialization_and_destruction_0ii+0x45>
59    c5:   81 7d f8 ff ff 00 00    cmpl   $0xffff,-0x8(%rbp)
60    cc:   75 28                   jne    f6
          <_Z41__static_initialization_and_destruction_0ii+0x45>
61    ce:   bf 00 00 00 00          mov    $0x0,%edi
62    d3:   e8 00 00 00 00          callq  d8
          <_Z41__static_initialization_and_destruction_0ii+0x27>
63    d8:   ba 00 00 00 00          mov    $0x0,%edx
64    dd:   be 00 00 00 00          mov    $0x0,%esi
65    e2:   bf 00 00 00 00          mov    $0x0,%edi
66    e7:   e8 00 00 00 00          callq  ec
          <_Z41__static_initialization_and_destruction_0ii+0x3b>
67    ec:   bf 00 00 00 00          mov    $0x0,%edi
68    f1:   e8 00 00 00 00          callq  f6
          <_Z41__static_initialization_and_destruction_0ii+0x45>
69    f6:   90                      nop
70    f7:   c9                      leaveq
71    f8:   c3                      retq
72
73  00000000000000f9 <_GLOBAL__sub_I_bTest>:
74    f9:   55                      push   %rbp
```

```
75    fa:   48 89 e5                mov    %rsp,%rbp
76    fd:   be ff ff 00 00          mov    $0xffff,%esi
77   102:   bf 01 00 00 00          mov    $0x1,%edi
78   107:   e8 a5 ff ff ff          callq  b1
         <_Z41__static_initialization_and_destruction_0ii>
79   10c:   5d                     pop    %rbp
80   10d:   c3                     retq
81
82 Disassembly of section .text._ZN6RefObjC2Ev:
83
84 0000000000000000 <_ZN6RefObjC1Ev>:
85     0:   55                     push   %rbp
86     1:   48 89 e5               mov    %rsp,%rbp
87     4:   48 89 7d f8            mov    %rdi,-0x8(%rbp)
88     8:   48 8b 45 f8            mov    -0x8(%rbp),%rax
89     c:   48 c7 00 00 00 00 00   movq   $0x0,(%rax)
90    13:   90                     nop
91    14:   5d                     pop    %rbp
92    15:   c3                     retq
```

Beispiel B.2: Attributzugriff ohne Optimierung.

```
1  ac.o:     Dateiformat elf64-x86-64
2
3
4  Disassembly of section .text:
5
6  0000000000000000 <main>:
7     0:   55                      push   %rbp
8     1:   48 89 e5                mov    %rsp,%rbp
9     4:   48 83 ec 20             sub    $0x20,%rsp
10    8:   c7 45 e0 00 00 00 00    movl   $0x0,-0x20(%rbp)
11    f:   c7 45 e4 08 00 00 00    movl   $0x8,-0x1c(%rbp)
12   16:   e8 00 00 00 00          callq  1b <main+0x1b>
13   1b:   48 89 45 e8             mov    %rax,-0x18(%rbp)
14   1f:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 26
        <main+0x26>
15   26:   48 89 45 f0             mov    %rax,-0x10(%rbp)
16   2a:   48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
17   31:   00
18   32:   8b 45 e4                mov    -0x1c(%rbp),%eax
19   35:   89 c6                   mov    %eax,%esi
20   37:   bf 00 00 00 00          mov    $0x0,%edi
21   3c:   e8 00 00 00 00          callq  41 <main+0x41>
22   41:   be 00 00 00 00          mov    $0x0,%esi
23   46:   48 89 c7                mov    %rax,%rdi
24   49:   e8 00 00 00 00          callq  4e <main+0x4e>
25   4e:   48 8b 45 f0             mov    -0x10(%rbp),%rax
26   52:   48 89 c6                mov    %rax,%rsi
27   55:   bf 00 00 00 00          mov    $0x0,%edi
28   5a:   e8 00 00 00 00          callq  5f <main+0x5f>
29   5f:   be 00 00 00 00          mov    $0x0,%esi
30   64:   48 89 c7                mov    %rax,%rdi
31   67:   e8 00 00 00 00          callq  6c <main+0x6c>
32   6c:   48 8b 45 e8             mov    -0x18(%rbp),%rax
33   70:   48 89 c6                mov    %rax,%rsi
34   73:   bf 00 00 00 00          mov    $0x0,%edi
35   78:   e8 00 00 00 00          callq  7d <main+0x7d>
36   7d:   be 00 00 00 00          mov    $0x0,%esi
37   82:   48 89 c7                mov    %rax,%rdi
38   85:   e8 00 00 00 00          callq  8a <main+0x8a>
39   8a:   48 8b 45 f8             mov    -0x8(%rbp),%rax
40   8e:   48 8b 00                mov    (%rax),%rax
41   91:   48 89 c6                mov    %rax,%rsi
42   94:   bf 00 00 00 00          mov    $0x0,%edi
43   99:   e8 00 00 00 00          callq  9e <main+0x9e>
44   9e:   be 00 00 00 00          mov    $0x0,%esi
45   a3:   48 89 c7                mov    %rax,%rdi
```

