technische universität
dortmund

Bachelorarbeit

Globale Programmanalyse in AspectC++

Benedikt Freisen
8. August 2016

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Christoph Borchert

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

Zusammenfassung

AspectC++ ist als aspektorientierte aber gleichzeitig C++-basierte Programmiersprache vor die Herausforderung gestellt, sowohl den C++-üblichen inkrementellen Übersetzungsprozess zu unterstützen, als auch dem Anwender Mittel zur Verfügung zu stellen, die Modellierung der sogenannten querschneidenden Belange, für die die aspektorientierte Programmierung entwickelt wurde, so effizient wie möglich zu betreiben. Letzteres kann die globale Sicht auf das ganze Programm erfordern, ersteres erschwert diese.

Diese Arbeit untersucht Möglichkeiten, neben einem allgemeinen Ausbau der statischen Analysemöglichkeiten in AspectC++ durch geschickte Kombination von Wissensrevision und statischer Codeanalyse eine *globale Programmanalyse* zu ermöglichen, ohne das Prinzip des inkrementellen Übersetzens aufzugeben. Kernpunkt dieser Untersuchungen ist das vom AspectC++-Compiler bereits verwendete *persistente Modell*, in dem globales Wissen über das Projekt abgelegt und verwaltet wird.

Die Benutzbarkeit der entstandenen Lösungen wird sowohl anhand von einfachen Codebeispielen, als auch anhand von praktisch eingesetztem Code des aspektorientierten PUMA-Parsers und des Caching-Systems MEMCACHED überprüft.

Inhaltsverzeichnis

1. Einleitung

Im Bereich der Programmiersprachenunterstützung aspektorientierter Programmierung im Umfeld der Sprachen C und C++ kann seit Anfang des Jahrtausends auf die C++-Spracherweiterung AspectC++ [12] und den Werkzeugsatz der Referenzimplementierung zurückgegriffen werden.

Schon früh ergaben sich jedoch Probleme durch den Gegensatz zwischen der gängigen Arbeitsweise eines C++-Compilers, der Programme im Wesentlichen Quelltextdatei für Quelltextdatei übersetzt, und den Anforderungen an den Compiler einer aspektorientierten Sprache, der zum Ziel hat, den Anwender querschneidende Belange, die sich durch das gesamte Softwareprojekt ziehen, effizient modellieren zu lassen, wofür eine möglichst globale Sicht auf das Softwareprojekt gebraucht wird. [13, Kapitel 1.1]

Weil für diese Probleme im Umfeld von AspectC++ bislang keine praxistaugliche Lösung bereitsteht, müssen neue Lösungen gefunden werden, die die bewährte Arbeitsweise der C++-Compiler mit den Anforderungen der aspektorientierten Programmierung vereinen.

Diese Bachelorarbeit umfasst die Konzeption, Implementierung und Bewertung verschiedener Mechanismen, die auf Basis von globaler Programmanalyse und Wissensrevision die Benutzbarkeit der AspectC++-Sprache und -Infrastruktor verbessern sollen.

Das Verständnis der Konzepte und Begrifflichkeiten aus dem Grundlagenkapitel (2) wird im Folgenden vorausgesetzt.

1.1. Motivation

Bereits in seiner aktuellen Fassung gibt AspectC++ dem Entwickler viele Möglichkeiten an die Hand, eine Menge von Joinpoints mithilfe verschieder mächtiger vordefinierter Pointcutfunktionen und zusammengesetzter Ausdrücke kurz und präzise zu beschreiben.

Manche dieser vordefinierten Pointcutfunktionen, darunter `base` und `cflow`, können die von ihnen erwarteten Aufgaben jedoch mit dem begrenzten Wissen über die Programmstruktur, das in einer einzelnen Übersetzungseinheit enthalten ist, nur eingeschränkt erfüllen. Ihr Ergebnis kann nach wie vor korrekt sein, ist aber in

anderen Fällen je nach Strenge der Definition semantisch falsch, mindestens aber unintuitiv.

Weil die Aufteilung in unabhängige Übersetzungseinheiten für die Arbeit mit großen Softwareprojekten im praktischen Einsatz unabdingbar ist, sind an dieser Stelle neue Lösungen gefragt.

Nachfolgend wird zunächst allgemein beschrieben, welche Folgen die Unterteilung in Übersetzungseinheiten mit sich bringt. Im direkten Anschluss wird dies anhand von Beispielen konkretisiert.

1.1.1. Übersetzung am Stück vs. Übersetzungseinheiten

Abbildung 1.1 zeigt einen exemplarischen Übersetzungsvorgang, bei dem dem Compiler alle Quelltextdateien auf einmal übergeben werden. Der Compiler hat folglich prinzipiell Zugriff auf die komplette Programmstruktur.

Beim in Abbildung 1.2 gezeigten Übersetzungsvorgang hat der Compiler insbesondere bei der ersten (oberen) Übersetzungseinheit prinzipbedingt nur eingeschränktes Wissen über die globale Programmstruktur. Während das Wissen aus dieser Übersetzungseinheit im persistenten Modell festgehalten wird und somit prinzipiell bei einer nachfolgenden Übersetzungseinheit zur Verfügung stünde, ist zum Beispiel das Wissen aus b.cc und b.h in der ersten Übersetzungseinheit unzugänglich, weil es in der zweiten liegt.

Ferner ist anzumerken, dass die aktuelle Implementierung selbst das im persistenten Modell bereits verfügbare globale Wissen bei der Pointcutauswertung nicht benutzt.

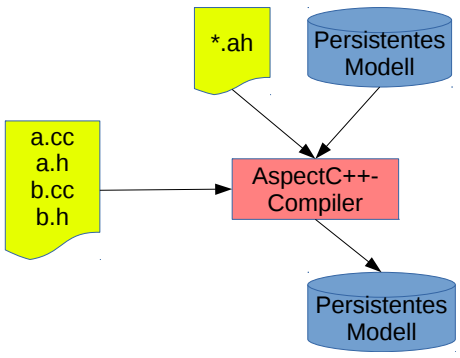


Abbildung 1.1.: Übersetzung in einem Schritt

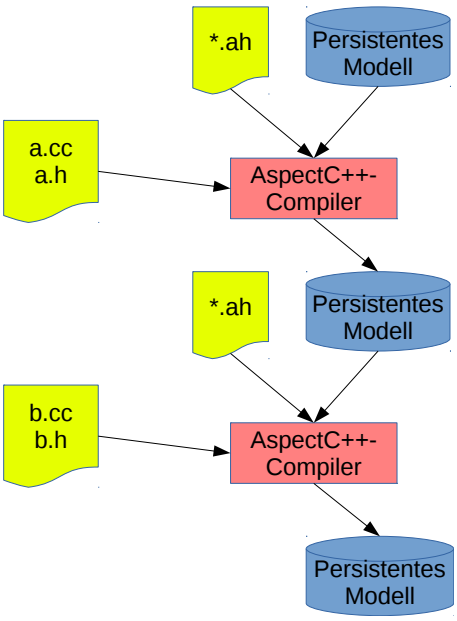


Abbildung 1.2.: Sequenzielle Übersetzung

1.1.2. Konkrete Probleme durch getrenntes Übersetzen

Wie zuvor beschrieben, kann bei der Pointcut-Auswertung zur Berechnung eines in Bezug auf das gesamte Programm korrekten Ergebnisses Wissen erforderlich sein, das in der Menge aus bisherigen Übersetzungseinheiten, einschließlich der aktuellen, nicht enthalten ist.

Bisher arbeitet in einem solchen Fall der AspectC++-Compiler nur mit dem Wissen aus der aktuellen Übersetzungseinheit, kennzeichnet das Ergebnis aber nicht als potenziell inkorrekt, wodurch es zu einem funktional inkorrekten Kompilat kommen kann, weil Annahmen zur globalen Programmstruktur, die sich im Nachhinein als falsch erweisen, nicht revidiert werden können.

Auch wenn Wissen über die globale Programmstruktur aus bisherigen Übersetzungseinheiten mithilfe des persistenten Modells hinzugezogen wird, bleibt das Ergebnis potenziell inkorrekt. Die Möglichkeit zur Revision bleibt erforderlich.

Die genaue Problematik wird anhand der folgenden Beispiele schnell offensichtlich.

1.1.2.1. Unbekannte Vererbungsbeziehungen

Das erste Beispiel besteht aus zwei Übersetzungseinheiten mit jeweils einer Klasse. Wie in Abbildung 1.3 zu sehen ist, erbt Klasse B von Klasse A.

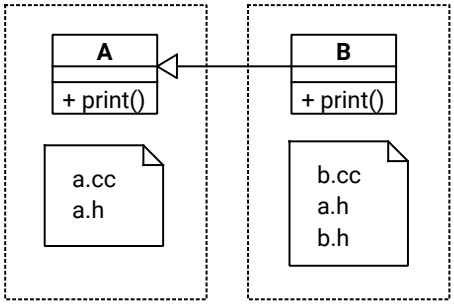


Abbildung 1.3.: Beispiel für die `base`-Pointcutfunktion

Nun soll innerhalb der Übersetzungseinheit von A der Advice aus 1.1 ausgewertet werden. Es soll also bestimmt werden, ob am Ende von `void A::print()` der Advice-Code eingewoben werden muss. In der aktuellen Implementierung wird die Antwort fälschlicherweise „nein“ lauten, weil das für die richtige Antwort notwendige Wissen in einer anderen Übersetzungseinheit liegt.

Vernachlässigt man den Umstand, dass der AspectC++-Compiler bisher Wissen aus anderen Übersetzungseinheiten bei der Pointcutauswertung nicht nutzen kann, wäre die Idee naheliegend, die Übersetzungseinheiten in umgekehrter Reihenfolge zu verarbeiten. Dies würde in der Tat bei diesem Beispiel zu einem korrekten Ergebnis führen.

Das vollständige Beispiel liegt in Anhang B.1 dieser Arbeit bei.

Listing 1.1: Aspekt zum base-Beispiel

```
1 aspect Ab {
2
3   advice execution(base("void B::print()")) : after() {
4     // advice code
5   }
6
7 };
```

1.1.2.2. Unbekannter Kontrollfluss

Beim folgenden Beispiel wird in der aktuellen Implementierung die funktionale Korrektheit durch obligatorische Laufzeittests gewährleistet. Dies geht jedoch zulasten der Effizienz.

Abbildung 1.4 zeigt zwei Klassen in unterschiedlichen Übersetzungseinheiten. Sie repräsentieren in vereinfachter Form ein Projekt. A::foo() und B::foo() rufen beide als einzige Funktionen A::bar() auf. Mit diesem Wissen ist sofort ersichtlich, dass der nachfolgende Advice 1.2 auf execution("void A::bar()") immer wirkt.

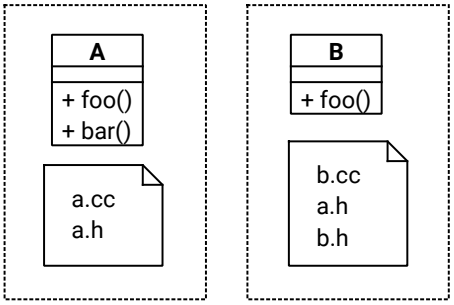


Abbildung 1.4.: Beispiel für die `cflow`-Pointcutfunktion

Ein Laufzeittst wäre also nicht erforderlich. Weil die aktuelle Implementierung im gegebenen Beispiel aber nur auf die Hälfte des für diese Optimierung notwendigen Wissens zurückgreifen kann, kann keine derartige Optimierung vorgenommen werden. Stattdessen muss, wenn der Pointcutausdruck keine weiteren Einschränkungen vorgibt, an jedem Joinpoint passenden Typs ein Laufzeittest eingewoben werden, der den Ursprung des Kontrollflusses dynamisch überprüft.

Listing 1.2: Aspekt zum cflow-Beispiel

```
1 aspect Ab {
2
3   advice cflow(call("% %::foo(...)")) &&  !within("Cflow") : after() {
4     // advice code
5   }
6
7 };
```

Darüber hinaus kann neben der Vergangenheit des Kontrollflusses, die mit `cflow` abgefragt werden kann, auch der zukünftige Verlauf desselben von Interesse sein.

Ob beispielsweise ein Programm bei der Ausführung einer gegebenen Funktion Ein- oder Ausgaben tätigt, kann mit den vorhandenen Pointcutfunktionen nicht bestimmt werden.

1.2. Zielsetzung

Dieses Kapitel beschreibt die Ziele dieser Arbeit, die aus der Abstraktion der in der Motivation aufgeführten Probleme abgeleitet werden können.

Während zum Beispiel die anhand der Beispiele in der Einleitung verdeutlichten Probleme Werkzeugerweiterungen erforderlich machen, kann der Wunsch nach einem Komplement zur `cflow`-Funktion am besten durch eine Spracherweiterung erfüllt werden.

Diese Erweiterungen sollen nach Möglichkeit so allgemein gehalten sein, dass sie einen Mehrwert haben, der über die Lösung der genannten Probleme hinaus geht und zukünftige Verbesserungen der Sprache und der Werkzeuge im Bereich globaler Programmanalyse vereinfacht.

1.2.1. Werkzeugerweiterungen zur globalen Programmanalyse

Für den zum Übersetzen von AspectC++-Projekten benutzten Werkzeugsatz sollen Konzepte erarbeitet und umgesetzt werden, die es dem Compiler erlauben, unter Beibehaltung der Unterteilung in Übersetzungseinheiten und der damit verbundenen Parallelisierbarkeit eine globale Programmanalyse zu betreiben.

Es sollen also Wege gefunden werden, dem Compiler Wissen über die Programmstruktur des gesamten Softwareprojekts, also in Bezug auf dieses Softwareprojekt globales Wissen, entweder bei jedem Übersetzungsvorgang zur Verfügung zu stellen, oder auf andere Weise ein mit der Ausgabe einer solchen Lösung äquivalente Ausgabe zu erzeugen.

1.2.2. Spracherweiterungen zur globalen Programmanalyse

Auf Basis dieser Erweiterungen soll die AspectC++-Sprache um Elemente ergänzt werden, mit denen der Anwender bei der Modellierung querschneidender Belange auf die Struktur des Softwareprojekts selbst Bezug nehmen kann. Das Mittel der globalen Programmanalyse soll also in der Sprache selbst zur Verfügung stehen.

Entsprechende Sprachelemente können beispielsweise die Analyse des Kontrollflussgraphs oder der Vererbungsbeziehungen zwischen Klassen ermöglichen, müssen sich aber nicht darauf beschränken.

1.3. Aufbau dieser Arbeit

In den bisherigen Teilen dieses Kapitels wurde die Motivation dieser Arbeit erläutert. Nachdem in Kapitel 2 „Grundlagen“ die wichtigsten in dieser Arbeit verwendeten Begriffe erklärt werden, werden in Kapitel 3 die Anforderungen zusammengefasst, die eine mögliche Implementierung in Anbetracht der in 1.2 beschriebenen Ziele erfüllen muss. Das Kapitel 4 „Entwurf“ leitet aus diesen Anforderungen mögliche Lösungsansätze ab, deren Auswahl und Umsetzung für die im Rahmen dieser Arbeit entstandene Implementierung im gleichnamigen Kapitel 5 detailliert beschrieben wird. Nachdem in Kapitel 6 „Bewertung“ die praktische Einsetzbarkeit der entstandenen Implementierung der einzelnen Komponenten und deren tatsächlicher Wert für die globale Programmanalyse reflektiert und mit bisherigen Lö-

sungen verglichen wird, gibt das Kapitel 7 „Zusammenfassung und Ausblick“ abschließend neben einer Zusammenfassung der Ergebnisse und daraus entstandenen neuen Möglichkeiten einen Ausblick auf mögliche und wünschenswerte zukünftige Entwicklungen, die die Ergebnisse dieser Arbeit ergänzen würden.

2. Grundlagen

Um in dieser Arbeit häufig vorkommende Grundbegriffe zu klären, enthält dieses Kapitel eine kurze Einführung in die wichtigsten Begriffe und Konzepte.

Dies umfasst Grundlagen der aspektorientierten Programmierung am Beispiel von AspectC++, eine Erläuterung gängiger Konzepte aus dem Bereich der Programmiersprache C++ und einen Einblick in die Repräsentation globalen Wissens in AspectC++ sowie in globale Programmanalyse im Allgemeinen.

2.1. AspectC++ allgemein

Der folgende Abschnitt erläutert die Grundbegriffe der aspektorientierten Programmierung am Beispiel von AspectC++, das in dieser Arbeit verwendet wird.

Kernkonzept der aspektorientierten Programmierung ist das Auslagern sogenannter querschneidender Belange (engl. Cross Cutting Concerns) [14] in Module.

Listing 2.1: Codebeispiel für einen Tracing-Aspekt in AspectC++

```
1 aspect Tracing {
2   pointcut foo_functions() = "% ...::foo_%(...)";
3   advice execution(foo_functions()) : before() {
4     cerr << tjp->signature() << " wurde betreten." << endl;
5   }
6   advice execution(foo_functions()) : after() {
7     cerr << tjp->signature() << " wird verlassen" << endl;
8   }
9 };
```

Im Folgenden werden die wesentlichen Konzepte der Sprache AspectC++ erläutert.

2.1.1. Querschneidende Belange

Der in der aspektorientierten Programmierung zentrale Begriff der querschneidenden Belange bezeichnet solche Belange, die sich bei der Softwareentwicklung weder mit prozeduraler, noch mit objektorientierter Programmierung effizient modularisieren lassen. [7, S. 1]

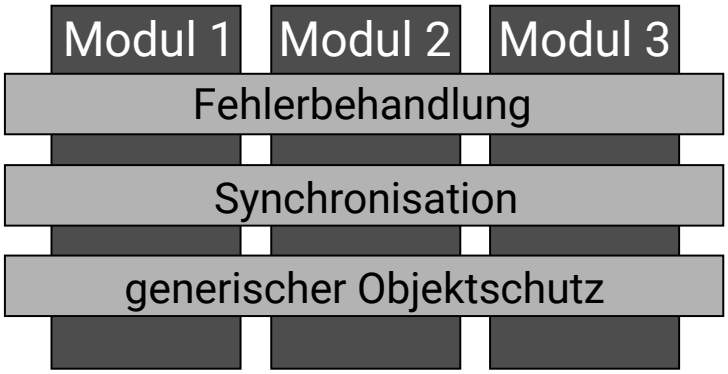


Abbildung 2.1.: Gängige Visualisierung querschneidender Belange mit Beispielen aus [7, S. 9] und [3, S. 2]

Bei den dort gegebenen Modularisierungsmöglichkeiten ziehen sie sich wie in Abbildung 2.1 illustriert quer durchs Programm, daher der Name querschneidende Belange.

2.1.2. Pointcuts und Joinpoints

Zwei wesentliche Grundbegriffe aus der aspektorientierten Programmierung, die in dieser Arbeit Verwendung finden, sind der des Pointcuts und der des Joinpoints.

2.1.2.1. Joinpoints

Ein Joinpoint ist eine Stelle im Programm oder dessen Struktur, an der ein Advice wirken kann. [13, Kapitel 4.1]

Es gibt zwei grundlegende Klassen von Joinpoints: Name-Joinpoints und Code-Joinpoints. Bei einem Name-Joinpoint handelt es sich um ein benanntes Element in der Programmstruktur, bei einem Code-Joinpoint um ein Ereignis während der Ausführung des Programms.

2.1.2.2. (Named) Pointcuts

Ein Pointcut ist eine Menge aus Joinpoints. [13, Kapitel 4.1] Mittels einer Pointcut-Deklaration kann einem — potenziell noch unbekannten — Pointcutausdruck ein Name zugeordnet werden. [14, Kapitel 2.1.4]

```
1 aspect A {
2   pointcut virtual a_named_pontcut() = 0;
3 };
4 aspect B : public A {
5   pointcut a_named_pointcut() = call("void foo()");
6 };
```

2.1.2.3. Vordefinierte Pointcutfunktionen

Die AspectC++-Sprache kennt eine Reihe verschiedener vordefinierter Pointcutfunktionen. [14] In dieser Arbeit werden davon die folgenden benutzt:

`base(pct)` gibt alle Klassen zurück, die direkte oder indirekte Basisklassen von Elementen aus `pct` sind, ebenso ihre Funktionen und Attribute. Darüber hinaus alle durch Elemente aus `pct` überschriebenen Funktionen und Attribute.

`derived(pct)` gibt alle Klassen zurück, die direkt oder indirekt von Elementen aus `pct` abgeleitet sind, ebenso ihre Funktionen und Attribute. Darüber hinaus alle Funktionen und Attribute, die Elemente aus `pct` überschreiben.

`cflow(pct)` gibt alle Joinpoints zurück, die im Kontrollfluss von einem Element aus `pct` aus erreichbar sind.

`call(pct)` gibt alle Code-Joinpoints zurück, die eine Funktion aus `pct` aufrufen.

`execution(pct)` gibt für jede Funktion in `pct` den Code-Joinpoint zurück, der den Rumpf dieser Funktion repräsentiert.

`construction(pct)/destruction(pct)` gibt alle Code-Joinpoints zurück, die die Konstruktion/Destruktion eines Objekts mit Typ aus `pct` darstellen.

Darüber hinaus gibt es die Operatoren **&&**, **||** und **!** für die logische Konjunktion, Disjunktion und Negation.

2.1.2.4. Pointcutausdrücke

Pointcutausdrücke in AspectC++ sind Sätze der Joinpoint-Beschreibungssprache, die aus den oben genannten Funktionen und Operatoren, Named Pointcuts und den nachfolgend anhand von Beispielen erklärten Ausdrücken gebildet werden können. [13, Kapitel 4.1]

`"MeineKlasse"` beschreibt den Pointcut mit allen Klassen oder Namensräumen mit Namen MeineKlasse.

`"char variable"` beschreibt den Pointcut mit allen Character-Variablen mit Namen „variable“.

`"% printf(...)"` beschreibt den Pointcut mit allen Funktionen mit Namen „printf“, beliebigem Rückgabetyp und beliebig vielen Parametern beliebigen Typs.

`"void %::foo()"` beschreibt den Pointcut mit allen Funktionen, die den Namen „foo“ und keinen Rückgabewert haben und in einer beliebigen Klasse oder einem beliebigen Namensraum liegen.

2.1.3. Aspekte und Advices

In AspectC++ gibt es Code-Advices und Introduction-Advices.

2.1.3.1. Code-Advices

Der Begriff Code-Advice bezeichnet die Kombination aus einem Code-Fragment und einer Beschreibung der Positionen in der dynamischen Programmausführung, an denen es eingebunden werden soll. [14, Kapitel 2.3]

```
1 advice call("% printf(...)") : before() {
2   // advice code
3 }
```

2.1.3.2. Introduction-Advices

Zweck von Introduction-Advices ist es, eine bestehende Klasse im Programm von außen abzuändern.

Einerseits können einer Klasse mithilfe einer *slice class* neue Elemente hinzugefügt werden.

```
1 advice "MyClass" : class slice {
2   // new members
3 }
```

Andererseits kann einer Klasse eine neue Basisklasse gegeben werden. [14, Kapitel 2.3.1]

```
1 advice "MyClass" : class slice : public NewBaseClass {
2   // redefinition of pure virtual member functions
3 }
```

Wird eine abstrakte Klasse als neue Basisklasse eingefügt, müssen ihre rein virtuellen Funktionen an dieser Stelle überschrieben werden.

2.1.3.3. Kapselung und Wiederverwendung

Zum Zwecke der Modularisierung werden ein oder mehrere Advices in einem Aspekt gekapselt. Dieser kann darüber hinaus Zustandsinformationen, Named Pointcuts und beliebige Elemente beinhalten, die auch in C++-Klassen vorkommen dürfen. [14, Kapitel 2.4]

2.1.4. AspectC++-Compiler

Die Referenzimplementierung des AspectC++-Compilers setzt sich aus einem Aspektweber und einem regulären C++-Compiler zusammen. Der Aspektweber erzeugt aus AspectC++-Code regulären C++-Code, der prinzipiell von jedem beliebigen C++-Compiler weiterverarbeitet werden kann. [13, Kapitel 3.2]

Wo die Unterscheidung zwischen Aspektweber und C++-Compiler im Zusammenhang dieser Arbeit nicht relevant ist, wird im Folgenden vereinfachend vom AspectC++-Compiler oder kurz Compiler gesprochen.

2.2. C++-Grundlagen

Einige Konzepte aus dem Bereich der Programmiersprache C++, die für diese Arbeit von besonderer Bedeutung sind, werden im Folgenden kurz zusammengefasst. Bei diesen Konzepten handelt es sich um den bei C++ üblichen Übersetzungsprozess, die Notation der Vererbungsbeziehungen zwischen Klassen und die Möglichkeit zur Umbenennung von Datentypen und Namespaces, sowie die Einbindung von Namespaces.

2.2.1. Übersetzungsprozess in den Sprachen C und C++

In C-verwandten Programmiersprachen wie C und C++ ist es üblich, nicht das komplette Programm am Stück zu übersetzen, sondern das Programm in einzelne Module, im Folgenden Übersetzungseinheiten genannt, aufzuteilen. Sie bestehen jeweils aus einer Quelltextdatei mit allen von ihr eingebundenen Dateien, werden einzeln übersetzt [6, Kapitel 5.1.1.1] [5, S. 17] und anschließend zum fertigen Programm gebunden.

2.2.2. Vererbung in C++

Die Vererbungsbeziehung zwischen einer Basisklasse und einer abgeleiteten Klasse muss in C++ bedingt durch die Syntax von C++-Klassen [5, S. 216] nur im Umfeld der abgeleiteten Klasse bekannt sein.

```
1  class Basisklasse {  // Vererbungsbeziehung wird nicht hier notiert,
2  public:
3    int ein_attribut;
4  };
5
6  class AbgeleiteteKlasse : public Basisklasse {  // sondern hier
7  public:
8    int noch_ein_attribut;
9  };
```

Werden die Klassen in unterschiedlichen Übersetzungseinheiten definiert, ist die Vererbungsbeziehung in der Übersetzungseinheit der Basisklasse nicht notwendigerweise bekannt.

2.2.3. Typ- und Namensraumaliase in C++

In C++ kann beliebigen Datentypen mittels `typedef` [5, Kapitel 7.1.3] für die spätere Benutzung ein alternativer Name zugeordnet werden. Namensräume können entweder direkt eingebunden, oder mit einem alternativen Namen versehen werden. Die Schlüsselwörter `namespace` und `using` [5, Kapitel 7.3], sowie `typedef` werden dabei wie folgt benutzt:

```
1  typedef alter_name neuer_name;
2  // Statt alter_name kann ab hier alternativ neuer_name benutzt werden
3
4  using namespace std;
5  // Elemente aus std koennen ab hier ohne std:: benutzt werden
6
7  namespace standard = std;
8  // Erlaubt z.B. standard::cout statt std::cout
```

2.3. Modellierung globaler Programmstrukturen

Wie im Motivationskapitel beschrieben wurde, kann es sinnvoll sein, nicht nur die lokale Programmstruktur innerhalb einzelner Übersetzungseinheiten, sondern die globale Struktur des gesammten Programms zu analysieren. Wissen über die Struktur des Programms muss dazu in geeigneter Form verfügbar gemacht werden.

Dieser Abschnitt geht zunächst auf globale Programmanalyse im Allgemeinen ein, um anschließend die Repräsentation globalen Wissens in AspectC++ und das Zusammenspiel mit dem Make-Werkzeug zu erläutern.

2.3.1. Globale Programmanalyse

Globale Programmanalyse, besser bekannt unter dem englischen Namen *whole-program analysis*, bezeichnet prinzipiell alle Verfahren, die ein Programm als ganzes analysieren. Im Folgenden wird exemplarisch auf die dynamische und statische Kontrollflussanalyse und die Analyse der Vererbungshierarchie eingegangen.

2.3.1.1. Dynamische Kontrollflussanalyse

Der Kontrollfluss eines Programms kann dynamisch, das heißt zur Laufzeit, analysiert werden. Zu diesem Zweck wird in das Programm manuell oder automatisch Tracing-Code eingefügt, der dann an einem bestimmten Punkt während der Programmausführung Aufschluss über die Vergangenheit des Kontrollflusses geben kann. Ein derartiger Ansatz wird in der bisherigen Implementierung der `cflow`-Pointcutfunktion in AspectC++ verwendet.

Aussagen über die Zukunft des Kontrollflusses sind mit diesem Ansatz prinzipbedingt nicht möglich.

2.3.1.2. Statische Kontrollflussanalyse

Im Bereich der Compileroptimierungen wurde schon früh dazu übergegangen, Teile eines Programms als gerichteter Graph zu modellieren, um Aussagen über den Kontrollfluss schon beim Übersetzen des Programms treffen zu können. [1]

Eine solche Analyse kann mit gängigen Graphenalgorithmen, insbesondere Suchalgorithmen, zum Beispiel auf Instruktions- oder Funktionsebene erfolgen.

2.3.1.3. Vererbungshierarchie

Ähnlich wie der Funktionsaufrufgraph kann auch die Vererbungshierarchie als gerichteter Graph modelliert werden. Dies erlaubt Analysen auch der indirekten Vererbungsbeziehungen zwischen einzelnen Klassen.

Weil es in C++ gängige Praxis ist, ab einer gewissen Größe jeder Klasse eine eigene Quelltextdatei zu geben, erfordert dies ebenfalls die dateiübergreifende globale Sicht.

2.3.2. Das Joinpoint-Modell

Für die interne Verarbeitung wird alles relevante Wissen über die Programmstruktur in eine interne Datenstruktur, das Joinpoint-Modell, übertragen. Dieses Wissen wird maßgeblich aus dem vom Frontend zur Verfügung gestellten abstrakten Syntaxbaum, wie er aus den Quelltextdateien erstellt wird, gewonnen. AspectC++

verwendet primär wie in [8] beschrieben das Clang-Frontend. Grundlagen zum Weg vom Quelltext zum Syntaxbaum können [9] entnommen werden.

Das Modell wird persistent gespeichert, um für die Verwendung bei späteren Übersetzungsprozessen partielles globales Wissen vorzuhalten. [13, S. 646, 6.1.3]

Wie in Beispiel 2.2 zu erkennen ist, referenziert jede Klasse ihre Basisklassen. Funktionen speichern Kindelemente, darunter auch Funktionsaufrufe, die ihrerseits die jeweilige Zielfunktion referenzieren. Es kann also sowohl die Vererbungshierarchie zwischen Klassen, als auch ein einfacher Funktionsaufrufgraph direkt aus dem Modell abgeleitet werden.

Der genaue Zusammenhang zwischen Quelltext und persistentem Modell lässt sich auch anhand des `base`-Beispiels in Anhang B.1 nachvollziehen.

2.3.3. Make-Prozess – GNU Make versus ACMake

Ein AspectC++-Projekt kann prinzipiell wie jedes gewöhnliche C++-Projekt mit einem GNU-Makefile erstellt werden. Dieses muss jedoch für die aspektorientierte Programmierung angepasst werden.

So muss zum Beispiel der C++-Compiler durch die AspectC++-Compiler ausgetauscht werden. Während ag++ mit dem Parameter `-p` die Aspektheader selbst findet, müssen diese jedoch in den Abhängigkeiten des make-Ziels mit angegeben werden. Dies kann direkt im Makefile, oder aber in separaten `*.d`-Dateien erfolgen.

Um diese Anpassungen am Makefile unnötig zu machen, wurde im Rahmen von [2] von Karen Bieling das Python-Werkzeug ACMake geschrieben, das als Eingabe ein unmodifiziertes Makefile eines C++-Projekts akzeptiert und dieses Projekt unter Zuhilfenahme von Aspektheaderdateien automatisch um die enthaltenen Aspekte erweitert.

Wie an der Gegenüberstellung zweier exemplarischer Makefiles in Abbildung 2.2 zu erkennen ist, muss sich der Autor des Makefiles dank ACMake nicht mehr mit den AspectC++-spezifischen Charakteristika des Software-Erstellungsprozesses auseinandersetzen. Weil die in dieser Arbeit präsentierten Lösungen diesen Prozess dynamisch abändern können müssen, wird die Verwendung von ACMake zur essenziellen Voraussetzung.

Das Werkzeug arbeitet parallelisiert und optional im Hintergrund, reagiert also selbstständig auf Änderungen in den Quelltextdateien.

Listing 2.2: Persistentes Modell in XML-Form

```
1  <?xml version="1.0"?>
2  <ac-model version="2.0" ids="3">
3    <files>
4      <TUnit filename="main.cc" len="42" time="1469823043" id="0"/>
5    </files>
6    <root>
7      <Namespace name="::">
8        <children>
9          <Class name="A" id="1">
10           <children>
11 ( Inhalt der Klasse A )
12           </children>
13           <source>
14             <Source kind="1" file="0" line="1" len="3"/>
15             <Source kind="2" file="0" line="1" len="1"/>
16           </source>
17         </Class>
18         <Class bases="1" name="B">
19           <children>
20 ( Inhalt der Klasse B )
21           </children>
22           <source>
23             <Source kind="1" file="0" line="5" len="3"/>
24             <Source kind="2" file="0" line="5" len="1"/>
25           </source>
26         </Class>
27         <Function kind="1" cv_qualifiers="0" name="myFunction" id="2">
28           <result_type>
29             <Type signature="void"/>
30           </result_type>
31           <children>
32             <Execution/>
33           </children>
34           <source>
35             <Source kind="1" file="0" line="9" len="3"/>
36           </source>
37         </Function>
38         <Function kind="1" cv_qualifiers="0" name="main">
39           <result_type>
40             <Type signature="int"/>
41           </result_type>
42           <arg_types>
43             <Type signature="int"/>
44             <Type signature="char␣**"/>
45           </arg_types>
46           <children>
47             <Execution/>
48             <Call target="2" lid="0">
49               <source>
50                 <Source kind="0" file="0" line="14" len="1"/>
51               </source>
52             </Call>
53           </children>
54           <source>
55             <Source kind="1" file="0" line="13" len="3"/>
56           </source>
57         </Function>
58       </children>
59     </Namespace>
60   </root>
61 </ac-model>
```

```
1  PROG:=base
2
3  SOURCES := a.cc b.cc main.cc
4
5  OBJECTS := Junk/a.o Junk/b.o Junk/main.o
6
7  HEADERS := a.h b.h
8
9
10
11 $(PROG): $(OBJECTS)
12     @echo Linking $@
13     @g++ -o $@ $(OBJECTS)
14
15 clean:
16     @rm -rf *.o Junk $(PROG)
17
18 Junk/%.o : %.cc
19     @mkdir -p Junk
20     @echo Compiling $<
21     @g++ -x c++ -c $< -o $@
22
23
24
25
26
27
28
29
30
31
32 .PHONY: clean
33
34 # don't remove any intermediate files
35 .SECONDARY:
```

(a) für C++ und ACMake

```
1  PROG:=base
2
3  SOURCES := a.cc b.cc main.cc
4
5  OBJECTS := Junk/a.o Junk/b.o Junk/main.o
6  DEPS    := Junk/a.d Junk/b.d Junk/main.d
7  HEADERS := a.h b.h
8  ASPECTS := ab.ah
9  INCLUDE_LIST := -include ab.ah
10
11 $(PROG): $(OBJECTS)
12     @echo Linking $@
13     @g++ -o $@ $(OBJECTS)
14
15 clean:
16     @rm -rf *.o Junk $(PROG)
17     @rm repo.acp
18
19 Junk/%.o : %.cc
20     @echo Compiling $<
21     @ag++ -x c++ -c $< -o $@
22
23 Junk/%.d : %.cc
24     @mkdir -p Junk
25     @g++ -x c++ -MM $(INCLUDE_LIST) $< | \
26     sed -e "s/$*\.o/Junk\/&␣Junk\/$*.d/g" > $@
27
28 ifneq ($(MAKECMDGOALS),clean)
29 -include $(DEPS)
30 endif
31
32 .PHONY: clean
33
34 # don't remove any intermediate files
35 .SECONDARY:
```

(b) für AspectC++ mit GNU Make

Abbildung 2.2.: Gegenüberstellung zweier einfacher Makefiles

3. Anforderungen

Aus den in der Motivation geschilderten Problemen ergeben sich an verschiedenen Stellen neue Anforderungen.