```
46    a6:   e8 00 00 00 00          callq  ab <main+0xab>
47    ab:   b8 00 00 00 00          mov    $0x0,%eax
48    b0:   c9                      leaveq
49    b1:   c3                      retq
50
51  00000000000000b2 <_Z41__static_initialization_and_destruction_0ii>:
52    b2:   55                      push   %rbp
53    b3:   48 89 e5                mov    %rsp,%rbp
54    b6:   48 83 ec 10             sub    $0x10,%rsp
55    ba:   89 7d fc                mov    %edi,-0x4(%rbp)
56    bd:   89 75 f8                mov    %esi,-0x8(%rbp)
57    c0:   83 7d fc 01             cmpl   $0x1,-0x4(%rbp)
58    c4:   75 31                   jne    f7
          <_Z41__static_initialization_and_destruction_0ii+0x45>
59    c6:   81 7d f8 ff ff 00 00    cmpl   $0xffff,-0x8(%rbp)
60    cd:   75 28                   jne    f7
          <_Z41__static_initialization_and_destruction_0ii+0x45>
61    cf:   bf 00 00 00 00          mov    $0x0,%edi
62    d4:   e8 00 00 00 00          callq  d9
          <_Z41__static_initialization_and_destruction_0ii+0x27>
63    d9:   ba 00 00 00 00          mov    $0x0,%edx
64    de:   be 00 00 00 00          mov    $0x0,%esi
65    e3:   bf 00 00 00 00          mov    $0x0,%edi
66    e8:   e8 00 00 00 00          callq  ed
          <_Z41__static_initialization_and_destruction_0ii+0x3b>
67    ed:   bf 00 00 00 00          mov    $0x0,%edi
68    f2:   e8 00 00 00 00          callq  f7
          <_Z41__static_initialization_and_destruction_0ii+0x45>
69    f7:   90                      nop
70    f8:   c9                      leaveq
71    f9:   c3                      retq
72
73  00000000000000fa <_GLOBAL__sub_I_bTest>:
74    fa:   55                      push   %rbp
75    fb:   48 89 e5                mov    %rsp,%rbp
76    fe:   be ff ff 00 00          mov    $0xffff,%esi
77   103:   bf 01 00 00 00          mov    $0x1,%edi
78   108:   e8 a5 ff ff ff          callq  b2
         <_Z41__static_initialization_and_destruction_0ii>
79   10d:   5d                      pop    %rbp
80   10e:   c3                      retq
81
82  Disassembly of section .text._ZN6RefObjC2Ev:
83
84  0000000000000000 <_ZN6RefObjC1Ev>:
85     0:   55                      push   %rbp
86     1:   48 89 e5                mov    %rsp,%rbp
87     4:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
```