Dies sind zum einen Anforderungen an die Werkzeuge, darunter AspectC++-Compiler und ACMake, zum anderen Anforderungen an die Sprache selbst.

3.1. Neue Anforderungen an den Compiler

Der Compiler soll bei der Benutzung von Konstrukten zur globalen Programmanalyse in mehr Anwendungsszenarien als bisher korrekte und performante Kompilate erstellen können.

Dabei soll die Backend-Unabhängigkeit beibehalten werden, die sich aus der Quelltext-zu-Quelltext-Übersetzung ergibt und die Informationen im persistenten Modell sollen nicht an anderer Stelle dupliziert werden müssen.

Dies schließt beispielsweise Varianten der in [4] beschriebenen Link Time Optimization bereits aus, weil diese den Teil des Übersetzungsprozesses, der globales Wissen benötigt, vollständig in den Linker verlagert.

Im Detail bestehen darüber hinaus folgende Anforderungen:

3.1.1. Korrekte sequenzielle und inkrementelle Übersetzung

Die `base`-Pointcutfunktion muss die in [14, Kapitel 4.1] beschriebene Semantik in Bezug auf das Gesamtprojekt einhalten, auch wenn das Projekt in Form einzelner Übersetzungseinheiten verarbeitet wird.

Wenn eine im Rahmen dieser Arbeit entstehende Optimierung für `cflow` die semantische Korrektheit des Kompilats unter bestimmten Bedingungen gefährdet, muss sie abschaltbar sein.

Wird als Lösungsansatz Wissensrevision gewählt, dann muss der Compiler in die Lage versetzt werden, unzutreffende Annahmen zu erkennen und zu markieren. Bei einem weiteren Aufruf mit der gleichen Übersetzungseinheit muss er in diesem Fall die anfängliche Annahme korrigieren können, er darf diese also nicht wiederholen.

3.1.2. Kein Performanceeinbruch bei `cflow`-Nutzung

Wie in 1.1.2.2 beschrieben, kann in der bisherigen Implementierung das auch nur einmalige Benutzen der `cflow`-Funktion das obligatorische Einweben von Laufzeittests an allen Joinpoints passenden Typs zur Folge haben, sofern weitere Einschränkungen fehlen.

Die Implementierung sollte so abgeändert werden, dass an Joinpoints, die sich gar nicht im Kontrollfluss des Pointcuts befinden können, keine Laufzeittests eingewoben werden. Gleiches gilt für Joinpoints, die sich ausschließlich im Kontrollfluss des Pointcuts befinden können.

Um dies zu erreichen, muss der Compiler um Mittel zur statischen Kontrollflussanalyse erweitert werden, die es erlauben, die Anzahl benötigter Tests zur Laufzeit stark zu reduzieren.

3.2. Neue Anforderungen an ACMake

Das Make-Werkzeug ACMake muss im oben gennanten Fall in die Lage versetzt werden vom Compiler entsprechend markierte Übersetzungseinheiten zu erkennen und muss für diese in geeigneter Weise einen neuen Übersetzungsvorgang anstoßen.

Hierbei sollte die Parallelisierbarkeit bestmöglich gewahrt bleiben.

3.3. Neue Anforderungen an die Sprache

Die Sprache muss um Möglichkeiten erweitert werden, von einem Punkt im Kontrollfluss Aussagen nicht nur über den bisherigen, sondern auch über den zukünftigen Kontrollfluss treffen zu können.

Dies sollte in Form einer neuen Pointcutfunktion geschehen.

4. Entwurf

In diesem Kapitel werden im Unterkapitel Spekulatives Weben die Probleme behandelt, die sich durch sequenzielles Weben und Übersetzen für die statische Analyse ergeben. Ein weiteres Unterkapitel befasst sich mit einer für die statische Analyse wünschenswerten Spracherweiterung, die zu `cflow` komplementär ist.

4.1. Spekulatives Weben

Dieses Unterkapitel umfasst Lösungsansätze, die sowohl eine Revidierbarkeit falscher Annahmen durch Einfügung entsprechender Markierungen ermöglichen, als auch diese Revision vornehmen.

4.1.1. Identifikation bisher impliziter Spekulationen

Bisher geht der Compiler beim Verarbeiten der `base`-Pointcutfunktion stillschweigend davon aus, allein auf Basis des Wissens aus der aktuellen Übersetzungseinheit entscheiden zu können, ob eine Basisklassenbeziehung vorliegt.

Wie in der Motivation gezeigt, können Basisklassenbeziehungen aber durch andere — auch durch bisher unbekannte — Übersetzungseinheiten beeinflusst werden.

Dieses Problem kann auf zwei Arten angegangen werden.

4.1.1.1. Obligatorische Markierung ausgewählter Pointcuts als spekulativ

Dies ist der triviale Lösungsansatz. Bei Pointcutausdrücken, die die `base`-Funktion enthalten, wird das Ergebnis der Auswertung immer als spekulativ markiert. Dieser spekulative Wert wird fortan bei Erweiterung der Wissensbasis immer überprüft. Der Overhead ist entsprechend groß.

4.1.1.2. Einzelfallentscheidung auf Basis des Pointcutausdrucks

Eine eingehendere Analyse des untergeordneten Pointcutausdrucks kann Aufschluss darüber geben, ob im Einzelfall hinzukommendes Wissen das Ergebnis überhaupt beeinflussen kann.

Sind alle Bezeichner im Pointcutausdruck, in dem die Named Pointcuts zuvor expandiert wurden bekannt und enthält er auch keine Platzhalter, dann kann der Pointcutausdruck mit Blick auf das bekannte Wissen als *geschlossen* angesehen werden, weil er in diesem Sinne keine offenen Variablen mehr enthält. Ist dies nicht der Fall, kann man den Ausdruck als *nicht geschlossen* oder auch *offen* ansehen.

Ein entsprechender Mechanismus kann in einer Funktion `bool is_open()` abgelegt werden.

4.1.2. Revidierbarkeit bisher impliziter Spekulationen

Um die oben genannte Revidierbarkeit gewährleisten zu können, müssen die Komponenten der Pointcut-Auswertung, deren Ergebnisse sich durch später hinzu kommendes Wissen ändern können, Metainformationen im persistenten Modell ablegen.

Hierbei kann es sich um Informationen zu jedem Primitiv eines Pointcutausdrucks handeln, d.h. zu jeder Pointcutfunktion und jedem Pointcutoperator, oder um aggregierte Informationen zu je einem kompletten Pointcutausdruck, d.h. das aktuell spekulierte Ergebnis.

Weil Pointcutausdrücke derzeit nur in `string`-Form, nicht aber als geparste Baumstruktur im persistenten Modell abgelegt werden, ist der zweite Ansatz praktikabler. Er soll deshalb Basis der Implementierung werden.

Es bietet sich an, in das Joinpointmodell eine Spekulationsklasse zu integrieren, deren Objekte pro Advice und Übersetzungseinheit die spekulative Zugehörigkeit von Joinpoints zum Ergebnispointcut festhalten können.

Aufgrund des Umstands, dass die `cflow`-Auswertung notfalls auf Laufzeittests zurückgreifen können muss, müssen drei verschiedene Werte möglich sein:

- `true`: Der Joinpoint ist Element des Pointcuts
- `false`: Der Joinpoint ist kein Element des Pointcuts
- `conditional`: Die Entscheidung muss zur Laufzeit getroffen werden

Der spekulative Charakter besagt, dass ein korrektes Ergebnis zwar generell zur Übersetzungszeit berechnet werden kann, möglicherweise aber nicht ohne das Wissen aus weiteren Übersetzungseinheiten.

4.1.3. Logische Verknüpfung spekulativer Werte

Bei der logischen Verknüpfung zweier Werte kann in bestimmten Fällen der spekulative Charakter der Operanden im Ergebnis wegfallen. Auch entfällt abhängig

&&	**false**	**true**	**cond.**	**sp. false**	**sp. true**	**sp. cond.**
false	false	false	false	false	false	false
true	false	true	cond.	sp. false	sp. true	sp. cond.
cond.	false	cond.	cond.	sp. false	sp. cond.	sp. cond.
sp. false	false	sp. false	sp. false	sp. false	sp. false	sp. false
sp. true	false	sp. true	sp. cond.	sp. false	sp. true	sp. cond.
sp. cond.	false	sp. cond.	sp. cond.	sp. false	sp. cond.	sp. cond.

Tabelle 4.1.: Logische UND-Verknüpfung zweier spekulativer Werte

||	**false**	**true**	**cond.**	**sp. false**	**sp. true**	**sp. cond.**
false	false	true	cond.	sp. false	sp. true	sp. cond.
true	true	true	true	true	true	true
cond.	cond.	true	cond.	sp. cond.	sp. true	sp. cond.
sp. false	sp. false	true	sp. cond.	sp. false	sp. true	sp. cond.
sp. true	sp. true	true	sp. true	sp. true	sp. true	sp. true
sp. cond.	sp. cond.	true	sp. cond.	sp. cond.	sp. true	sp. cond.

Tabelle 4.2.: Logische ODER-Verknüpfung zweier spekulativer Werte

von den Operanden der Laufzeittest. Tabelle 4.1 zeigt die Wahrheitstabelle für die UND-Verknüpfung, Tabelle 4.2 die Wahrheitstabelle für die ODER-Verknüpfung.

Der möglicherweise unintuitive Charakter dieser Logik besteht darin, dass `conditional` streng genommen keinen Ergebniswert, sondern dessen Abwesenheit bezeichnet. Der tatsächliche Ergebniswert muss in diesem Fall zur Laufzeit des Programms berechnet werden.

4.1.4. Möglichkeiten für sequenziellen Übersetzungsvorgang

Es gibt in erster Linie zwei verschiedene Möglichkeiten bei einem spekulativen Webe- und Übersetzungsvorgang, der Quelldateien sequenziell einzeln verarbeitet, die Korrektheit des Kompilats sicher zu stellen.

Beide arbeiten in unterschiedlicher Weise mit dem Make-Wekzeug zusammen. Eine Kombination der Ansätze ist möglich.

4.1.4.1. Heuristik und abschließender Korrekturdurchlauf

Die spekulativen Werte für das Ergebnis der Auswertung eines Pointcut-Ausdrucks können mittels einer Heuristik bestimmt werden. Das Ergebnis ist in diesem Fall nicht notwendigerweise mit dem aktuellen Zustand des Modells konsistent.

Der Compiler braucht in diesem Fall zwei separate Betriebsmodi. Im ersten Modus werden heuristisch Entscheidungen getroffen, die mit dem Modell nicht konsistent sein müssen, aber eventuell zukünftiges Wissen korrekt voraussehen und so ein Neuübersetzen überflüssig machen. Der erste Modus führt keine interne Konsistensprüfung durch. Im zweiten Modus muss die Konsistenz mit der aktuellen Version des Modells garantiert sein. In diesem zweiten Modus wird darüber hinaus bei jedem Übersetzungsvorgang die Konsistenz mit dem Modell geprüft und entsprechende Informationen als Metainformationen ins Modell geschrieben. Die Arbeitsweise beider Betriebsmodi wird in Abbildung 4.1 veranschaulicht.

Es wäre Aufgabe des Make-Werkzeugs diese Modi so zu kombinieren, dass ein korrektes Kompilat entsteht.

4.1.4.2. Permanente Konsistenzprüfung

Dieser Ansatz entspricht dem zweiten oben genannten Modus. Die Auswahl spekulativer Ergebniswerte wird bei der Auswertung von Pointcut-Ausdrücken so gestaltet, dass die Ausgabe stets mit der aktuellen Version des Modells konsistent ist beziehungsweise der Umstand der Inkonsistenz im Modell festgehalten wird.

4.1.4.3. Zusammenspiel mit dem Make-Werkzeug

Beim ersten Ansatz wird ein zusätzlicher Schritt im Make-Prozess erforderlich, der nach Verarbeitung aller Quelldateien den Modus wechselt und alle Quelldateien noch einmal abarbeitet, um sie auf Konsistenz mit dem Modell prüfen und gegebenenfalls neu übersetzen zu lassen.

Beim zweiten Ansatz müssen nach der Abarbeitung aller Quelltextdateien lediglich alle im Modell als inkonsistent markierten Übersetzungseinheiten so lange in einem weiteren Durchlauf neu übersetzt werden, bis keine mehr als inkonsistent markiert ist.

In beiden Fällen würden diese Aufgaben dem existierenden AspectC++-Make-Werkzeug ACMake zufallen. Prinzipiell kann der zweite Ansatz jedoch auch ohne spezialisiertes Make-Werkzeug arbeiten. Dazu müssten die Quelldateien des gesamten Projekts mehrfach hintereinander übersetzt werden. Beim ersten Ansatz hingegen müssten immer zwei Make-Targets hintereinander erstellt werden. Eine Deklaration als `PHONY`-Target kann dabei sicher stellen, dass der jeweils zweite Übersetzungsvorgang für eine unveränderte Quelldatei nicht übersprungen wird. [15]

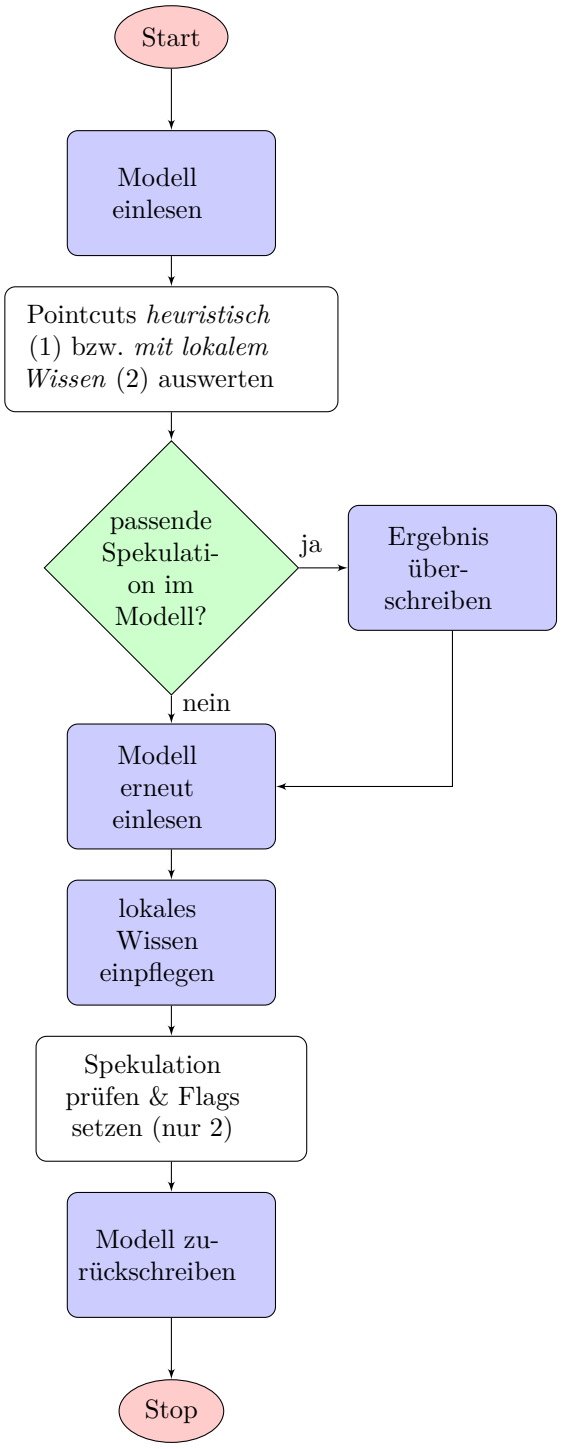


Abbildung 4.1.: Flussdiagramm zum Übersetzungsvorgang — beide Modi

4.2. Analyse des Funktionsaufrufgraphs

Die neue Pointcutfunktion muss es erlauben, einen Konfidenzwert anzugeben und darf in diesem Fall nur Ergebnisse mit der angegebenen Konfidenz zurückgeben.

Die Notwendigkeit des Konfidenzwertes ergibt sich aus den Unzulänglichkeiten der statischen Analyse. So kann zum Beispiel in einer Sprache wie C++, die Zeiger oder Referenzen enthält, im Voraus weder entschieden werden, ob diese zu einem bestimmten Zeitpunkt während der Programmausführung auf ein gegebenes Element zeigen müssen, noch ob sie auf dieses Element überhaupt zeigen können. [11, S. 1] Dies kann eine statische Analyse des Funktionsaufrufgraphs stark behindern, weil auch Funktionsaufrufe in C++ über Zeiger oder Referenzen erfolgen können. Das Ziel beziehungsweise der Ursprung eines solchen Funktionsaufrufs entzieht sich dann der statischen Analyse, es können allenfalls Spezialfälle erfasst werden.

Ebenso kann bedingt durch das Halteproblem nicht zwingend in endlicher Zeit ermittelt werden, ob ein Codeblock, der in der Programmausführung zeitlich vor einem Joinpoint liegt, überhaupt terminiert. Die Erreichbarkeit des Joinpoints kann also auch ohne zeigerbasierte Funktionsaufrufe weder belegt noch ausgeschlossen werden.

Der Anwender muss daher die Möglichkeit erhalten, ob er nur Joinpoints auswählen möchte, für die sich die Erreichbarkeit belegen lässt, oder solche für die sich die Erreichbarkeit nicht ausschließen lässt. Ersteres wäre ein korrektes, aber nicht notwendigerweise vollständiges Ergebnis, letzteres ein garantiert vollständiges Ergebnis, das aber möglicherweise nicht korrekt ist.

Der korrekte Pointcut wäre schlimmstenfalls leer, während der vollständige im ungünstigsten Fall alle Joinpoints passenden Typs im Modell enthielte.

4.2.1. Einführung der `cflowto`-Pointcutfunktion

Als zur `cflow`-Funktion komplementäre Funktion kann die in [3, Kapitel 6] beschriebene `cflowto`-Funktion eingebaut werden.

Sie erlaubt eine Selektion der Joinpoints, die von einem Element des Parameter-Pointcuts aus erreichbar sind und muss zu diesem Zweck Zugriff auf das globale Joinpoint-Modell haben.

Die `cflowto`-Funktion ist parametrisierbar. Sie akzeptiert als zusätzlichen Parameter einen Prozentwert, der die gewünschte Konfidenz des Ergebnisses spezifiziert. Die Notation ist an die C++-Templatesyntax [5, S. 321] angelehnt.

Die Ergebnismenge von `cflowto<0>` soll garantiert vollständig sein, die Ergebnismenge von `cflowto<100>` garantiert korrekt.

Die Schreibweise `cflowto(...)` soll dabei `cflowto<0>(...)` abkürzen.

4.2.2. Semantik für `cflowto<0>`

Gegeben pct setzt sich der Pointcut `cflowto<0>`(pct) aus allen Joinpoints jpl aus dem Modell zusammen, für die gilt:

- Gibt es einen Pfad aus ACM_Call- und ACM_Function-Objekten von jpl zu einem Element von pct, dann ist jpl Element von cflowto<0>(pct).
- Gibt es einen Pfad aus ACM_Call- und ACM_Function-Objekten von jpl zu einem ACM_Function-Objekt, zu dem keine Funktionsdefinition bekannt ist, dann ist jpl Element von cflowto<0>(pct).
- Gibt es einen Pfad aus ACM_Call- und ACM_Function-Objekten von jpl zu einem ACM_Function-Objekt, das eine virtuelle Funktion repräsentiert, dann ist jpl Element von cflowto<0>(pct).
- Gibt es einen Pfad aus ACM_Call- und ACM_Function-Objekten von jpl zu einem ACM_CallRef-Objekt, dann ist jpl Element von cflowto<0>(pct).

4.2.3. Semantik für `cflowto<1>`

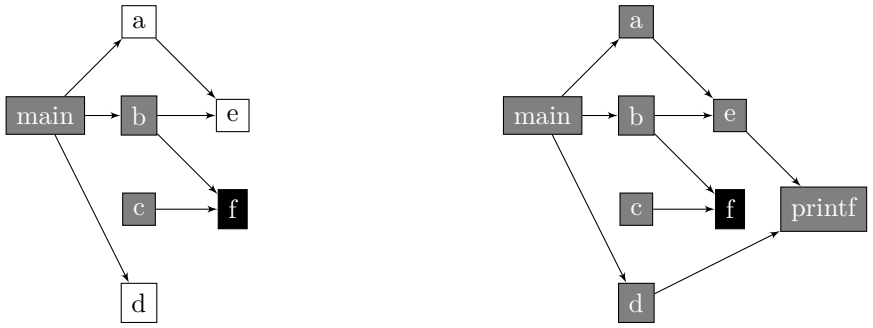
Bei der praktischen Anwendung von `cflowto<0>` würde zum Problem werden, dass pessimistisch davon ausgegangen werden muss, dass sobald der Kontrollfluss eine Funktion erreicht, zu der keine Definition bekannt ist, von dort aus alle Code-Joinpoints im Modell erreicht werden können.

Weil aber nahezu jedes Softwareprojekt Bibliotheksfunktionen aufruft, diese aber üblicherweise ihrerseits keine Funktionen aus dem aktuellen Projekt aufrufen, schränkt dies die Präzision des `cflowto`-Ausdrucks in vielen Anwendungsszenarien unnötig ein. Zur Veranschaulichung soll in Abbildung 4.2 `cflowto<0>(execution("% f()"))` bestimmt werden. Durch die Bibliotheksfunktion `printf`, deren Definition nicht bekannt ist, verliert das Ergebnis an Aussagekraft, weil pessimistisch davon ausgegangen werden muss, dass `f` aus `printf` heraus erreichbar ist.

Zur Lösung dieses Problems lässt sich die Spezialisierung `cflowto<1>` so definieren, dass sie weitestgehend zu `cflowto<0>` äquivalent ist, aber Aufrufe von Funktionen mit unbekannter Definition nicht beachtet. Der zweite Punkt aus 4.2.2 entfällt.

4.2.4. Semantik für `cflowto<n>` mit $2 \leq n \leq 100$

Weil zur Bestimmung nicht-trivialer Elemente eines korrekten `cflowto`-Pointcuts der Kontrollfluss innerhalb von Funktionsrümpfen analysiert werden müsste, dieser



(a) Alle Funktionen definiert (b) Funktion `printf` ohne Definition

Abbildung 4.2.: Visualisierung des „printf-Problems“

aber im Joinpoint-Modell nicht erfasst wird und ein Nachrüsten entsprechender Funktionalität den Rahmen dieser Arbeit sprengen würde, erscheint es an dieser Stelle wenig sinnvoll für die verbleibenden 99 `cflowto`-Funktionen eine genaue Semantik zu definieren.

Für Konfidenzwerte von 2 bis 100 Prozent sollte die `cflowto`-Funktion jedoch Ergebnisse mit steigender Korrektheit und potenziell fallender Vollständigkeit zurückgeben.

Auch wenn an dieser Stelle keine genaue Semantik definiert wird, sollten die entsprechenden Werte für diesen Zweck reserviert werden.

4.2.5. Konzeption der `cflow`-Optimierung

Die bisherige Implementierung der `cflow`-Pointcutfunktion soll um Mittel zur statischen Kontrollflussanalyse erweitert werden, die es erlauben, in vielen Fällen auszuschließen, dass ein gegebener Joinpoint im Kontrollfluss von Elementen des Pointcuts erreicht werden kann.

Dies erspart an Stellen, an denen ein `cflow`-Advice niemals wirken kann den ansonsten obligatorischen Laufzeittest.

Zu diesem Zweck muss bei der `cflow`-Auswertung ausgehend von jedem Element des Parameter-Pointcuts die Erreichbarkeit eines jeden Joinpoints über Kontrollflusselemente wie Funktionsaufrufe und Funktionsabläufe überprüfen.

Das Verfahren entspricht weitestgehend der `cflowto<0>`-Analyse. Für weitere Details sei auf das Unterkapitel 4.2.2 verwiesen.

Darüber hinaus muss ein Mechanismus eingebaut werden, der im Kontrollflussgraph rückwärts sucht und prüft, ob der jeweilige Joinpoint von einem Element des Pointcuts erreicht worden sein muss.

5. Implementierung

Dieses Kapitel beschreibt detailliert die Implementierung, die als Ergebnis dieser Arbeit entstanden ist. Dies beinhaltet insbesondere die Auswahl und Kombination der Lösungsansätze aus Kapitel 4, die Anwendung fanden, und deren genaue Umsetzung. Dabei wird zwischen Erweiterungen des Compilers, des Make-Werkzeugs, der Sprache und der Benutzerschnittstelle unterschieden.

5.1. Erweiterung des AspectC++-Compilers

Die folgenden Erweiterungen wurden auf Basis der Entwürfe aus dem vorangegangenen Kapitel für den AspectC++-Compiler erarbeitet.

5.1.1. Verwaltung persistenter Spekulationsobjekte

Das Joinpoint-Modell ACModel wurde um eine Spekulationsklasse erweitert. Die Spekulationsklasse und ihre Beziehungen zu anderen Klassen sind in Abbildung 5.1 gezeigt. Das nachfolgende Listing enthält exemplarische Spekulationseinträge.

Die Unterteilung der Liste der vom Spekulationsobjekt betroffenen Joinpoints in `true_jps`, `false_jps` und `conditional_jps` rührt daher, dass wie im Beispiel 5.1 zu sehen ist, Joinpoint-IDs so kompakter aufgelistet werden können als wenn jeweils ein Tupel aus Joinpoint-ID und Wert gespeichert werden würde.

Listing 5.1: Spekulationseinträge im persistenten Modell

```
1    </root>
2    <speculations>
3      <Speculation advice="4" tunit="0" true_jps="12␣13" false_jps="14␣15"/>
4      <Speculation advice="4" tunit="1" true_jps="16␣17" false_jps="18␣19"/>
5      <Speculation advice="4" tunit="2" true_jps="12␣20" false_jps="14␣15"/>
6    </speculations>
7  </ac-model>
```

Darüber hinaus wurde der AspectC++-Compiler dahingehend erweitert, dass er nun am Ende des Übersetzungsvorgangs jeder Übersetzungseinheit wie in Abbildung 5.2 alle Spekulationsobjekte im aktualisierten globalen Modell vor dem Zurückschreiben auf Konsistenz mit diesem testet und bei negativem Ergebnis die

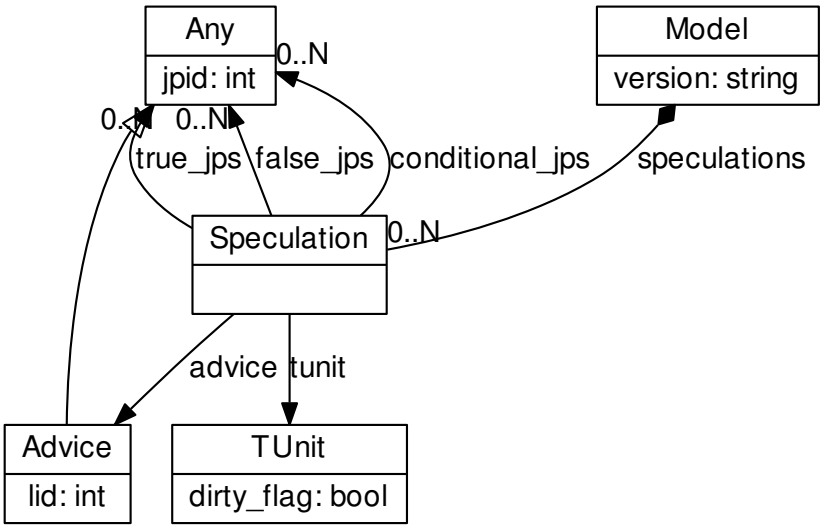


Abbildung 5.1.: Spekulationsklasse im Modell

betreffende Übersetzungseinheit wie in Beispiel 5.2 zu sehen mit dem Attribut `dirty_flag="true"` versieht.

Listing 5.2: Dirty-Flag im persistenten Modell

```
1    <files>
2      <TUnit filename="a.cc" len="42" time="1448831797" dirty_flag="true" id="0"/>
3      <Header in="0 1" filename="ab.ah" len="17" time="1465897216" id="2"/>
4      <Header in="0 1" filename="a.h" len="13" time="1448820280" id="5"/>
5      <Header in="1" filename="b.h" len="15" time="1448820286" id="7"/>
6      <TUnit filename="b.cc" len="42" time="1448817084" id="1"/>
7    </files>
```

Bei der Verarbeitung der mit den Spekulationen über Advices assoziierten Pointcutausdruck-Strings aus dem globalen Joinpoint-Modell wird über einen neu eingeführten On-Demand-Mechanismus sichergestellt, dass jeder dieser Pointcutausdrücke pro Übersetzungseinheit nur einmal neu geparst werden muss.

Um alle Spekulationsobjekte gegen das globale Modell prüfen zu können, musste eine Möglichkeit geschaffen werden, Named Pointcuts aus den von den Spekulationen über die Advices referenzierten Pointcutausdrücken auch im globalen Modell auszuwerten.

Weil die zuvor einzige zur Verarbeitung von Named Pointcuts befähigte Implementierung der abstrakten Klasse PointCutSearcher, Phase1, nur auf den geparsten

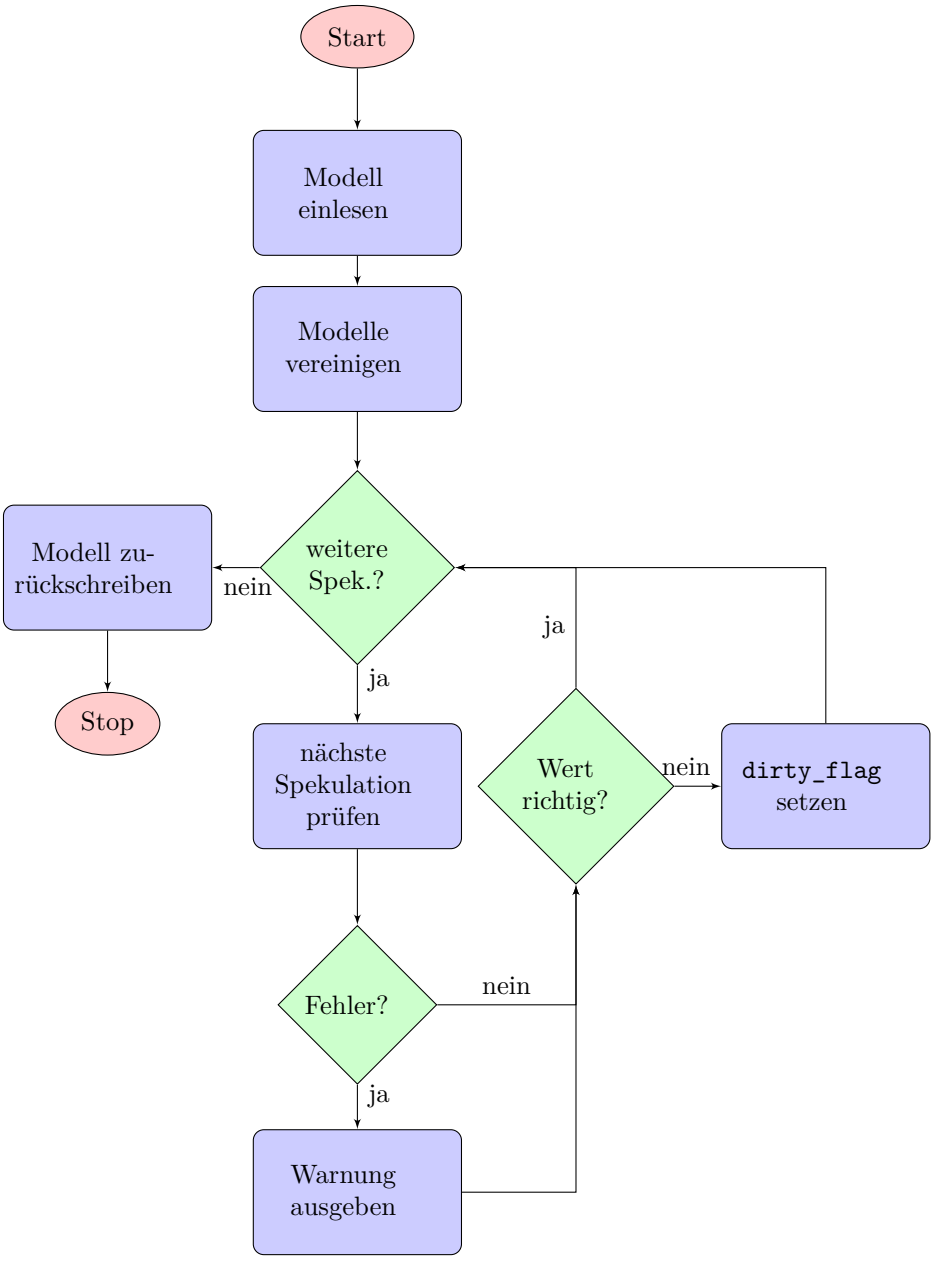


Abbildung 5.2.: Flussdiagramm zur Konsistenzprüfung

Strukturen der aktuellen Übersetzungseinheit arbeitet, wurde die enthaltene Logik in eine neue Klasse mit dem Namen ModelAwarePCSearcher portiert, die auf einem Joinpoint-Modell arbeiten kann.