```
88      8:    48 8b 45 f8              mov    -0x8(%rbp),%rax
89      c:    48 c7 00 00 00 00 00     movq   $0x0,(%rax)
90     13:    90                       nop
91     14:    5d                       pop    %rbp
92     15:    c3                       retq
93
94  Disassembly of section
        .text._ZN19__ac_attr__struct_015__ac_attr__AttrIN2AC4Attr3acc9classAttrELi0ELi
95
96  0000000000000000
        <_ZN19__ac_attr__struct_015__ac_attr__AttrIN2AC4Attr3acc9classAttrELi0ELi0EE9si
97      0:    55                       push   %rbp
98      1:    48 89 e5                 mov    %rsp,%rbp
99      4:    b8 00 00 00 00           mov    $0x0,%eax
100     9:    5d                       pop    %rbp
101     a:    c3                       retq
```

B.2 Mit Optimierungen (-O3)

Beispiel B.3: Konstantenzugriff mit Optimierung `-O3`.

```
1  opt.o:     Dateiformat elf64-x86-64
2
3
4  Disassembly of section .text.startup:
5
6  0000000000000000 <main>:
7     0:   53                      push   %rbx
8     1:   48 8b 1d 00 00 00 00    mov    0x0(%rip),%rbx        # 8
         <main+0x8>
9     8:   be 08 00 00 00          mov    $0x8,%esi
10    d:   bf 00 00 00 00          mov    $0x0,%edi
11   12:   e8 00 00 00 00          callq  17 <main+0x17>
12   17:   48 89 c7                mov    %rax,%rdi
13   1a:   e8 00 00 00 00          callq  1f <main+0x1f>
14   1f:   48 89 de                mov    %rbx,%rsi
15   22:   bf 00 00 00 00          mov    $0x0,%edi
16   27:   e8 00 00 00 00          callq  2c <main+0x2c>
17   2c:   48 89 c7                mov    %rax,%rdi
18   2f:   e8 00 00 00 00          callq  34 <main+0x34>
19   34:   be 00 00 00 00          mov    $0x0,%esi
20   39:   bf 00 00 00 00          mov    $0x0,%edi
21   3e:   e8 00 00 00 00          callq  43 <main+0x43>
22   43:   48 89 c7                mov    %rax,%rdi
23   46:   e8 00 00 00 00          callq  4b <main+0x4b>
24   4b:   48 8b 35 00 00 00 00    mov    0x0(%rip),%rsi        # 52
         <main+0x52>
25   52:   bf 00 00 00 00          mov    $0x0,%edi
26   57:   e8 00 00 00 00          callq  5c <main+0x5c>
27   5c:   48 89 c7                mov    %rax,%rdi
28   5f:   e8 00 00 00 00          callq  64 <main+0x64>
29   64:   31 c0                   xor    %eax,%eax
30   66:   5b                      pop    %rbx
31   67:   c3                      retq
32   68:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
33   6f:   00
34
35 0000000000000070 <_GLOBAL__sub_I_bTest>:
36   70:   48 83 ec 08             sub    $0x8,%rsp
37   74:   bf 00 00 00 00          mov    $0x0,%edi
38   79:   e8 00 00 00 00          callq  7e <_GLOBAL__sub_I_bTest+0xe>
39   7e:   ba 00 00 00 00          mov    $0x0,%edx
40   83:   be 00 00 00 00          mov    $0x0,%esi
41   88:   bf 00 00 00 00          mov    $0x0,%edi
```

```
42    8d:    e8 00 00 00 00          callq  92 <_GLOBAL__sub_I_bTest+0x22>
43    92:    48 c7 05 00 00 00 00    movq   $0x0,0x0(%rip)        # 9d
          <_GLOBAL__sub_I_bTest+0x2d>
44    99:    00 00 00 00
45    9d:    48 83 c4 08             add    $0x8,%rsp
46    a1:    c3                      retq
```

Beispiel B.4: Attributzugriff mit Optimierung `-O3`.