Bei der Benutzung des ModelAwarePCSearcher muss sicher gestellt sein, dass auf im vorliegenden Pointcutausdruck auf die Named Pointcuts weder mithilfe von Namensraumeinbindung noch über Typaliase zugegriffen wird, weil diese im Joinpoint-Modell noch nicht erfasst werden.

Ist eine dieser Bedingungen nicht gegeben, dann kann der umgebende Pointcutausdruck nicht geparst werden und die Überprüfung der zugehörigen Spekulation wird übersprungen. Der in diesem Fall verwendete Wert basiert folglich nur auf den Informationen aus der aktuellen Übersetzungseinheit. Auf diesen Umstand wird der Anwender mit einer Warnmeldung hingewiesen.

5.1.2. Verbesserung der statischen `cflow`-Analyse

Die bisherige Implementierung der `cflow`-Pointcutfunktion wurde im Wesentlichen um die Mittel zur statischen Kontrollflussanalyse erweitert, die in 4.2.5 beschrieben wurden.

Die `cflow`-Auswertungsfunktion `PCE_Cflow::evaluate` führt nun ausgehend von jedem Element des Parameter-Pointcuts eine Tiefensuche durch, die die Erreichbarkeit eines jeden Joinpoints über Kontrollflusselemente wie Funktionsaufrufe und Funktionsabläufe überprüft. Genau wie der Entwurf entspricht auch die Implementierung weitestgehend der der `cflowto<0>`-Analyse.

Aus Gründen der Abwärtskompatibilität ist die statische Kontrollflussanalyse per Voreinstellung nicht aktiviert. Zum Aktivieren steht ein Kommandozeilenparameter zur Verfügung.

Ein Mechanismus, der über eine Tiefensuche, die im Kontrollflussgraph rückwärts sucht prüfen soll, ob der jeweilige Joinpoint von einem Element des Pointcuts erreicht worden sein muss, wurde vorbereitend eingebaut. Hierfür wurde die Call-Relation zwischen ACM_Function und ACM_Call im nicht- persistenten Modell bidirektional gemacht.

Weil aber in der aktuellen Version der AspectC++-Implementierung keine Möglichkeit besteht, auf einfachem Wege die Menge der Funktionen zu bestimmen, für die Zeiger oder Referenzen gebildet werden, was für die Rückwärtssuche erforderlich ist, kann der Mechanismus inkorrekt arbeiten und muss deswegen separat aktiviert werden.

5.2. Erweiterung des Werkzeugs ACMake

Das Make-Werkzeug acmake überprüft nun neben dem Änderungsdatum einer Quelltextdatei oder einer ihrer Abhängigkeiten auch, ob das persistente Modell für die zugehörige Übersetzungseinheit das XML-Attribut `dirty_flag` auf `true` gesetzt hat. Auch in diesem Fall wird nun neu übersetzt.

Zusätzlich erfolgt die Überprüfung der Quellen und Neuübersetzung jetzt in einer Schleife, weil das `dirty_flag` prinzipiell auch direkt nach dem Übersetzungsvorgang wieder gesetzt sein kann.

Nach einer konfigurierbaren maximalen Durchlaufzahl wird mit einer Warnmeldung abgebrochen. Die Kommandozeilenkonfiguration wurde wie folgt angepasst:

`--max-passes=N` legt fest, wie viele zusätzliche Übersetzungsdurchläufe ACMake bis zum Abbruch maximal durchführt. `N` ist dabei eine natürliche Zahl oder „0“.

`--cflow` teilt ACMake mit, dass das Projekt von den Kontrollflussanalyse-Pointcuts `cflow` und `cflowto` Gebrauch macht. ACMake übergibt in Folge dessen die notwendigen Parameter an den AspectC++-Compiler.

Um in diesem neuen Einsatzszenario einen fehlerfreien Betrieb zu gewährleisten, wurde die Threadsynchronisation überarbeitet. Sie basiert nun auf den internen Synchronisationsmechanismen von `Queue` [10], insbesondere auf `Queue.task_done()`.

5.3. Erweiterung der AspectC++-Sprache

Die AspectC++-Sprache wurde um eine weitere Pointcutfunktion erweitert.

Dieser Abschnitt beschreibt zunächst die neue Pointcutfunktion `cflowto<n>` allgemein, um anschließend die detaillierte Semantik der einzelnen Spezialisierungen zu erläutern.

5.3.1. `cflowto`-Pointcutfunktion

Als zur `cflow`-Funktion komplementäre Funktion wurde die in Kapitel 6 von [3] beschriebene `cflowto`-Funktion eingebaut. Sie erlaubt eine Selektion der Joinpoints, die von einem Element des Parameter-Pointcuts erreichbar sind.

Zur Realisierung der `cflowto`-Funktion, wie sie in 4.2.1 beschrieben wurde, wurde analog zu PCE_CFlow eine neue Unterklasse von PCE_SingleArg angelegt und der Parser PointCutExprParser in die Lage versetzt, `cflowto`-Ausdrücke in Strings

zu verarbeiten. Der Parameterpointcut wird in PCE_CFlowTo::semantics vorbereitet.

Für die einzelnen Konfidenzwerte wurde die `cflowto`-Funktion wie folgt implementiert:

5.3.2. `cflowto<0>`

Für die Überprüfung der Punkte aus 4.2.2 analog zu den anderen Pointcutfunktionen die Funktion PCE_CFlowTo::evaluate angelegt. Sie führt ausgehend vom übergebenen Joinpoint eine einzelne stapelbasierte Tiefensuche durch, die den Punkten aus 4.2.2 entsprechende Abbruchbedingungen hat.

Das Ergebnis wird in dieser Implementierung immer als spekulativ markiert.

5.3.3. `cflowto<1>`

Für `cflowto<1>` wurde keine separate Implementierung erstellt. Vielmehr wird die Implementierung für `cflowto<0>` mitverwendet und die gemäß 4.2.3 entfallende Abbruchbedingung per Fallunterscheidung ausgelassen.

5.3.4. `cflowto<n>` mit $2 \leq n \leq 100$

Für diesen Fall wurden die beiden trivialen Fälle implementiert.

Ein Joinpoint, der im Parameterpointcut enthalten ist, wird Element des Ergebnispointcuts, ebenso ein Joinpoint, der Ziel eines Call-Joinpoints aus dem Parameterpointcut ist.

5.4. Kommandozeilenparameter

Zur Steuerung der `cflow`-Optimierungen wurden zwei neue Kommandozeilenparameter integriert.

`--cflow_use_optimizations` fordert den Compiler dazu auf, die neu eingeführten `cflow`-Optimierungen für den `false`-Fall zu benutzen. Hiervon sollte abgesehen werden, wenn nicht garantiert werden kann, dass der Ursprung eines Funktionsaufrufs innerhalb des aktuellen Projekts liegt, so zum Beispiel bei Bibliotheken.

`--cflow_explicit_calls_only` kann benutzt werden, um dem Compiler manuell die Abwesenheit von Funktionsaufrufen via Zeiger oder Referenz zu ga-

rantieren. Dies aktiviert die im letzten Absatz von 5.1.2 beschriebene Rückwärtssuche im Kontrollflussgraph.

Darüber hinaus wurde die kommandozeilenbasierte Auswertung von Pointcutausdrücken via `-x` beziehungsweise `--expression` auf den neuen Pointcut-Searcher ModelAwarePCSearcher umgestellt. Es ist somit möglich, mit denselben Einschränkungen, die auch für die Spekulationsprüfung gelten Named Pointcuts per Kommandozeilenbefehl auszuwerten. Der Parameter `--cflow_use_optimizations` kann auch in Kombination mit `ac++ -x` benutzt werden.

6. Bewertung

In diesem Kapitel wird untersucht, inwiefern die zuvor genannten Änderungen die Benutzbarkeit und Ausführungsgeschwindigkeit des Compilers selbst und die Korrektheit und Effizienz der erzeugten Programme beeinflussen.

6.1. Korrektheit

Die vorgenommenen Änderungen an der Implementierung haben die Ergebnisse der im AspectC++-Repository enthaltenen automatisierten Regressionstests nicht verändert. Lediglich deren Laufzeit hat sich von 1m 52,3s auf 1m 53,6s erhöht.

Für die Pointcutfunktionen `base`, `cflow` und `cflowto` wurden Beispielprogramme erstellt, mit denen die Plausibilität der korrekten Arbeitsweise des Compilers in Kombination mit ACMake geprüft wurde.

Die Beispielprogramme bestehen jeweils aus mehreren Quelltextdateien, damit mit ihnen der Spekulationsmechanismus überprüft werden kann. Die Funktion der Pointcutfunktionen innerhalb der gleichen Übersetzungseinheit wurde nicht separat überprüft, weil dieser Fall für `base` und `cflow` bereits durch die Regressionstests abgedeckt ist und die gegenwärtige `cflowto`-Implementierung den Spekulationsmechanismus unabhängig von der Anzahl der Übersetzungseinheiten immer benutzt.

Als Ergebnis kann festgehalten werden, dass nach derzeitigem Erkenntnisstand die Auswertung der Pointcutfunktionen `base` und `cflowto` einwandfrei mit Spekulationsverwaltung und ACMake zusammenarbeitet, während die Benutzung der statischen `cflow`-Analyse zunächst auf das persistente Modell beschränkt bleibt. Details hierzu können dem folgenden Unterkapitel entnommen werden.

Eine über den Inhalt des folgenden Unterkapitels hinausgehende Verifikation steht noch aus.

6.2. Benutzbarkeit der statischen Analyse

Zur Bewertung der Benutzbarkeit der neu geschaffenen Möglichkeiten zur statischen globalen Programmanalyse wird im Folgenden zwischen zwei Einsatzszenarien unterschieden. Bei diesen handelt es sich um die spekulative globale Programm-

analyse während des Übersetzungsvorgangs und die nicht-spekulative Auswertung eines vollständigen persistenten Joinpoint-Modells nach vollendeter Übersetzung beziehungsweise zwischen zwei Übersetzungsdurchläufen des Projekts.

6.2.1. Globale Programmanalyse durch spekulatives Weben

In diesem Abschnitt wird in erster Linie bewertet, inwiefern es gelungen ist, den in 4.1 beschriebenen Spekulationsmechanismus bis zur Funktionsfähigkeit zu entwickeln. Dazu wird für jede betroffene Pointcutfunktion einzeln die Korrektheit des Kompilats untersucht.

6.2.1.1. Verwendung der `base`-Funktion mit ACMake

Bei `base` umfasst dies separate Beispielprogramme für Code- und Introduction-Advices. Diese beiden Programme zur Demonstration der Zusammenarbeit zwischen der `base`-Pointcutfunktion, dem Spekulationsmechanismus und ACMake liegen in Anhang B.1 und B.2 dieser Arbeit bei.

ACMake und Compiler arbeiten korrekt zusammen. Für das erste Beispiel läuft der Vorgang wie folgt ab:

Dem Makefile folgend lässt ACMake dabei zunächst die Datei `a.cc` übersetzen. Der Pointcutausdruck im Advice wird für Klasse A während der Übersetzung auf Basis lokalen Wissens zu `false` ausgewertet. Dies wird als Spekulation festgehalten. Aus dem bislang verfügbaren Wissen wird das persistente Modell erzeugt. Als nächstes lässt ACMake `b.cc` übersetzen. Der Advice wird für die Klasse B ebenfalls zu `false` ausgewertet. Noch während der Übersetzung von `b.cc` werden die Spekulationen aus dem persistenten Modell mit dem nun verfügbaren kombinierten Wissen aus `a.cc` und `b.cc` überprüft. Weil nun bekannt ist, das A Basisklasse von B ist, wird die oben genannte Spekulation auf `true` korrigiert und die Übersetzungseinheit `a.cc` als „dirty“ markiert. Nachdem ACMake auch `main.cc` übersetzt hat, prüft es das persistente Modell auf als „dirty“ markierte Übersetzungseinheiten und übersetzt diese erneut. In diesem Fall `a.cc`. In diesem Übersetzungsvorgang wird das auf Basis lokalen Wissens berechnete Ergebnis durch den Wert des passenden Spekulationseintrags aus dem Modell überschrieben. ACMake findet schließlich keine markierten Übersetzungseinheiten mehr und beendet sich regulär.

Das Ergebnis ist korrekt. Auf die detaillierte Beschreibung weiterer Übersetzungsvorgänge wird an dieser Stelle verzichtet. Auch für das zweite Beispiel ist das Ergebnis korrekt.

6.2.1.2. Verwendung der `cflowto`-Funktion mit ACMake

Ein an das Beispiel aus Abbildung 4.2 angelehntes Programm wurde zur Verifikation der in 5.3.1 beschriebenen `cflowto`-Implementierung in Bezug auf die in 4.2.2 und 4.2.3 beschriebene Semantik eingesetzt. Dabei konnten weder beim Zusammenspiel zwischen Compiler und ACMake noch bei der Ausführung des Kompilats unregelmäßigkeiten festgestellt werden.

Das Ergebnis ist korrekt. Dieses Beispiel zeigt, dass Spekulationen auch korrekt von `true` auf `false` geändert werden können.

Eine Momentaufnahme des verwendeten Programms liegt in Anhang B.3 dieser Arbeit bei.

6.2.1.3. Verwendung der `cflow`-Funktion mit ACMake

Bei den Tests der `cflow`-Analyse in Kombination mit dem Spekulationsmechanismus ist aufgefallen, das aus bisher ungeklärten Gründen die im persistenten Modell festgehaltenen spekulativen Ergebnisse mit dem Wert `conditional` entweder nicht korrekt übernommen werden oder die Konsistenzprüfung fehlschlägt. Dies hat neben potenziell inkorrekten Ergebnissen zur Folge, dass die gleiche Übersetzungseinheit laufend neu mit einem `dirty_flag` versehen wird, was wiederum ACMake nach Abarbeitung der Maximalzahl erneuter Übersetzungsdurchläufe abbrechen lässt.

Trotzt dieses Umstandes kann mit dem Verfahren aus dem folgenden Unterkapitel die, wie dort beschrieben ist, prinzipiell funktionierende statische `cflow`-Analyse dennoch zur Codeoptimierung genutzt werden.

6.2.2. Globale Programmanalyse auf dem persistenten Modell

Neben der Auswertung von Pointcutausdrücken während des Übersetzungsvorgangs besteht die Möglichkeit, mittels des Kommandozeilenparameters `-x` beziehungsweise `--expression` des Aspektwebers `ac++` einen Pointcutausdruck auf einem persistenten Joinpoint-Modell auszuwerten.

Da dieses persistente Modell nach einem vollständigen Übersetzungsdurchlauf des Projektes mit Wissen über das gesamte Projekt gefüllt ist, eignet sich auch dieser Weg prinzipiell zur globalen Programmanalyse.

Durch die im Rahmen dieser Arbeit entstandenen Funktionen zur statischen Kontrollflussanalyse wird dieser Einsatzzweck von `ac++ -x` praxistauglich.

6.2.2.1. Erzeugung von Hilfsausdrücken aus dem persistenten Modell

Die kommandozeilenbasierte Auswertung von Pointcutausdrücken kann dazu verwendet werden, Optimierungen auf der Basis statischer Codeanalyse ohne Spekulationsmechanismus und ohne spezialisiertes Make-Werkzeug zu betreiben, weil prinzipiell nichts dagegen spricht, die Ausgabe von `ac++ -x` mittels einfacher Text-Nachbearbeitung in einen Pointcutausdruck umzuwandeln.

Aus dem persistenten Modell des in Anhang B.3 beiliegenden Codebeispiels können zum Beispiel mit dem folgenden Kommandozeilenbefehl die relevanten Joinpoints bestimmt werden:

Listing 6.1: `cflow`-Auswertung mit `ac++ -x`

```
1 > ac++ -x 'cflow(execution("void c()")) && execution("% %(...)")' -r repo.acp \
2 > --cflow_use_optimizations
3 abcd.cc:14:     Execution "void c()" ; speculative
4 ef.cc:8:        Execution "void f()" ; condition: cflow[0] ; speculative
```

Wird diese Ausgabe mit dem Stream-Editor `sed` nachbearbeitet, kann sie während eines weiteren Übersetzungsdurchlaufs als Pointcutausdruck dienen.

Listing 6.2: Kommandozeile zur Erzeugung eines Hilfsausdrucks

```
1 > ac++ -x 'cflow(execution("void c()")) && execution("% %(...)")' -r repo.acp \
2 > --cflow_use_optimizations | \
3 > sed -r 's/.*:[[:space:]]*([a-zA-Z]+) (".*").*/\1\1(\2)/' > hilfsausdruck.h
```

Listing 6.3: Inhalt der Datei `hilfsausdruck.h`

```
1 execution("void c()") ||
2 execution("void f()") ||
```

Dieser Nachbearbeitungsschritt ist auch für die Einbindung in ein Makefile kompakt genug. Die so erzeugte Datei `hilfsausdruck.h` kann wie folgt eingebunden werden:

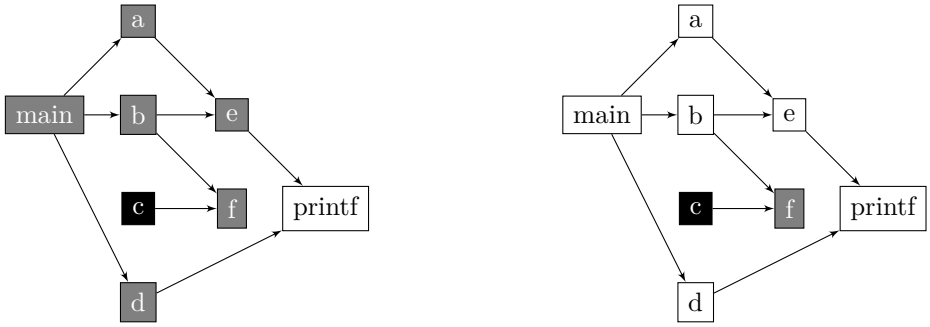
```
1 pointcut constraints() =
2 #include "hilfsausdruck.h"
3   execution("Dies* ist::nur::der(Tuerstopper*)");
4   // ^-- kommt nach dem letzten ||
5 aspect Foo {
6   advice cflow(execution("void c()")) && constraints() : before {
7     // code
8   }
9 };
```

Alternativ kann der erzeugte Hilfsausdruck auf der Konsole ausgegeben und manuell übernommen werden.

6.2.2.2. Statische Kontrollflussanalyse

Wie an den vorangegangenen Listings bereits zu erkennen und in Abbildung 6.1 illustriert ist, reduziert das Verfahren die Anzahl der benötigten Laufzeittests stark.

In Abwesenheit der Analyse des Kontrollflussgraphs müssen die Laufzeittests im vorliegenden Beispiel an jedem Execution-Joinpoint im Modell eingewoben werden. Der durch Kontrollflussanalyse erstellte Hilfsausdruck schränkt dies auf die Funktion `f` ein.



(a) Ohne Optimierungen (b) Mit Optimierungen

Abbildung 6.1.: `cflow`-Optimierung durch Hilfsausdruck

Für `cflowto` ähnelt dieser Ansatz stark dem in 6.6 beschriebenen Ansatz aus [3], der jedoch für die Kontrollflussanalyse auf ein zusätzliches speziell für diesen Zweck geschriebenes Werkzeug zurückgreifen muss.

6.2.2.3. Analyse der Vererbungshierarchie

Ein ähnlicher Ansatz lässt sich auch für die `base`-Pointcutfunktion benutzen. Dies stellt jedoch keine Neuerung dar und wird darum an dieser Stelle nicht weiter behandelt.

6.3. Einschränkungen der entstandenen Lösungen

Durch die im Rahmen dieser Arbeit entstandene Implementierung wurde die Vielfalt möglicher Einsatzszenarien für die `base`-Pointcutfunktion und die statische Kontrollflussanalyse erhöht. Für die neuen Einsatzszenarien ergeben sich aber auch einige Einschränkungen, die im Folgenden detaillierter beschrieben werden.

6.3.1. Named Pointcuts mit `namespace` und `typedef`

Enthält ein Pointcutausdruck Named Pointcuts, dann muss in der gegenwärtigen Implementierung wie in 5.1.1 angemerkt sicher gestellt sein, dass auf diese weder über Namensraumeinbindung noch über Typaliase zugegriffen wird.

Beim Entwurf eines neuen Softwareprojekts kann auf diese Einschränkung Rücksicht genommen werden, dies schränkt jedoch die Nutzung von `typedef`s und Namensraumumbenennungen in Aspektheadern stark ein.

Der denkbare Notbehelf, Spekulationseinträge im persistenten Modell manuell zu bearbeiten, ist nicht praktikabel.

6.3.2. `cflow`-Optimierung beim Übersetzen von Bibliotheken

Die `cflow`-Optimierung basiert auf der Annahme, dass alle Kontrollflüsse ihren Ursprung innerhalb des Projekts haben. Wird also eine Bibliothek übersetzt, also ein Projekt dessen Funktionen von außen aufrufbar sind, dann kann für dieses Projekt keine `cflow`-Optimierung verwendet werden, es sei denn solche Aufrufe können ignoriert werden.

Müssen sie erfasst werden, kann der neu eingebaute Kommandozeilenparameter `--cflow_use_optimizations` nicht benutzt werden. Sämtliche Vorteile der `cflow`-Optimierung sind dann in Bibliotheken nicht nutzbar.

6.4. Einfluss der `cflow`-Optimierungen zur Laufzeit

Die Eignung der neu eingebauten `cflow`-Optimierungen zur Minimierung der Anzahl von Laufzeittests wurde ebenfalls anhand der Softwareprojekte Puma und memcached untersucht. Tabelle 6.1 stellt die Anzahl eingewobener Laufzeittests für Puma mit und ohne Optimierungen gegenüber. Tabelle 6.2 enthält die entsprechende Gegenüberstellung für memcached. Aus der Reduktion der Anzahl an Laufzeittests folgt unmittelbar eine Geschwindigkeitssteigerung. Die Tabellen 6.3 und 6.4 zeigen den Einfluss der Laufzeittests auf die Codegröße. Der Grund dafür, dass sich die resultierende Codegröße bei wegoptimierten Laufzeittests von der Größe des unveränderten Programms unterscheidet könnten die zusätzlichen Kommandozeilenparameter `--data_joinpoints` und `--builtin_operators` sein, die bei der Benutzung von `cflow` erforderlich sind.

Puma	**unverändert**	**`cflow` aus 6.4**
ohne Optimierungen	0	119385
mit Optimierungen	0	0

Tabelle 6.1.: Vergleich der Anzahl eingewobener Laufzeittests (Puma)

memcached	**unverändert**	**`cflow` aus 6.5**
ohne Optimierungen	0	101
mit Optimierungen	0	0

Tabelle 6.2.: Vergleich der Anzahl eingewobener Laufzeittests (memcached)

Puma	**unverändert**	**`cflow` aus 6.4**
ohne `cflow`-Optimierungen	154729512	165799954
mit `cflow`-Optimierungen	154729512	154753742

Tabelle 6.3.: Vergleich der resultierenden Codegröße (Puma)

memcached	**unverändert**	**`cflow` aus 6.5**
ohne `cflow`-Optimierungen	1117064	1187848
mit `cflow`-Optimierungen	1117064	1086088

Tabelle 6.4.: Vergleich der resultierenden Codegröße (memcached)

Listing 6.4: `cflow`-Advice, der nirgendwo wirkt (Puma)

```
1 aspect EmptyCflow {
2   advice cflow(execution("void␣funktionDieNichtExistiert()"))
3       && execution("%␣Puma::CT_%::%(...)") : after() {
4     puts("Dies␣sollte␣nie␣ausgegeben␣werden!\n");
5   }
6 };
```

Listing 6.5: `cflow`-Advice, der nirgendwo wirkt (memcached)

```
1 aspect EmptyCflow {
2   advice cflow(execution("void␣funktionDieNichtExistiert()"))
3       && execution("%␣Memcached::%(...)") : after() {
4     puts("Dies␣sollte␣nie␣ausgegeben␣werden!\n");
5   }
6 };
```

6.5. Ausschöpfung des Optimierungspotenzials

Aufgrund von nicht erfüllten Anforderungen der Optimierungstechniken an die bestehende Implementierung zum einen und zum anderen durch pragmatische Designentscheidungen konnte das theoretische Optimierungspotenzial an manchen Stellen nicht voll ausgeschöpft werden.

6.5.1. Spekulationen mit mehreren `cflow`s in einem Ausdruck

Wie zuvor erwähnt wurde im Rahmen dieser Arbeit die Designentscheidung getroffen, die Granularität der Buchführung über spekulative Werte auf ganze Pointcutausdrücke zu begrenzen, um den Einfluss des Spekulationsmechanismus auf die Größe des Modells gering zu halten.

Lässt sich ein Pointcutausdruck für einen Joinpoint zu `true` oder `false` auswerten, ist dies unproblematisch. Ebenso, wenn er zu `conditional` ausgewertet wird, vorausgesetzt, der Pointcutausdruck enthält nur einen `cflow`-Ausdruck.

Verknüpft ein Pointcutausdruck aber mehrere `cflow`-Ausdrücke, dann können die `cflow`-Laufzeittests nur entweder für alle oder für keinem dieser `cflow`-Ausdrücke herausoptimiert werden, weil ein Spekulationsobjekt pro Joinpoint nur einen Wert speichert.

6.5.2. Problematik `cflow` + Funktionszeiger

Wie in Kapitel 5.1.2 bereits erwähnt wurde, scheitert ein Teil der `cflow`-Optimierung an dem Umstand, dass nur ermittelt werden kann, *von wo aus* per Zeiger oder Referenz eine Funktion aufgerufen wird, nich aber *welche* Funktionen jemals per Zeiger oder Referenz aufgerufen werden.

Aus diesem Grund kann die in `cflow` verbaute statische Kontrollflussanalyse nur untersuchen, ob ein Joinpoint garantiert nicht im dynamischen Kontrollfluss des Parameterpointcuts liegt, nicht aber, ob er garantiert im dynamischen Kontrollfluss des Parameterpointcuts liegt, weil letzteres von eben dieser zweiten Information über Funktionen abhängt.

Vereinfacht gesprochen kann also der `false`-Fall, nicht aber der `true`-Fall der Auswertung des `cflow`-Ausdrucks optimiert werden.

Weil der `true`-Fall jedoch der ist, in dem Advice-Code eingewoben wird, fällt die Anwesenheit eines Laufzeittests an dieser Stelle weniger schwer ins Gewicht, als im `false`-Fall, wo der Laufzeittest als unerwünschte und unerwartete Nebenwirkung Code vergrößern und verlangsamen könnte, der mit dem `cflow`-Ausdruck in keinerlei Verbindung steht.

6.5.3. Präzision und Benutzung von `is_open`

Sind im Pointcutausdruck Platzhalter enthalten, gibt die entstandene `is_open`-Implementierung immer `true` zurück. Steht der Platzhalter an Stelle eines Elements einer bekannten Klasse, könnte `is_open` prinzipiell für jedes mögliche Element der Klasse berechnet werden. Lautet das Ergebnis in allen Fällen `false`, dann könnte dies als `is_open`-Ergebnis für den ursprünglichen Ausdruck übernommen werden.

6.6. Vergleich mit bisherigen Lösungen

In Quelle [3] wird ein Verfahren beschrieben, das eine der neuen `cflowto`-Funktion ähnliche Funktionalität durch ein Zusatzprogramm zur verfügung stellt.

Dieses Zusatzprogramm liest ein vorbereitetes persistentes Modell des gesamten Projekts in XML-Form ein und bestimmt die Menge der von einem Joinpoint aus erreichbaren Funktionen mithilfe von XQuery.

Diese Menge wird als zusätzlicher Aspektheader bereitgestellt, der bei einem weiteren Übersetzungsdurchlauf berücksichtigt werden kann. Das Verfahren ähnelt der Vorgehensweise aus Unterkapitel 6.2.2.1.

Der Übersetzungsprozess der verschiedenen Lösungen lässt sich wie folgt zusammenfassen:

`cflowto` via XQuery bzw. `ac++ -x`

1. Neuübersetzung geänderter Übersetzungseinheiten
2. einmalige Neuberechnung aller cflowto-Pointcuts
3. Neuübersetzung aller Übersetzungseinheiten

internes `cflowto` + ACMake

1. Neuübersetzung geänderter Übersetzungseinheiten mit gleichzeitiger Überprüfung aller cflowto-Pointcuts des Projekts in jeder dieser Übersetzungseinheiten
2. Neuübersetzung der Übersetzungseinheiten mit negativem Überprüfungsergebnis

Es wird also bei der XQuery-Lösung potenziell mehr neu übersetzt, während bei der ACMake-basierten Lösung der `cflowto`-Ausdruck in jeder Übersetzungseinheit neu ausgewertet wird. Welcher Ansatz letztendlich performanter ist, hängt also vom Anwendungsfall ab.

Sollte bei einem Projekt mit einer besonders großen Anzahl an Quelldateien die Mehrfachauswertung zum Problem werden, so kann auf den Ansatz, einen Aspektheader vorzugenerieren, mit dem Ansatz aus 6.2.2.1 weiterhin zurückgegriffen werden.

Für die Auswertung des Ausdrucks
`cflowto<1>(execution("% ...::pthread_mutex_lock(...)"))`
auf dem Joinpoint-Modell von memcached benötigt `ac++` aber, wie in Tabelle 6.5 zu sehen ist, zwischen vier und fünf mal weniger Rechenzeit (bereinigt) als die XQuery-basierte Lösung für eine vergleichbare Analyse braucht.

Lösung	**Prozessor**	**Laufzeit (s)**	**Laufzeit (s·GHz)**
XQuery	AMD Athlon II P320, 2,1GHz	0,36	0,756
ac++ -x	Intel Xeon E5-4650, 2,7GHz	1,3	3,51

Tabelle 6.5.: Laufzeitvergleich mit XQuery-basierter `cflowto`-Lösung

Beim direkten Vergleich beider Lösungen fällt auf, dass die Ergebnispointcuts sich zwar stark ähneln, aber nicht identisch sind. So beinhaltet zum Beispiel der

von der internen Lösung berechnete `cflowto`-Pointcut Konstruktor- und Destruktorfunktionen, während die XQuery-Lösung diese ausspart. Es bleibt zu klären, welche Semantik im praktischen Einsatz nützlicher ist.

Hiervon abgesehen kann dieser Teil der XQuery-Lösung insbesondere aufgrund der zusätzlichen Werkzeuganforderungen und der niedrigeren Geschwindigkeit jetzt als weitestgehend obsolet angesehen werden.

6.7. Geschwindigkeit des Compilers

Der Einbau der genannten Mechanismen hat die Geschwindigkeit exemplarischer Übersetzungsvorgänge erwartungsgemäß negativ beeinflusst.

Tabelle 6.6 zeigt die Zeiten für Puma[1] [16], Tabelle 6.7 die Zeiten für memcached[2]. Unterschieden wird jeweils zwischen dem unveränderten AspectC++-Zweig (trunk) und und dem im Rahmen dieser Arbeit entstandenen Zweig (Speculations) und zwischen der Ausführung mit einem oder zwei Threads.

Es ist anzumerken, dass sich die Puma-Bibliothek in ihrer derzeitigen Form nicht mit ACMake erstellen lässt, weil das zuständige Makefile zwischen Quelldateien unterscheidet, in die Advices eingewoben werden sollen und solchen, in die keine Advices eingewoben werden sollen. ACMake zählt alle Quelldateien zur ersten Kategorie. Möglichkeiten, auch die zweite zu berücksichtigen, werden im Ausblick beschrieben.

An den Werten beider Tabellen lässt sich ablesen, dass auch ohne die explizite Benutzung neuer Funktionalität der Compiler nun mehr Zeit braucht. Der Umstand, dass sich beim Übersetzen mit zwei Threads die Zeitdifferenz zwischen den Entwicklunkszweigen mehr als verdoppelt, während sich die Gesamtzeit beinahe halbiert, könnte auf erhöhten Synchronisierungsaufwand beim Lesen und Schreiben des persistenten Modells hindeuten, das nun zweimal eingelesen wird, aber mindestens beim Schreiben exklusiven Dateizugriff braucht.

[1] PUMA - Source code analysis and transformation library, `http://puma.aspectc.org`
[2] memcached - a distributed memory object caching system, `https://memcached.org`

Puma	**1 Thread**	**2 Threads**	**Beschleunigung**
AspectC++ trunk	10m 31,9s	5m 33,7s	1,89
AspectC++ Speculations	10m 33,0s	5m 38,7s	1,87
Differenz	1,1s	5,0s	—

Tabelle 6.6.: Compiler-Geschwindigkeit (Puma)

memcached	**1 Thread**	**2 Threads**	**Beschleunigung**
AspectC++ trunk	1m 5,0s	0m 32,6s	1,99
AspectC++ Speculations	1m 5,4s	0m 33,5s	1,95
Differenz	0,4s	0,9s	—

Tabelle 6.7.: Compiler-Geschwindigkeit (memcached)

7. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit ist es gelungen zu zeigen, dass ein spekulative Codeerzeugung dokumentierender Compiler in Kombination mit einem angepassten Make-Werkzeug dazu geeignet ist, das Problem der bei der Verarbeitung jeder einzelnen Übersetzungseinheit begrenzten Wissensbasis zu lösen.

Dem Anwender wurde die Möglichkeit gegeben, mittels einer einfachen Abfrage aus einem persistenten Joinpoint-Modell Informationen über sowohl den eingehenden als auch den ausgehenden Kontrollfluss eines Joinpoints zu gewinnen.

Es wurde gezeigt, dass es auf Basis dieser Informationen möglich ist, durch Einsparung von Laufzeittests die Effizienz der Kompilate zu steigern.

An dieser Stelle wird noch einmal zusammengefasst, welche neuen Möglichkeiten sich hierdurch für die Arbeit mit AspectC++ ergeben und abschließend ein Ausblick auf mögliche zukünftige Erweiterungen und Verbesserungen gegeben.

7.1. Zusammenfassung

Als Ergebnis dieser Arbeit wurden sowohl dem AspectC++-Compiler, als auch dem AspectC++-Anwender neue Möglichkeiten eröffnet. Diese untergliedern sich in die drei Teilbereiche *Wissensrevision*, *statische Optimierung* und Analyse der *Zukunft des Kontrollflusses*, die alle dem Bereich der globalen Programmanalyse entstammen.