```
opt.o:     Dateiformat elf64-x86-64


Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   53                      push   %rbx
   1:   48 8b 1d 00 00 00 00    mov    0x0(%rip),%rbx        # 8
      <main+0x8>
   8:   be 08 00 00 00          mov    $0x8,%esi
   d:   bf 00 00 00 00          mov    $0x0,%edi
  12:   e8 00 00 00 00          callq  17 <main+0x17>
  17:   48 89 c7                mov    %rax,%rdi
  1a:   e8 00 00 00 00          callq  1f <main+0x1f>
  1f:   48 89 de                mov    %rbx,%rsi
  22:   bf 00 00 00 00          mov    $0x0,%edi
  27:   e8 00 00 00 00          callq  2c <main+0x2c>
  2c:   48 89 c7                mov    %rax,%rdi
  2f:   e8 00 00 00 00          callq  34 <main+0x34>
  34:   be 00 00 00 00          mov    $0x0,%esi
  39:   bf 00 00 00 00          mov    $0x0,%edi
  3e:   e8 00 00 00 00          callq  43 <main+0x43>
  43:   48 89 c7                mov    %rax,%rdi
  46:   e8 00 00 00 00          callq  4b <main+0x4b>
  4b:   48 8b 35 00 00 00 00    mov    0x0(%rip),%rsi        # 52
      <main+0x52>
  52:   bf 00 00 00 00          mov    $0x0,%edi
  57:   e8 00 00 00 00          callq  5c <main+0x5c>
  5c:   48 89 c7                mov    %rax,%rdi
  5f:   e8 00 00 00 00          callq  64 <main+0x64>
  64:   31 c0                   xor    %eax,%eax
  66:   5b                      pop    %rbx
  67:   c3                      retq
  68:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  6f:   00

0000000000000070 <_GLOBAL__sub_I_bTest>:
  70:   48 83 ec 08             sub    $0x8,%rsp
  74:   bf 00 00 00 00          mov    $0x0,%edi
  79:   e8 00 00 00 00          callq  7e <_GLOBAL__sub_I_bTest+0xe>
  7e:   ba 00 00 00 00          mov    $0x0,%edx
  83:   be 00 00 00 00          mov    $0x0,%esi
  88:   bf 00 00 00 00          mov    $0x0,%edi
  8d:   e8 00 00 00 00          callq  92 <_GLOBAL__sub_I_bTest+0x22>
  92:   48 c7 05 00 00 00 00    movq   $0x0,0x0(%rip)        # 9d
      <_GLOBAL__sub_I_bTest+0x2d>
```

```
44    99:    00 00 00 00
45    9d:    48 83 c4 08              add     $0x8,%rsp
46    a1:    c3                       retq
```

Die beiden Ausgaben B.3 und B.4 unterscheiden sich nicht!

C. GOP Änderungen

Beispiel C.1: Änderungen am Mechanismus.

```
1  diff -c -r GOP(Alt)/ecc_classes/ChecksumConfiguration.ah
       GOP/ecc_classes/ChecksumConfiguration.ah
2  *** GOP(Alt)/ecc_classes/ChecksumConfiguration.ah    2016-01-15
       14:46:21.000000000 +0100
3  --- GOP/ecc_classes/ChecksumConfiguration.ah     2016-05-10
       12:52:31.197215350 +0200
4  ***************
5  *** 7,23 ****
6    #include "GOP_Common.ah"
7    #include "GOP_Static_Optimization.ah"
8
9
10   aspect ChecksumConfiguration : public GOP_Common {
11   public:
12     // they get a virtual checksum method (and thus a vtable)
13 !   pointcut criticalClasses() = derived("XYZ");
14
15     // use with caution (i.e. blacklisted classes must not access
           inherited protected attributes)
16     pointcut blacklist() = "Guarded_%";
17
18     // classes that have no derived classes an no 'critical' base
           classes //TODO: enforce that
19 !   pointcut standAloneCriticalClasses() = "Alone" || "ABC";
20
21     // multithreading
22     pointcut synchronizedClasses() = (criticalClasses() &&
           !blacklist()) || standAloneCriticalClasses();
23 --- 7,28 ----
24   #include "GOP_Common.ah"
25   #include "GOP_Static_Optimization.ah"
26
27 + namespace GOP {
28 +   attribute critical();
29 +   attribute StandAlone();
30 +   attribute uncriticalMember();
31 + }
```

```
32
33    aspect ChecksumConfiguration : public GOP_Common {
34    public:
35      // they get a virtual checksum method (and thus a vtable)
36  !   pointcut criticalClasses() = derived("[[GOP::critical]]");
37
38      // use with caution (i.e. blacklisted classes must not access
            inherited protected attributes)
39      pointcut blacklist() = "Guarded_%";
40
41      // classes that have no derived classes an no 'critical' base
            classes //TODO: enforce that
42  !   pointcut standAloneCriticalClasses() = "[[GOP::StandAlone]]";
43
44      // multithreading
45      pointcut synchronizedClasses() = (criticalClasses() &&
            !blacklist()) || standAloneCriticalClasses();
46
47  diff -c -r GOP(Alt)/ecc_classes/JPTL.h GOP/ecc_classes/JPTL.h
48  *** GOP(Alt)/ecc_classes/JPTL.h 2015-03-29 14:05:14.000000000 +0200
49  --- GOP/ecc_classes/JPTL.h  2016-05-09 23:30:52.812174576 +0200
50  ***************
51  *** 40,46 ****
52
53    // For looking up EXEC, use this dummy ones if the real one has no
          members/functions
54    // FIXME: this needs maintenance if future ac++ versions change
          this structures
55  ! template <int __D=0> struct DummyMember { typedef char Type;
      typedef AC::Referred<Type>::type ReferredType; static const
      AC::Protection prot = AC::PROT_PRIVATE; static const
      AC::Specifiers spec = AC::SPEC_NONE; static ReferredType *pointer
      (const void* const obj = 0) { return 0; } static const char *name
      () { return "null"; } };
56    template <int __D=0> struct DummyFunction { static const
          AC::Protection prot = AC::PROT_PRIVATE; static const
          AC::Specifiers spec = AC::SPEC_NONE; };
57    template <int __D=0> struct DummyConstructor { static const
          AC::Protection prot = AC::PROT_PRIVATE; static const
          AC::Specifiers spec = AC::SPEC_NONE; };
58    template <int __D=0> struct DummyDestructor { static const
          AC::Protection prot = AC::PROT_PRIVATE; static const
          AC::Specifiers spec = AC::SPEC_NONE; };
59  --- 40,46 ----
60
61    // For looking up EXEC, use this dummy ones if the real one has no
          members/functions
62    // FIXME: this needs maintenance if future ac++ versions change
```

```
      this structures
63 ! template <int __D=0> struct DummyMember { typedef char Type;
     typedef AC::Referred<Type>::type ReferredType; static const
     AC::Protection prot = AC::PROT_PRIVATE; static const
     AC::Specifiers spec = AC::SPEC_NONE; template<typename Attr, int
     I> struct Attribute {static const int VALID = 0;}; static
     ReferredType *pointer (const void* const obj = 0) { return 0; }
     static const char *name () { return "null"; } };
64   template <int __D=0> struct DummyFunction { static const
       AC::Protection prot = AC::PROT_PRIVATE; static const
       AC::Specifiers spec = AC::SPEC_NONE; };
65   template <int __D=0> struct DummyConstructor { static const
       AC::Protection prot = AC::PROT_PRIVATE; static const
       AC::Specifiers spec = AC::SPEC_NONE; };
66   template <int __D=0> struct DummyDestructor { static const
       AC::Protection prot = AC::PROT_PRIVATE; static const
       AC::Specifiers spec = AC::SPEC_NONE; };
67
68 diff -c -r GOP(Alt)/ecc_classes/ObjectSize.h
     GOP/ecc_classes/ObjectSize.h
69 *** GOP(Alt)/ecc_classes/ObjectSize.h    2016-01-15 14:46:21.000000000
     +0100
70 --- GOP/ecc_classes/ObjectSize.h     2016-05-09 23:35:32.742317894
     +0200
71 ***************
72 *** 56,61 ****
73 --- 56,62 ----
74             IS_STATIC_CONST = ((IS_STATIC == true) && (IS_CONST ==
                   true)),
75             IS_PLAIN    = (FoldArray<typename
                   MemberInfo::Type>::isClass == 0),
76             IS_UNSIZED_ARRAY = (__isUnsizedArray<typename
                   MemberInfo::Type>::value == true),
77 +           HAS_UNCRIT_ATTR = MemberInfo::template
     Attribute<AC::Attr::GOP::uncriticalMember, 0>::VALID,
78   //          IS_POINTER = (TypeTraits::pointerType<typename
       MemberInfo::Type>::result != TypeTraits::NO_POINTER_TYPE),
79             IS_CHECKSUMMED = (
80   #if GOP_USE_GET_SET_ADVICE
81 ***************
82 *** 66,71 ****
83 --- 67,73 ----
84                               && (IS_UNSIZED_ARRAY==false)
85                               && (IS_STATIC_CONST==false)
86                               && (IS_STATIC == STATIC)
87 +                             && (HAS_UNCRIT_ATTR==false)
88   //                            && (IS_POINTER==false)
89                               ) };
```

Im Listing C.1 sind die am GOP-Mechanismus selbst vorgenommen Änderungen dargestellt. In Listing C.2 dagegen sind die Änderungen am Beispielcode in `main.cpp` von GOP zu finden. Dabei sind tatsächlich geänderte Zeilen mit einem „`!`“ gekennzeichnet. Beim Rest handelt es sich lediglich um Kontext zum besseren Verständnis der Anpassungen.

Beispiel C.2: Änderungen des Beispielcodes.

```
diff -c -r GOP(Alt)/ecc_classes/main.cpp GOP/ecc_classes/main.cpp
*** GOP(Alt)/ecc_classes/main.cpp    2015-11-12 11:17:54.000000000
    +0100
--- GOP/ecc_classes/main.cpp     2016-05-10 10:54:48.243940710 +0200
***************
*** 1,11 ****
  #include <iostream>
  using namespace std;

! class XYZ {
  protected:
    char x; //2
    short y; //+2
!   long z; //+4 = 8 byte

  public:

--- 1,11 ----
  #include <iostream>
  using namespace std;

! class [[GOP::critical]] XYZ {
  protected:
    char x; //2
    short y; //+2
!   [[GOP::uncriticalMember]] long z; //+4 = 8 byte

  public:

***************
*** 22,28 ****
  private:
  //public:
    enum { FOASDASdsa, sdfsdfs };
!   long a; // 8 byte
    char b;
  public:
    void bar() { a = 4711; b = 12; /*((XYZ*)this)->*/foo(); }
--- 22,28 ----
  private:
  //public:
    enum { FOASDASdsa, sdfsdfs };
!   [[GOP::uncriticalMember]] long a; // 8 byte
    char b;
  public:
    void bar() { a = 4711; b = 12; /*((XYZ*)this)->*/foo(); }
```

```
46 ***************
47 *** 53,59 ****
48
49   };
50
51 ! class Alone {
52     int a[64+32];
53     private: static int alone;
54     public:
55 --- 53,59 ----
56
57   };
58
59 ! class [[GOP::StandAlone]] Alone {
60     int a[64+32];
61     private: static int alone;
62     public:
```