7.1.1. Wissensrevision

Durch die neue Befähigung des Compilers und des Make-Werkzeugs, Annahmen zu revidieren, die auf Basis unvollständigen Wissens über die Programmstruktur getroffen wurden, wird der Anwender in die Lage versetzt, bisher problematische Konstrukte, wie Advices auf der Basis von `base`-Pointcuts bei Verwendung von ACMake als Make-Werkzeug regulär zu benutzen, ohne die Korrektheit des Kompilats separat sicher stellen zu müssen. Gegebenenfalls auftretende Inkonsistenzen werden vom Compiler selbst erkannt und anschließend durch das Zusammenspiel von ACMake und Compiler eigenständig korrigiert. Bekannte Einschränkungen des

Spekulationsmechanismus bleiben aufgrund entsprechender Warnmeldungen nicht unbemerkt.

7.1.2. Statische Optimierung

Die `cflow`-Pointcutfunktion war in der Vergangenheit aufgrund ihres Laufzeitoverheads insbesondere für die Anwendung in ressourcenbeschränkten Umgebungen oder geschwindigkeitskritischen Situationen ungeeignet. Durch die Verlagerung eines Großteils der Kontrollflussanalyse in den Compiler kann ohne zusätzliche Spezialwerkzeuge dieser Laufzeitoverhead reduziert werden und der Einsatz in derartigen Umgebungen und Situationen wird praktikabel.

7.1.3. Zukunft des Kontrollflusses

Durch die `cflowto`-Funktionen besteht nun erstmals die Möglichkeit, Advices direkt vom zukünftigen Verlauf des Kontrollflusses abhängig zu machen.

Der Anwender kann zum Beispiel mit einem einfachen Pointcutausdruck alle Funktionen auswählen, bei deren Aufruf eventuell eine Ausgabe getätigt wird.

7.2. Ausblick

Es gibt einige Punkte, an denen zukünftige Arbeit auf die Ergebnisse dieser Arbeit aufbauen kann. Zum einen kann mit Modell- und Schnittstellenerweiterungen die Effizienz der bestehenden Optimierungen erhöht werden, zum anderen kann das Konzept der globalen Programmanalyse ebenfalls dadurch und durch den Einbau neuer Funktionalität konsequent erweitert werden.

7.2.1. Vervollständigung des Spekulationsmechanismus

Wie in Kapitel 6 erwähnt wurde, arbeitet der Spekulationsmechanismus noch nicht fehlerfrei mit `conditional`-Spekulationseinträgen zusammen. Der nächste logische Schritt wäre also, ein fehlerfreies Funktionieren zu gewährleisten, da nur dann eine Benutzung der `cflow`-Optimierungen im Übersetzungsvorgang selbst möglich ist.

7.2.2. Alias-Behandlung im Joinpoint-Modell

Wie in 6.3 erwähnt, kann der Mechanismus zur Überprüfung von Spekulationen Namespace-Einbindungen und typedefs in Named Pointcuts nicht auflösen und muss die betroffenen Spekulationen von der Konsistenzprüfung ausnehmen.

Zukünftige Arbeit in diesem Bereich könnte das Modell um eine Klasse ACM_ClassAlias erweitern, die mit einer Referenz auf ein ACM_Class-Objekt einen typedef modellieren könnte. Eine solche Klasse muss den Gültigkeitsbereich ebenfalls modellieren.

Dies würde erlauben, die in 5.1.1 angesprochene Portierung des PointCutSearcher-Codes aus Phase1 nach ModelAwarePCSearcher zu komplettieren und so die oben genannten Probleme zu lösen.

7.2.3. Zusätzliche `cflow`-Optimierungen

Ein konsequenter nächster Schritt im Bereich der `cflow`-Optimierung wäre, den Compiler um Mittel zu erweitern, die es erlauben, aus dem abstrakten Syntaxbaum die Menge der Funktionen zu extrahieren, auf die an einer beliebigen Stelle im Programm per Zeiger oder Referenz zugegriffen wird.

Dies würde den Compiler in die Lage versetzen, den `cflow`-Laufzeittest nicht nur im Fall garantierter Unerreichbarkeit, sondern auch im Fall garantierter Erreichbarkeit auszulassen. Letztere kann der Compiler gegenwärtig nur ableiten, wenn die Abwesenheit von Funktionszeigern und -referenzen vom Anwender per Kommandozeilenparameter garantiert wird.

Zudem könnte bei virtuellen Funktionen jede bekannte erbende Funktion analysiert werden, anstatt an dieser Stelle von der Erreichbarkeit aller Funktionen auszugehen.

7.2.4. Ausweitung des Spekulationsmechanismus

Die Einschränkung der Optimierungsmöglichkeiten bei mehrfacher Verwendung der `cflow`-Funktion im gleichen Pointcutausdruck ließe sich durch eine Erweiterung aufweichen, die im Spekulationsobjekt Einzelergebnisse für jeden `cflow`-Ausdruck festhält.

Der Spekulationsmechanismus kann auch auf Construction-Advices erweitert werden, die nicht wirken dürfen, wenn eine Aggregatinitialisierung vorliegt. Dies würde eine Parsererweiterung zur Erkennung der Aggregatinitialisierung und eine Modellerweiterung für die Buchführung erfordern.

Der Spekulationsmechanismus selbst sollte um Mittel zur Buchführung über die Benutzung oder Nichtbenutzung von Spekulationseinträgen aus dem persistenten Modell erweitert werden. Auf diese Art müssten nicht mehr benutzte Spekulationseinträge nicht mehr bis zum nächsten Aufruf von `make clean` beziehungsweise `acmake clean` beibehalten werden.

7.2.5. Automatisierte Tests mit ACMake

Für zukünftige Entwicklungen im Bereich der statischen Analyse im AspectC++-Compiler wäre es von Vorteil, ACMake in dessen automatisierte Regressionstests zu integrieren.

7.2.6. ACMake und Weber-Steuerung

Wie der Versuch, die Puma-Bibliothek mit ACMake zu übersetzen gezeigt hat, ist ACMake noch nicht universell einsetzbar, weil dem Aspektweber nicht mitgeteilt werden kann, welche Quelltextdateien vom Webevorgang ausgenommen werden sollen.

Eine entsprechende Liste, wie sie derzeit in den Makefiles von Puma existiert, könnte ACMake beispielsweise per Kommandozeilenparameter oder in einer per Kommandozeilenparameter referenzierten Datei übergeben werden.

Alternativ könnte eine neue Pointcutfunktion geschaffen werden, die es erlaubt, Joinpoints nach Übersetzungseinheit zu filtern.

7.2.7. Vollständiger Kontrollflussgraph im Modell

Für die statische Codeanalyse kann es an verschiedenen Stellen interessant sein, Kontrollstrukturen wie Verzweigungen, Schleifen oder Sprungbefehle mitzumodellieren.

Dem aktuellen Joinpoint-Modell (A.1) kann nur ein eingeschränkter Funktionsaufrufgraph entnommen werden. Während dieser für `cflowto<0>` und dessen triviale Abwandlung `cflowto<1>` ausreichend ist, ist es beispielsweise unmöglich, einen Funktionsaufruf zu garantieren. Dieser könnte nämlich im Rumpf eines `if (false)` stehen.

Zukünftige Arbeit in diesem Bereich könnte das Joinpoint-Modell um Techniken zur Modellierung der Kontrollstrukturen innerhalb von Funktionen erweitern.

7.2.8. Mögliche Semantik für `cflowto<n>` mit $2 \leq n \leq 100$

Mit den oben genannten Modellerweiterungen kann `cflowto<100>` auch für nichttriviale Fälle ausgewertet werden.

Eine mögliche Implementierung könnte ein Pointcut-Element als erreichbar deklarieren, wenn es über *alle* vom zu überprüfenden Joinpoint ausgehenden Pfade im Kontrollflussgraph erreichbar ist. Im Einzelfall bedeutet dies, dass zum Beispiel ein Funktionsaufruf in einer `if`-Anweisung auch im zugerörigen `else`-Zweig stehen muss. Schleifen, deren Rumpf nicht zwingend ausgeführt werden muss, wie

`for` oder `while () {...}`, nicht aber `do {...} while`, müssten ignoriert werden. Jegliche Art von Sprungbefehlen, wie `goto`, `return`, `break` oder `continue` müssten ebenfalls berücksichtigt werden.

Literatur

[1] F. E. Allen. “Control Flow Analysis”. In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, S. 1–19. DOI: 10.1145/800028.808479. URL: http://doi.acm.org/10.1145/800028.808479.

[2] K. Bieling. *ACDT: Ein AspectC++-Plugin für Eclipse 4*. Bachelorarbeit. TU Dortmund. 2014.

[3] C. Borchert und O. Spinczyk. “Hardening an L4 Microkernel Against Soft Errors by Aspect-oriented Programming and Whole-program Analysis”. In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS ’15. Monterey, California: ACM, 2015, S. 1–7. ISBN: 978-1-4503-3942-1. DOI: 10.1145/2818302.2818304. URL: http://doi.acm.org/10.1145/2818302.2818304.

[4] P. Briggs u. a. “WHOPR-Fast and Scalable Whole Program Optimizations in GCC”. In: *Initial Draft* 12 (2007).

[5] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 28. Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[6] ISO. *ISO/IEC 9899:201x Committee Draft N1570 — Programming languages — C*. Geneva, Switzerland, 12. Apr. 2011, 683 (est.) URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[7] G. Kiczales u. a. “Aspect-oriented programming”. In: *European conference on object-oriented programming*. Springer. 1997, S. 220–242.

[8] B. Kramer. *AspectClang: Moving AspectC++’s Weaver to the Clang C++ Front End*. Bachelorarbeit. TU Dortmund. 2013.

[9] T. Æ. Mogensen. *Basics of Compiler Design*. lulu.com, 2010. ISBN: 978-87-993154-0-6. URL: http://www.diku.dk/~torbenm/Basics.

[10] Python Software Foundation. *8.10. Queue – A synchronized queue class – Python 2.7.12 documentation*. 2016. URL: https://docs.python.org/2/library/queue.html (besucht am 12. 07. 2016).

[11] G. Ramalingam. “The Undecidability of Aliasing”. In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sep. 1994), S. 1467–1471. ISSN: 0164-0925. DOI: 10.1145/186025.186041. URL: http://doi.acm.org/10.1145/186025.186041.

[12] O. Spinczyk, A. Gal und W. Schröder-Preikschat. “AspectC++: An Aspect-oriented Extension to the C++ Programming Language”. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT ’02. Sydney, Australia: Australian Computer Society, Inc., 2002, S. 53–60. ISBN: 0-909925-88-7. URL: http://dl.acm.org/citation.cfm?id=564092.564100.

[13] O. Spinczyk und D. Lohmann. “The design and implementation of AspectC++”. In: *Knowledge-Based Systems* 20.7 (2007), S. 636–651.

[14] O. Spinczyk und pure-systems GmbH. *AspectC++ Language Reference*. 2016. URL: http://aspectc.org/doc/ac-languageref.pdf (besucht am 09. 07. 2016).

[15] *The GNU Make Manual*. 0.74. Free Software Foundation, Inc. Mai 2016.

[16] M. Urban, D. Lohmann und O. Spinczyk. “PUMA: An Aspect-Oriented Code Analysis and Manipulation Framework for C and C++”. In: *Transactions on AOSD VIII*. Hrsg. von C. Schwanninger und W. Joosen. Lecture Notes in Computer Science 6580. Springer-Verlag, 2011, S. 141–162. DOI: 10.1007/978-3-642-22031-9_5.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

A. Joinpoint-Modell

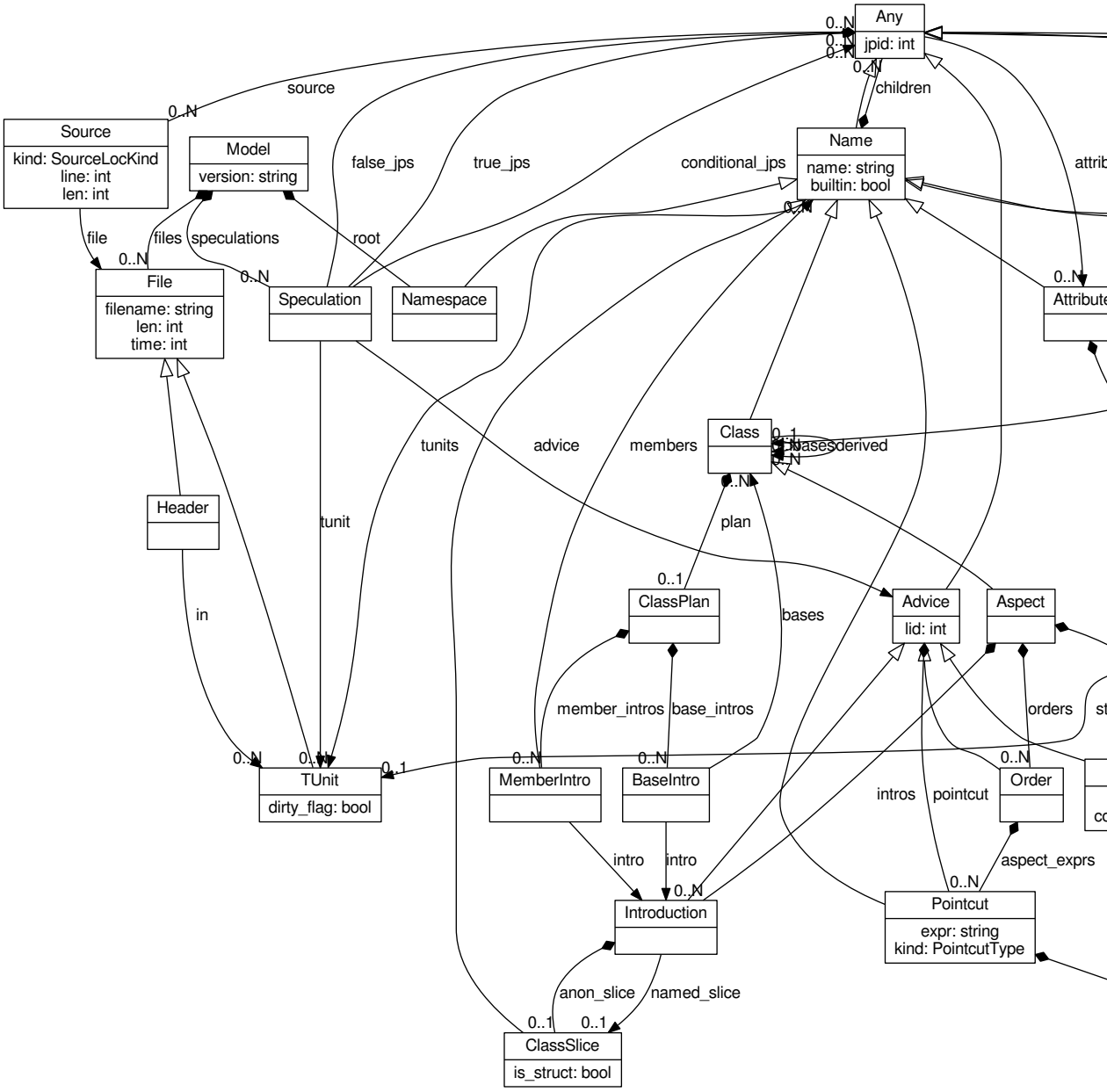
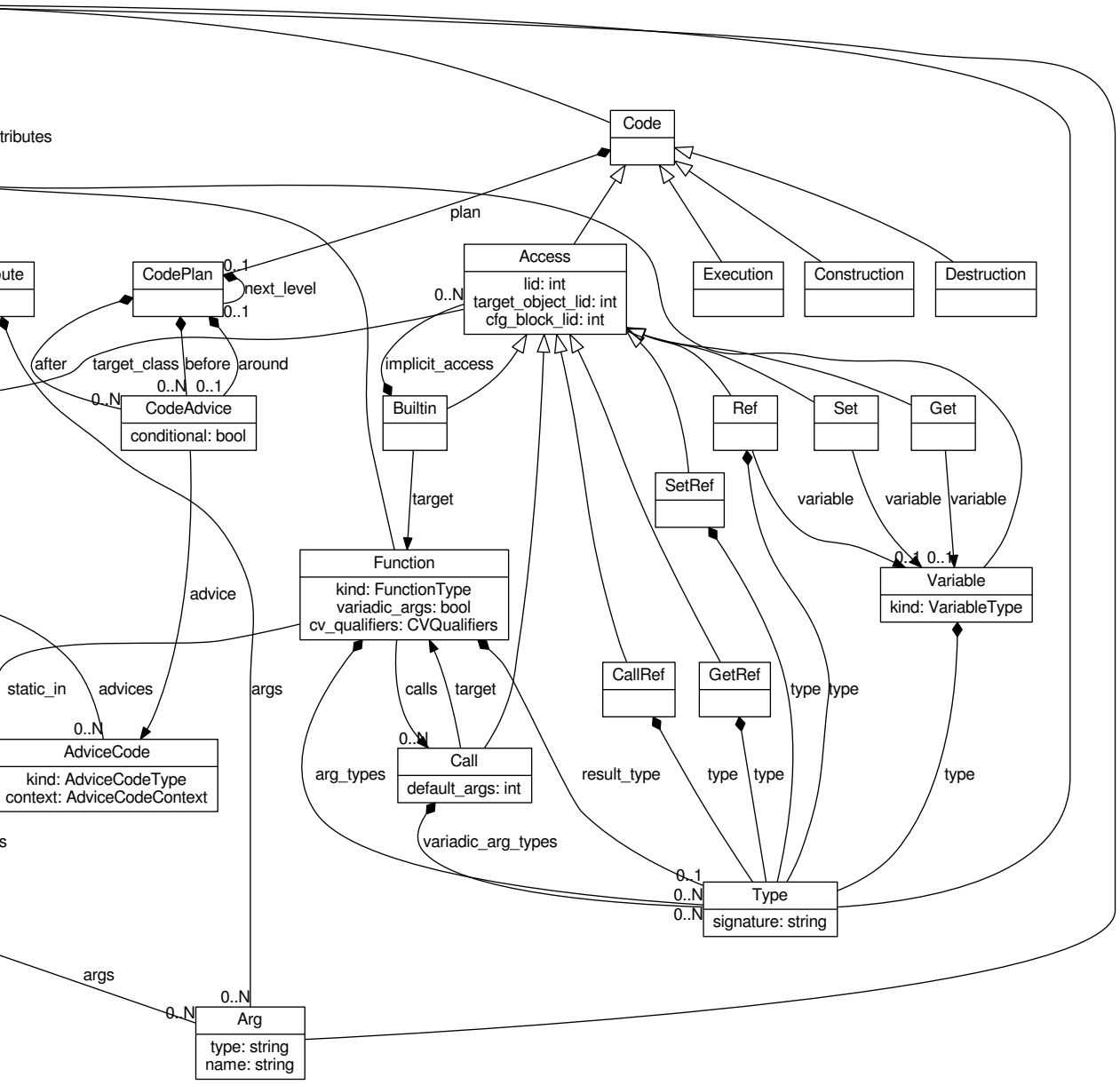


Abbildung A.1.: ACModel-Klassendiagramm


tributes
Code
plan
Access
lid: int
target_object_lid: int
cfg_block_lid: int
Execution
Construction
Destruction
CodePlan
0..1
next_level
0..1
0..N
after
target_class
before
around
0..N
0..1
0..N
CodeAdvice
conditional: bool
implicit_access
Builtin
Ref
Set
Get
SetRef
variable
variable
variable
target
0..1
0..1
Function
kind: FunctionType
variadic_args: bool
cv_qualifiers: CVQualifiers
Variable
kind: VariableType
advice
static_in
advices
args
calls
target
CallRef
GetRef
type
type
0..N
AdviceCode
kind: AdviceCodeType
context: AdviceCodeContext
0..N
Call
default_args: int
arg_types
result_type
type
type
type
variadic_arg_types
0..1
0..N
0..N
Type
signature: string
args
0..N
0..N
Arg
type: string
name: string

B. Vollständige Beispiele

B.1. base

Listing B.1: base a.cc

```
1  #include "a.h"
2  #include <iostream>
3
4  void A::print() {
5    std::cout << "void A::print() executed";
6  }
```

Listing B.2: base a.h

```
1  #ifndef __A_H__
2  #define __A_H__
3
4  class A {
5
6  public:
7
8    void print();
9
10 };
11
12 #endif
```

Listing B.3: base b.cc

```
1  #include "b.h"
2  #include <iostream>
3
4  void B::print() {
5    std::cout << "void B::print() executed";
6  }
```

Listing B.4: base b.h

```
1  #ifndef __B_H__
2  #define __B_H__
```

```
3
4  #include "a.h"
5
6  class B : public A {
7
8  public:
9
10   void print();
11
12 };
13
14 #endif
```

Listing B.5: base main.cc

```
1  #include "a.h"
2  #include "b.h"
3  #include <iostream>
4
5  int main() {
6    A a;
7    B b;
8
9    a.print();
10   std::cout << std::endl;
11
12   b.print();
13   std::cout << std::endl;
14 }
```

Listing B.6: base ab.ah

```
1  #ifndef __AB_AH__
2  #define __AB_AH__
3
4  #include <iostream>
5
6  aspect Ab {
7
8    pointcut b_print() = "void␣B::print()";
9
10   advice execution(base("void␣B::print()")) : after() {
11     std::cout << "␣and␣recognized␣as␣base␣of␣void␣B::print()";
12   }
13
14 };
15
16 #endif
```

Listing B.7: base Makefile

```
1  CXX ?= g++
2
3  PROG:=base
4
5  SOURCES := a.cc b.cc main.cc
6
7  JUNK    := Junk
8  OBJECTS := $(JUNK)/a.o $(JUNK)/b.o $(JUNK)/main.o
9  HEADERS := a.h b.h
10
11 all: $(PROG)
12
13 run: all
14    ./$(PROG)
15
16 $(PROG): $(OBJECTS)
17    @echo Linking $@
18    @$(CXX) $(CXXFLAGS) -o $@ $(OBJECTS) $(LDFLAGS)
19
20 clean:
21    @rm -rf *.o $(JUNK) $(PROG)
22
23 $(JUNK)/%.o : %.cc
24    @mkdir -p $(JUNK)
25    @echo Compiling $<
26    @$(CXX) $(CXXFLAGS) -x c++ -c $< -o $@
27
28 .PHONY: clean all
29
30 # don't remove any intermediate files
31 .SECONDARY:
```

Listing B.8: base repo.acp

```
1  <?xml version="1.0"?>
2  <ac-model version="2.0" ids="15">
3    <files>
4      <TUnit filename="a.cc" len="42" time="1448831797" id="0"/>
5      <Header in="0 1 2" filename="ab.ah" len="17" time="1465897216" id="3"/>
6      <Header in="0 1 2" filename="a.h" len="13" time="1448820280" id="6"/>
7      <Header in="1 2" filename="b.h" len="15" time="1448820286" id="8"/>
8      <TUnit filename="b.cc" len="42" time="1448817084" id="1"/>
9      <TUnit filename="main.cc" len="42" time="1448817238" id="2"/>
10   </files>
11   <root>
12     <Namespace name="::">
13       <children>
14         <Aspect name="Ab">
15           <advices>
16             <AdviceCode kind="1" context="0" lid="0" id="4">
```

```
17              <pointcut>
18                <Pointcut expr="execution(base(&amp;quot;void␣B::print()&amp;quot;))"
                      kind="0" name="%anon">
19                  <source>
20                    <Source kind="0" file="3" line="10" len="1"/>
21                  </source>
22                </Pointcut>
23              </pointcut>
24              <source>
25                <Source kind="0" file="3" line="10" len="3"/>
26              </source>
27            </AdviceCode>
28          </advices>
29          <children>
30            <Pointcut expr="&amp;quot;void␣B::print()&amp;quot;" kind="0" name="b_print">
31              <source>
32                <Source kind="0" file="3" line="8" len="1"/>
33              </source>
34            </Pointcut>
35            <Function kind="8" cv_qualifiers="0" name="~Ab" builtin="true">
36              <children>
37                <Destruction/>
38              </children>
39            </Function>
40            <Function kind="7" cv_qualifiers="0" name="Ab" builtin="true">
41              <children>
42                <Construction/>
43              </children>
44            </Function>
45            <Function kind="7" cv_qualifiers="0" name="Ab" builtin="true">
46              <arg_types>
47                <Type signature="const␣Ab␣&amp;amp;"/>
48              </arg_types>
49              <children>
50                <Construction/>
51              </children>
52            </Function>
53          </children>
54          <source>
55            <Source kind="1" file="3" line="6" len="9"/>
56          </source>
57        </Aspect>
58        <Class name="A" id="7">
59          <children>
60            <Function kind="8" cv_qualifiers="0" name="~A" builtin="true">
61              <children>
62                <Destruction/>
63              </children>
64            </Function>
65            <Function kind="7" cv_qualifiers="0" name="A" builtin="true">
66              <children>
67                <Construction/>
68              </children>
69            </Function>
70            <Function kind="7" cv_qualifiers="0" name="A" builtin="true">
71              <arg_types>
72                <Type signature="const␣A␣&amp;amp;"/>
73              </arg_types>
74              <children>
75                <Construction/>
```

```
76              </children>
77            </Function>
78            <Function kind="3" cv_qualifiers="0" name="print" id="9">
79              <result_type>
80                <Type signature="void"/>
81              </result_type>
82              <children>
83                <Execution id="14">
84                  <plan>
85                    <CodePlan>
86                      <after>
87                        <CodeAdvice advice="4"/>
88                      </after>
89                    </CodePlan>
90                  </plan>
91                </Execution>
92                <Call target="5" lid="0">
93                  <source>
94                    <Source kind="0" file="0" line="5" len="1"/>
95                  </source>
96                </Call>
97              </children>
98              <source>
99                <Source kind="2" file="6" line="8" len="1"/>
100               <Source kind="1" file="0" line="4" len="3"/>
101             </source>
102           </Function>
103         </children>
104         <source>
105           <Source kind="1" file="6" line="4" len="7"/>
106           <Source kind="2" file="6" line="4" len="1"/>
107         </source>
108       </Class>
109       <Namespace name="std" tunits="0 1 2">
110         <children>
111           <Namespace name="__debug" tunits="0 1 2"/>
112           <Class name="locale" tunits="0 1 2"/>
113           <Class name="ios_base" tunits="0 1 2"/>
114           <Function kind="1" cv_qualifiers="0" name="operator &lt;&lt;
                &lt;std::char_traits&lt;char&gt; &gt;" tunits="0 1" id="5">
115             <result_type>
116               <Type signature="std::basic_ostream&lt;char&gt; &amp;"/>
117             </result_type>
118             <arg_types>
119               <Type signature="std::basic_ostream&lt;char&gt; &amp;"/>
120               <Type signature="const char *"/>
121             </arg_types>
122           </Function>
123           <Class name="basic_ostream&lt;char,char_traits&lt;char&gt; &gt;"
                tunits="2" id="11">
124             <children>
125               <Function kind="7" cv_qualifiers="0" name="basic_ostream"
                    builtin="true">
126                 <arg_types>
127                   <Type signature="const std::basic_ostream&lt;char&gt; &amp;"/>
128                 </arg_types>
129                 <children>
130                   <Construction/>
131                 </children>
132               </Function>
```

```
133               <Function kind="3" cv_qualifiers="0" name="operator␣&lt;&lt;"
                      tunits="2" id="10">
134                 <result_type>
135                   <Type signature="std::basic_ostream&lt;char&gt;␣&amp;"/>
136                 </result_type>
137                 <arg_types>
138                   <Type signature="std::basic_ostream&lt;char&gt;␣
                          &amp;(*?)(std::basic_ostream&lt;char&gt;␣&amp;)"/>
139                 </arg_types>
140               </Function>
141             </children>
142           </Class>
143         </children>
144       </Namespace>
145       <Namespace name="__gnu_cxx" tunits="0␣1␣2"/>
146       <Namespace name="__gnu_debug" tunits="0␣1␣2"/>
147       <Namespace name="__cxxabiv1"/>
148       <Class bases="7" name="B" id="13">
149         <children>
150           <Function kind="8" cv_qualifiers="0" name="~B" builtin="true">
151             <children>
152               <Destruction/>
153             </children>
154           </Function>
155           <Function kind="7" cv_qualifiers="0" name="B" builtin="true">
156             <children>
157               <Construction/>
158             </children>
159           </Function>
160           <Function kind="7" cv_qualifiers="0" name="B" builtin="true">
161             <arg_types>
162               <Type signature="const␣B␣&amp;"/>
163             </arg_types>
164             <children>
165               <Construction/>
166             </children>
167           </Function>
168           <Function kind="3" cv_qualifiers="0" name="print" id="12">
169             <result_type>
170               <Type signature="void"/>
171             </result_type>
172             <children>
173               <Execution/>
174               <Call target="5" lid="0">
175                 <source>
176                   <Source kind="0" file="1" line="5" len="1"/>
177                 </source>
178               </Call>
179             </children>
180             <source>
181               <Source kind="2" file="8" line="10" len="1"/>
182               <Source kind="1" file="1" line="4" len="3"/>
183             </source>
184           </Function>
185         </children>
186         <source>
187           <Source kind="1" file="8" line="6" len="7"/>
188           <Source kind="2" file="8" line="6" len="1"/>
189         </source>
190       </Class>
```

```
191         <Function kind="1" cv_qualifiers="0" name="main">
192           <result_type>
193             <Type signature="int"/>
194           </result_type>
195           <children>
196             <Execution/>
197             <Call target="9" lid="0" target_class="7">
198               <source>
199                 <Source kind="0" file="2" line="9" len="1"/>
200               </source>
201             </Call>
202             <Call target="10" lid="1" target_class="11">
203               <source>
204                 <Source kind="0" file="2" line="10" len="1"/>
205               </source>
206             </Call>
207             <Call target="12" lid="2" target_class="13">
208               <source>
209                 <Source kind="0" file="2" line="12" len="1"/>
210               </source>
211             </Call>
212             <Call target="10" lid="3" target_class="11">
213               <source>
214                 <Source kind="0" file="2" line="13" len="1"/>
215               </source>
216             </Call>
217           </children>
218           <source>
219             <Source kind="1" file="2" line="5" len="10"/>
220           </source>
221         </Function>
222       </children>
223     </Namespace>
224   </root>
225   <speculations>
226     <Speculation advice="4" tunit="0" true_jps="14"/>
227   </speculations>
228 </ac-model>
```

B.2. base + introduction

Listing B.9: base + introduction a.cc

```
1 #include "a.h"
```

Listing B.10: base + introduction a.h

```
1 #ifndef __a_h__
2 #define __a_h__
3
4 class A {
5 public:
6   int a;
7 };
8
```

```
9 #endif
```

Listing B.11: base + introduction b.cc

```
1 #include "b.h"
```

Listing B.12: base + introduction b.h

```
1 #ifndef __b_h__
2 #define __b_h__
3
4 #include "a.h"
5
6 class B : public A {
7
8 };
9
10 #endif
```

Listing B.13: base + introduction main.cc

```
1 #include <iostream>
2 #include "a.h"
3
4 using namespace std;
5
6 class C {
7 public:
8   int a;
9 };
10
11 int main() {
12   cout << ((sizeof(A) != sizeof(C)) ? "introduction␣works" :
         "introduction␣does␣not␣work") << endl;
13 }
```

Listing B.14: base + introduction aspects.ah

```
1 #ifndef __aspects_ah__
2 #define __aspects_ah__
3
4 aspect MyAspect {
5   advice base("B") : slice class {
6   public:
7     int b;
8   };
9 };
10
11 #endif
```

B.3. cflowto

Listing B.15: cflowto abcd.cc

```
1  #include "abcd.h"
2  #include "ef.h"
3  #include <stdio.h>
4
5  void a() {
6    e();
7  }
8
9  void b() {
10   e();
11   f();
12 }
13
14 void c() {
15   f();
16 }
17
18 void d() {
19   printf("");
20 }
```

Listing B.16: cflowto abcd.h

```
1  #ifndef __abcd_h__
2  #define __abcd_h__
3
4  void a();
5  void b();
6  void c();
7  void d();
8
9  #endif
```

Listing B.17: cflowto ef.cc

```
1  #include "ef.h"
2  #include <stdio.h>
3
4  void e() {
5    printf("");
6  }
7
8  void f() {
9
```

```
10 }
```

Listing B.18: cflowto ef.h

```
1 #ifndef __ef_h__
2 #define __ef_h__
3
4 void e();
5 void f();
6
7 #endif
```

Listing B.19: cflowto main.cc

```
1 #include "abcd.h"
2 #include "ef.h"
3
4 class CallCOutsideMain {
5   int dummy;
6 public:
7   CallCOutsideMain() {
8     c();
9   }
10 };
11
12 CallCOutsideMain callCOutsideMain;
13
14 int main() {
15   a();
16   b();
17   d();
18 }
```

Listing B.20: cflowto aspects.ah

```
1 #ifndef __aspects_ah__
2 #define __aspects_ah__
3
4 #include <iostream>
5
6 aspect CflowToTest {
7
8   advice cflowto<0>(execution("void␣f()")) && execution("%␣%(...)") :
        before() {
9     std::cout << tjp->signature() << "␣liegt␣in␣
          cflowto<0>(execution(\"void␣f()\"))" << std::endl;
10  }
11
```

```
12    advice cflowto<1>(execution("void␣f()")) && execution("%␣%(...)") :
          before() {
13      std::cout << tjp->signature() << "␣liegt␣in␣
            cflowto<1>(execution(\"void␣f()\"))" << std::endl;
14    }
15
16    advice cflowto<100>(execution("void␣f()")) && execution("%␣%(...)")
          : before() {
17      std::cout << tjp->signature() << "␣liegt␣in␣
            cflowto<100>(execution(\"void␣f()\"))" << std::endl;
18    }
19
20  };
21
22  #endif
```

Listing B.21: cflowto Makefile

```
1  CXX ?= g++
2
3  PROG:=cflowto01
4
5  SOURCES := abcd.cc ef.cc main.cc
6
7  JUNK    := Junk
8  OBJECTS := $(JUNK)/abcd.o $(JUNK)/ef.o $(JUNK)/main.o
9  HEADERS := abcd.h ef.h
10
11 all: $(PROG)
12
13 run: all
14   ./$(PROG)
15
16 $(PROG): $(OBJECTS)
17   @echo Linking $@
18   @$(CXX) $(CXXFLAGS) -o $@ $(OBJECTS) $(LDFLAGS)
19
20 clean:
21   @rm -rf *.o $(JUNK) $(PROG)
22
23 $(JUNK)/%.o : %.cc
24   @mkdir -p $(JUNK)
25   @echo Compiling $<
26   @$(CXX) $(CXXFLAGS) -x c++ -c $< -o $@
27
28 .PHONY: clean all
29
30 # don't␣remove␣any␣intermediate␣files
```

```
31  .SECONDARY:
```

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift