technische universität
dortmund

Diplomarbeit

Erweiterung der Programmiersprache AspectC++ um Advice für Datenzugriffe

Markus Schmeing
30. März 2015

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

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

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

Dortmund, den 30. März 2015

Markus Schmeing

Zusammenfassung

Aspekt-orientierte Programmierung (kurz AOP) ist ein modernes Mittel zur Softwareentwicklung. Es stärkt die Modularisierung und ermöglicht die Schaffung leicht wiederverwendbaren Komponenten für querschneidende Belange, in dem es das Einweben der notwendigen Änderungen automatisiert. AspectC++ ist eine auf diesem Konzept aufbauende Spracherweiterung für C/C++. Bisher wurde nur die Manipulation von Funktionen und Klassen unterstützt. Diese Arbeit schafft nun die ersten grundlegenden Joinpoints Get und Set, die eine Reaktion auf Ereignisse in Verbindung mit Variablen ermöglichen.

Hierzu stellt sie den notwendigen Entwicklungsprozess vor, entwickelt die notwendigen Sprachelemente sowie Definitionen und setzt diese im AspectC++-Weber um. Ferner geht sie auf sich daraus ergebene Probleme, insbesondere auf die Alias-Problematik ein. Hier werden erste Lösungsansätze vorgeschlagen und ein Rahmen für weitere Untersuchungen abgesteckt.

Die Arbeit zeigt auch den Nutzen der Erweiterung an Hand von Anwendungsbeispielen auf und bewertet sie im Vergleich mit ähnlichen Ansätzen.

Danksagung

An dieser Stelle möchte ich allen Personen danken, die mich während der Erstellung dieser Arbeit sowie in meinen Studium unterstützt haben.

Zunächst gebührt daher mein Dank den Entwicklern des AspectC++-Projekts. Ohne den von ihnen in mühseliger Arbeit geschaffenen AspectC++-Weber, auf dem ich aufgesetzt habe, wäre diese Arbeit nicht möglich gewesen.

Ganz besonderer Dank gebührt meinen beiden Betreuern Prof. Olaf Spinczyk und Christoph Borchert. Ohne die intensive Betreuung und die fruchtbaren Diskussionen hätte diese Arbeit nicht gelingen können.

Bedanken möchte ich mich auch bei Norbert Staebler, meinem Informatik-Lehrer in der Schule. Er hat mir gezeigt, dass Informatik nicht das Bedienen eines Computers oder das Programmieren einer Anwendung ist, und mich damit ungewöhnlich gut auf das Studium vorbereitet.

Schließlich möchte ich meiner Familie und meinen Freunden für die bedingungslose Unterstützung und den Rückhalt auf meinem Weg danken. Er war lang, hat mich aber hoffentlich ein wenig Weisheit erlangen lassen.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Aspekt-orientierte Programmierung (kurz AOP)[KLM+97] ist ein verhältnismäßig neues Konzept um auf die Herausforderungen moderner Softwareentwicklung zu reagieren. Projekte werden immer umfangreicher und komplexer, müssen aber zunehmend häufiger an neue Entwicklungen angepasst werden. Die Idee hinter AOP ist die Entflechtung von Quellcode um stärkere Modularisierung zu ermöglichen und somit Wartbarkeit und Wiederverwendbarkeit zu stärken.

Ohne AOP und die dazugehörigen Werkzeuge lassen sich häufig Probleme bestimmter Art nicht oder nur sehr schlecht in Modulen kapseln. Ein einfaches Beispiel für diese querschneidenden Belange ist das Akquirieren/Freigeben eines Locks beim Betreten bzw. Verlassen klar umrissener Programmteile (z.B. in Betriebssystemen: Betreten des Schedulers, in Anwendungen mit komplexer Datenhaltung: Anfragen an die Datenbank etc.). Mit klassischen Mitteln wäre hierfür ein Eingriff an jeder Stelle im Quellcode nötig, an der die Grenze zu diesem Bereich überschritten wird. D.h. bei jedem Methodenaufruf der Datenbank, der möglicherweise von außerhalb dieses Programmteils erfolgt, ist eine Überprüfung notwendig. Hier entsteht das Problem, dass dies schwer und aufwendig zu pflegen ist und schnell einzelne Punkte vergessen werden. Weiterhin ergeben sich Probleme bei der Konfigurierbarkeit der Software, denn sobald hier verschiedene Varianten zum Einsatz kommen sollen, multipliziert sich der Aufwand, da an jeder Stelle die Konfiguration berücksichtigt werden muss.

Auf dieses Problem bietet AOP eine Antwort: Mittels Aspekten, welche Sätze von Regeln in Verbindung mit Codefragmenten enthalten, kann bestehender Programmcode manipuliert werden. In dem obigen Beispiel wäre es damit möglich einen Aspekt zu entwickeln, der beim Betreten des Programmteils automatisch den Lockingmechanismus prüft und gegebenenfalls das Lock anfordert, sowie analog dazu beim Verlassen wieder freigibt.

```
1 aspect AutoLock {
2   advice call( "% Database::%(...) && ! within( "Database" ) : around() {
3     lock.acquire();
4     tjp->proceed();
5     lock.release();
6   }
7 }
```

Dieses vereinfachte Beispiel zeigt wie der Programmierer nur an zentraler Stelle die Bedingungen definieren und die Fragmente bereitstellen muss, damit diese anschließend

vom Aspektweber während des Übersetzungsprozesses automatisiert an den definierten Stellen eingefügt werden können. Hierdurch kann eine Trennung der jeweiligen Kernfunktionalität von anderen Belangen erreicht werden, so dass Komponenten leichter modifiziert oder ausgetauscht werden können.

Auf diesem Wege lassen sich viele wiederkehrende Probleme lösen. Einfache Fälle sind hier ein Tracing des Programms zwecks Analyse und Fehlersuche. Aber auch für Nebenläufigkeitsprobleme sowie Sicherheitsmechanismen (die Prüfung von Benutzereingaben ist hier nur der einfachste Fall) lassen sich wiederverwendbare Komponenten entwickeln, die leicht in eine Vielzahl von Programmen integriert werden können. Im Bereich der eingebetteten Systeme spielt die Konfigurierbarkeit von entwickelten Komponenten eine große Rolle. Durch Verwendung einer einzigen adaptiven Produktlinie[LS06] kann der Entwicklungsaufwand reduziert und gleichzeitig die Softwarequalität gesteigert werden.

Insbesondere die leichte und schnelle Integration von neuer Funktionalität stellt einen wichtigen Faktor dar. Ein Beispiel sind hier die von der Arbeitsgruppe entwickelten Fehlertoleranzmechanismen. Diese sollen möglichst ohne manuelle Codeanpassungen in bestehende Software integriert werden können um so deren Ausfallsicherheit zu erhöhen. Hierbei stellt sich nun häufig ein Problem mit den bestehenden Werkzeugen. AspectC++ unterstützt bisher nur die Manipulation von Funktionen bzw. deren Aufrufen sowie von Klassen. Aber für einige Mechanismen wie die General Object Protection[BSS13] sind nicht die Funktionen an sich, sondern die Daten bzw. die Zugriffe auf diese relevant. Das Problem lässt sich teilweise durch Ausweichen auf Funktionsaufrufe sowie Kapselung und verstecken der Daten in Objekten umgehen, dies erfordert aber umfangreiche Eingriffe in den abzusichernden Quellcode.

Hier will diese nun Arbeit ansetzen. Durch die Schaffung von Sprachelementen und Erweiterung des Aspekt-Webers soll die Möglichkeit geschaffen werden auf Datenzugriffe ebenso zu reagieren wie es bereits im Zusammenhang mit Funktionen möglich ist. In AspectJ (eine Aspekt-orientierte Java-Variante) ist dies bereits durch die Pointcut-Funktionen “get()” und “set()” sowie die dazugehörigen Joinpoints seit einiger Zeit möglich[KHH+01, Abschnitte 3.1, 3.2]. Auch verspricht eine Daten-gewahre Erweiterung von AspectC++ weitere Möglichkeiten. So wären Erweiterungen denkbar, die bestehende Pointcut-Funktionen um Spezifität auf Basis des Objekts im Gegensatz zur bestehenden alleinigen Selektion auf Basis des Typs ergänzen. Dies würde erlauben eine feingranularere Auswahl der zu modifizierenden Teile des Programms zu treffen und so eine höhere Effizienz zu erreichen.

Bisher ist in dieser Richtung nicht gearbeitet worden, denn bestimmte Spracheigenschaften machen eine Lösung deutlich schwieriger. Zuvorderst ist hier das Alias- bzw. Points-to-Problem zu nennen, das auch aus dem Bereich der Codeanalyse und Optimierung bekannt ist. Da C und C++ Pointer- und Referenztypen unterstützen, ist häufig unbekannt auf welches Ziel ein Zugriff erfolgt. Zur Laufzeit des Programms ist dies natürlich in jedem konkreten Fall bestimmt, doch zur Übersetzungszeit, zu der die Codemanipulationen vom Weber durchgeführt werden müssen, ist dies in der Regel nur schwer oder gar nicht bestimmbar. AspectJ hat dieses Problem nicht, bzw. kann sich erlauben es weitestgehend dem Programmierer zu überlassen, da Java keine allgemeinen Referenztypen kennt. In C/C++ sind diese aber integraler Sprachbestandteil und ein

Verzicht auf diese stellt eine sehr große Einschränkung dar.

Ein Ansatz in dieser Richtung muss also auch diese besonderen Herausforderungen der Sprache berücksichtigen. Doch gerade der Einsatz im Bereich der eingebetteten Systeme erhöht diese nochmals, da hier besondere Anforderungen an den entstehenden Overhead bestehen und beispielsweise Lösungen mit umfangreichen Laufzeitkomponenten problematisch sind. Die Idee hinter dieser Arbeit basiert daher auf einer inkrementellen hybriden Lösungsstrategie. Beginnend mit der Erweiterung des Webers um grundlegende Fähigkeiten ähnlich denen in AspectJ, gefolgt von weiteren Elementen, sollte es möglich sein die Menge der abgedeckten Anwendungsfälle stückweise zu erhöhen. Es soll dabei zunächst bewusst nicht vollständig auf Laufzeitstrukturen verzichtet werden, denn diese erhöhen zwar den Overhead im Zielprogramm, aber der Verwender des Weber kann diesen erheblich besser gegen den Verzicht auf die Funktionalität abwägen.

Dieser Ansatz bietet die Möglichkeit bereits erste Ergebnisse zu erzielen und daraus gleichzeitig weitere Erkenntnisse über die Anforderungen an die Funktionalität des Webers sowie an die Sprache selbst zu gewinnen. Auch wird hierdurch möglich die Bedeutung der Alias-Problematik sowie Konzepte zu deren Lösung zu untersuchen. Es besteht die Hoffnung, dass am Ende dieses Prozesses basierend auf den gewonnenen Erkenntnissen und aktuellen Codeanalyseverfahren eine Möglichkeit gefunden wird, für eine große Zahl an Anwendungsfällen eine Verwendbarkeit zu erreichen, die gleichzeitig mit minimalem Overhead im Zielprogramm auskommt, und somit auch für eingebettete Systeme mit begrenztem Ressourcen nutzbar ist.

1.2. Ziele

Das Langzeitziel dieses Prozesses: umfangreiche Möglichkeiten in Verbindung mit Daten basierenden Advice zu schaffen und dabei minimalen Overhead zu erreichen, liegt klar außerhalb des im Rahmen dieser Arbeit erreichbaren. Es besteht nicht nur eine Vielzahl an Möglichkeiten: Von einfachen Joinpoints für Datenzugriffe, über Objekt-gewahre Argumenten-Filter bis hin zu Sprachelementen, die komplexe Datenflüsse berücksichtigen können. Sondern auch die bekannten (Alias-Problematik) oder noch zu erwartenden Probleme bei der Umsetzung sowie der Umfang statischer Codeanalyse lassen eine umfängliche Bearbeitung nicht zu. Daher müssen die Ziele dieser Arbeit entsprechend gewählt werden. Im Kern sind dies 3 Bereiche:

Zu Beginn muss eine zumindest grobe konzeptionelle Planung des Entwicklungsprozesses erfolgen, die ein inkrementelles Vorgehen berücksichtigt und somit die Nutzung erster erzielter Zwischenergebnisse ermöglicht, obwohl der Gesamtprozess noch nicht abgeschlossen ist. Insbesondere sind diejenigen Erweiterungen zu identifizieren, die das Potential haben gleichzeitig sowohl erste Anwendungsfälle abzudecken als auch ermöglichen eine solide Basis für die weitere Entwicklung zu schaffen, während sie klein genug sind um eine kurzfristige Umsetzung zu ermöglichen. Entscheidend bei der Priorisierung der Teile ist hier nicht nur das Abhängigkeitsverhältnis untereinander, sondern auch die Möglichkeit Abschätzungen für die weitere Entwicklung zu treffen und so den Entwicklungsprozess selbst zu verfeinern.

In einem zweiten Schritt kann dann eine Entwicklung grundlegender Sprachelemente und Weberkomponenten erfolgen um den Prozess zu starten und weitere Entwicklungsschritte sowie eine bessere Bewertung der Potentiale zu ermöglichen. Hierbei sind mehrere sich teils widersprechende Rahmenbedingungen zu beachten. Einerseits sollten diese als Proof-of-Concept möglichst viel Bereiche, Probleme und Anwendungsfälle abdecken, andererseits müssen sie bereits fest integriert werden und gerade im Bereich der Sprachelemente hinreichend stabil sein um so eine erste Nutzung zu ermöglichen. Gleichzeitig müssen sie aber modular und flexibel sein um als Basis für weitere Entwicklungen dienen zu können und spätere Erweiterungen und Verbesserungen zu ermöglichen. Eine weitest gehende Berücksichtigung der speziellen Anforderungen bei der Entwicklung für einbettete Systeme als wichtige Zielplattform ist wünschenswert, könnte aber zugunsten der initialen Funktionsfähigkeit zurückgestellt werden um sie in einem späteren Entwicklungsschritt im Rahmen einer Anpassung bzw. Optimierung zu berücksichtigen.

In einem letzten Schritt kann dann eine erste Einschätzung der Potentiale und Möglichkeiten für weitere Schritte erfolgen. Auch ist zu bewerten in wie weit die umgesetzten Teile die Erwartungen bereits erfüllen und bei welchen Komponenten noch Verbesserungsmöglichkeiten bestehen.

Zusammenfassend lassen sich die Ziele also wie folgt beschreiben: Schaffung von Grundlagen sowohl bei Sprachdesign als auch Umsetzung sowie Aufzeigen des Nutzens für erste Anwendungsfälle.

1.3. Überblick

Kapitel 2 beginnt mit der Analyse des Problemfelds und skizziert dann den möglichen Entwicklungsprozess. Auch umreißt es eine Klasse von Anwendungsfällen, die dem folgenden Kapitel 3 als Orientierung bei dem Entwurf der grundlegenden Definitionen für Advice, die Datenzugriffe behandeln, dienen. Das Kapitel erläutert dabei zunächst die Hintergründe der Definitionen, bevor es sich der Umsetzung und den damit verbunden Problemen widmet. Im Kapitel 4 geht es dann um die Alias-Problematik und Ansätze zu ihrer Lösung. Kapitel 5 schließlich bewertet die getroffenen Entscheidung und untersucht die entstehenden Laufzeitkosten empirisch. Den Schluss bildet Kapitel 6 mit einer Zusammenfassung und dem Ausblick auf künftige Möglichkeiten.

Die im Rahmen der Arbeit erfolgten Änderungen am AspectC++ Quellcode sind über das SVN-Repository(Branch: GetSet[1]) des Projekts verfügbar.

[1] `https://svn.aspectc.org/repos/AspectC++/branches/GetSet`

2. Analyse und Planung

Das bisher keine Unterstützung für Advice existiert, die auf Datenzugriffen aufbauen, liegt primär in den Eigenschaften von C++ als Basis von AspectC++ begründet (Vgl. [SL07, insbesondere Abschnitt 4.3.4]). Die Vielfalt der Sprachelemente und Freiheiten, die dem Programmierer zur Verfügung stehen, erhöht den nötigen Aufwand und die zu lösenden Schwierigkeiten beträchtlich. Insbesondere die Alias-Problematik, die sich aus der Existenz von Pointern und Referenzen ergibt, wird als Hindernis angesehen. Aber auch die Überladung von Operatoren[AÖ07, Abschnitt 4.3] und andere Sprachfeatures sind problematisch. Begünstigt wird dies durch Entwicklungsstruktur des AspectC++-Projektes. Auch wenn das Projekt prinzipiell Open-Source ist und damit jedem offensteht, ist die Zahl der permanenten Entwickler gering und ein Großteil der Entwicklung findet in Arbeiten wie diesen oder als Nebeneffekt anderer Arbeiten statt. Eine vollumfängliche Auseinandersetzung und Lösung der erwähnten Probleme erscheint daher in einem absehbaren Zeitraum unwahrscheinlich.

Andererseits zeigt sich immer wieder Bedarf für diese Funktionalität. Weiterhin gibt es beispielsweise in der Aspekt-orientierten Java-Variante AspectJ[KHH+01] bereits Unterstützung für diese. AspectJ profitiert hier von dem Fehlen der problematischen Sprachfeatures und den generellen Möglichkeiten durch die JVM. Java ist aber nur bedingt im Bereich der eingebetteten Systeme einsetzbar. In Rahmen dieser Arbeit soll daher der Versuch unternommen werden diese Lücke zumindest teilweise zu schließen.

Ein erster Versuch[Mag06] wurde von Magnusson schon 2006 unternommen. Basierend auf der Analyse[AÖ07] von Alexandersson & Öhman wurden hier die nötigen Sprachelemente ergänzt. Diese Variante ermöglichte es prototypische Probleme aus dem Bereich der Fehlertoleranz zu lösen, konnte aber keine Lösung für die Alias-Problematik anbieten. Leider sind diese Variante und die dazu gehörige Veröffentlichung erst deutlich später den Verantwortlichen bekannt geworden, so dass sie nicht in den Hauptentwicklungszweig übernommen werden konnte. Heute wäre dies auch nicht mehr möglich, da die aktuelle Version des AspectC++ Weber erheblich von der als Basis verwendeten abweicht. Sie kann daher nicht mehr als Basis, jedoch noch als Orientierung und Vergleichsmaterial dienen. Dabei muss allerdings auf sekundäre Literatur zurückgegriffen werden, da die Arbeit selbst nicht mehr verfügbar ist.

Um einen im Rahmen des AspectC++-Projekts gangbaren Weg der Entwicklung zu erhalten, ist die erste Herausforderung also das gesamte Problemfeld in einzelne möglichst kleine Schritte zu unterteilen. Diese müssen in sich geschlossen sein bzw. dürfen nur von vorhergehenden abhängen. Dabei ist es aber auch erforderlich, dass eine Integration zukünftiger Schritte ohne größere Anpassungen möglich ist, damit diese sich auf die Entwicklung neuer Funktionalität konzentrieren können.

2.1. Die 3 Entwicklungsebenen

Wie in der Einführung angesprochen widersprechen sich einige Anforderungen, die sich aus der Zielsetzung, den Anwendungsfällen und Eigenschaften ihrer Zielplattformen sowie dem gewünschten Entwicklungsprozess ergeben. Daher widmen wir uns zunächst den 3 Bereichen in denen die Entwicklung erfolgt.

2.1.1. Sprachdesign

Das Sprachdesign muss die notwendigen Sprachelemente schaffen um die neuen Anwendungsfälle zu unterstützen. Zu erwarten ist hier die Definition neuer Joinpointarten und die Erweiterung der Pointcutsprache um Advice für diese erstellen zu können. Auch in diesen Bereich fallen Betrachtungen zur Joinpoint-API, die dem Programmierer bei der Entwicklung der Advice zur Verfügung steht.

Da diese Erweiterungen Kern der später sichtbaren Sprache sind, müssen diese eine besondere Stabilität im Sinne des Entwicklungsprozess aufweisen. D.h. die müssen soweit wohldefiniert sein, dass später keine Rücknahme oder Veränderung erfolgen muss. Dies würde starke Auswirkungen auf die Kompatibilität zwischen einzelnen Versionen bewirken. Zu gleich dürfen sie nicht zu starr und eng sein. Sie müssen also die jeweiligen Ziele erfüllen können, dürfen aber späteren Schritten nicht im Wege stehen. Sie bedürfen also einer besonderen Aufmerksamkeit und Sorgfalt, da sie langfristiger Bestandteil der Sprache werden.

2.1.2. Erweiterung des AspectC++ Webers

Bei der Anpassung des Webers zur Unterstützung der neuen Funktionen spielt dagegen Modularität und spätere Erweiterbarkeit eine bedeutende Rolle. Hier können erheblich leichter spätere Änderungen durchgeführt werden, da im allgemeinen nur das Ergebnis sichtbar wird. Ebenso spielen Überlegungen zur Effizienz eine untergeordnete Rolle, da Laufzeit und Speicherverbrauch zur Übersetzungszeit nicht so stark beschränkt sind, wie zur Laufzeit des Zielprogramms.

Kern der Überlegungen in diesem Bereich ist also die Softwarearchitektur. Es gilt also Lösungen zu finden, die die Funktionalität des jeweiligen Schrittes umsetzen, dabei aber auch die Wiederverwendbarkeit und damit die Vereinfachung künftiger Schritte berücksichtigt. Ein wichtiges Prinzip wird hier die Abstraktion von der konkreten Problemlösung hin zu einer Struktur sein, die als Basis für die jeweilige aber auch zukünftige Erweiterungen dienen kann.

2.1.3. Laufzeitkomponenten

Die Laufzeitkomponenten unterliegen besonderen Anforderungen bezüglich ihre Effizienz, da sie im wesentlichen den im Zielprogramm entstehenden Overhead bestimmen. Gerade bei eingebetteten Systemen, die eine wichtige Zielplattform der Sprache darstellen, sind starke Einschränkungen der verfügbaren Ressourcen normal. Da auch klei-

ne Erhöhungen hier Auswirkungen auf die Umsetzbarkeit und die Kosten haben, muss die Vereinfachung der Entwicklung immer besonders genau gegen den gesteigerten Verbrauch an Ressourcen abgewogen werden. Aber auch für andere Zielplattformen muss dies berücksichtigt werden, da steigende Anforderung in diesem Bereich nicht als Steigerung der Produktqualität betrachtet werden können, was immer Ziel der Entscheidung für eine bestimmte Entwurfsmethodik sein sollte.

Aufgrund der geringen Interaktion mit anderen Komponenten der Sprache und Beschränkung des Umfeldes auf jeweils einen einzigen Übersetzungsvorgang des Zielprogramms können jedoch erheblich einfacher auch nachträgliche Änderungen erfolgen. Dies erlaubt einen hohen Grad an Flexibilität bei der Umsetzung und eine hohe Anpassbarkeit an konkrete Bedürfnisse. Somit können hier architektonische Überlegung zu Gunsten der Effizienz zurückgestellt werden.

2.2. Anforderungen der Anwendungsfälle

Einen ersten Ansatzpunkt bietet hier die Analyse[AÖ07] von Alexandersson & Öhman. Dort wurde untersucht in wie weit AspectC++ für Fehlertoleranzmechanismen geeignet ist und unter anderem festgestellt, dass die Reaktion auf Datenzugriffe nicht möglich ist. Dies erweist sich als entscheidender Faktor für die Umsetzbarkeit einiger Algorithmen. Auch im Rahmen der General Object Protection[BSS13] wurde immer wieder festgestellt, dass sich diese fehlende Unterstützung als Schwachstelle erweist. Dies kann zwar durch Anpassung des Zielprogramms umgangen werden, da es hauptsächlich bei Zugriffen auf öffentliche Attribute der jeweiligen Klasse ein Problem darstellt. Die Modifikationen stellen jedoch einen größeren Aufwand dar, sind aber auch wegen der fehlenden Trennung der Komponenten nicht wünschenswert.

Aber auch bei Verwendung von AOP-Konzepten als generellem Entwicklungsparadigma zur Umsetzung von Softwarekomponenten ergeben sich Anwendungsmöglichkeiten für eine AspectC++-Erweiterung, welche den Datenraum berücksichtigen kann. Beispiele hier sind Synchronisation, Locking oder Transaktionsmanagement. Auch diese querschneidenden Belange können von einer Reaktionsmöglichkeit auf Ereignisse während der Lebensdauer einer Variablen profitieren, da sie somit ihren Wirkungsbereich genauer beschreiben können. Andere werden durch diese Möglichkeit erst sinnvoll umsetzbar. Insbesondere ein Monitoring von Änderungen im Datenraum, das Erzwingen von Invarianten oder eine generell wiederverwendbare Implementation des Observer-Patterns aus der Objektorientierung werden dadurch erst möglich.

Es kristallisiert sich also schnell eine Gemeinsamkeit heraus. Alle Anwendungsfälle beruhen auf die eine oder andere Weise auf der Beobachtung einer Variablen und der Reaktion auf Ereignisse in ihrem Lebenszyklus. Im folgenden werden wir dieses erweiterte Observer-Pattern daher als prototypisch für die gesamte Gruppe an Anwendungsmöglichkeiten sehen.

Andere Anwendungsfälle[MK03], wie die automatische Prüfung von Benutzereingaben, wenn sie in sicherheitskritische Bereiche wie Datenbankabfragen gelangen, erfordern ein Verfolgen des Datenflusses, um so basierend auf Ursprung und Ziel Entscheidungen

treffen zu können. Auch könnten existierenden Aspekte davon profitieren, wenn sie ihren Einfluss auf spezifische Objekte begrenzen könnten, statt nur eine Typ-basierte Auswahl zu treffen. Insbesondere beim Einweben von Advice an allgemeinen Datenstrukturen wie Listen bieten sich hier große Potentiale, da es häufig ausreichend ist eine Teilmenge der Objekte zu beeinflussen.

2.3. Planung möglicher Entwicklungsschritte

Da nun eine Vorstellung der Möglichkeiten und Anforderungen besteht, ist es Zeit Abwägungen zu Größe und Reihenfolge der einzelnen Entwicklungsschritte zu tätigen. Hierbei besteht die Möglichkeit nach der für einen Anwendungsfall benötigten Funktionalität sowie deren Umfang in der Umsetzung in den einzelnen Schritten zu unterteilen. Genauso kann diese Unterteilung basierend auf dem Umfang des unterstützten Sprachumfangs beruhen.

Es erscheint dabei logisch, sich zunächst auf einfache Features zu konzentrieren, da zu erwarten ist, dass diese eine bessere Basis für spätere Umsetzung komplexer Erweiterungen bieten. So bietet es sich an, sich zunächst auf einfache Datentypen zu beschränken um Problemen wie dem Überschreiben von Operatoren aus dem Weg zu gehen. Liegt dann eine robuste Grundlage vor, können weitere Möglichkeiten zur Lösung, der sich für andere Typen ergebenen Probleme, untersucht werden. Auch bietet es sich an die Alias-Problematik zunächst nicht zu berücksichtigen. Sie stellt zwar ein Kernproblem dar, aber auch ohne eine Lösung für sie, werden sich einige Anwendungsfälle abdecken lassen und so der Aufwand für die Umsetzung einer ersten Befähigung reduzieren.

Bei der Abwägung zwischen einer einfachen Lösung mit Laufzeitkomponenten gegenüber einer Lösung ohne, die aber aufwendige Analysen erfordert, kann in frühen Schritten die einfachere Variante gewählt werden, um dann in späteren Schritten die bessere Lösung nachzureichen. Generell sollten solche Optimierungen auf spätere Schritte verschoben werden um eine frühe Unterstützung von Features zu erreichen, damit diese ausführlich auf ihren Nutzen hin getestet werden können.

2.4. konkrete erste Schritte

Diese Arbeit wendet sich daher zunächst der Unterstützung von Anwendungsfällen zu, denen das Observer-Pattern zu Grunde liegt. Dieses bietet eine klar definierte Anforderung an die notwendige Funktionalität. Die Simplizität der Anforderungen wird es ermöglichen grundlegende Definitionen und Konzepte zu erarbeiten. Trotzdem besteht das Potential eine große Zahl an Anwendungsfällen abzudecken. Weiterhin bieten die bereits diesen Bereich abdeckende Umsetzung in AspectJ bzw. die Variante von Magnusson gute Vergleichsmöglichkeiten.

Zunächst werden wir uns daher den Advice für einzelne Datenzugriffe zuwenden. Dabei wird sich sowohl die Art der unterstützten Ereignisse als auch der Umfang der verwendbaren Datentypen und Variablenarten auf ein Minimum beschränken um diesen ersten

Schritt klein zu halten. In weiteren Schritten kann dann eine Erweiterung des Sprachumfangs seitens C++ um unproblematische Teile geschehen, die wichtig für die Abdeckung von Anwendungsfällen sind. Es ist zur erwarten, dass diese einfachen Fälle, sowohl ohne Laufzeitkomponenten als ohne umfangreiche Analyse des Zielprogramms auskommen.

Danach soll es statt sich einer Vervollständigung des unterstützten C++-Sprachumfangs um Möglichkeiten gehen die Alias-Problematik zu handhaben. Dies geschieht um eine bessere Abschätzung der Sinnhaftigkeit von Daten-basierenden Joinpoints in einer Sprache wie C++ zu ermöglichen. Hierbei werden Laufzeitkomponenten zum Einsatz kommen, um eine aufwendige Programmanalyse zu vermeiden. Diese würden selbst mit einer Analyse benötigt, da viele Programme nicht vollständig analysiert werden können.

Bei der konkreten Umsetzung dieser Schritte wird dabei die Koordination der Änderungen mit den parallel von Simon Schröder im Rahmen der “Entwicklung und Bewertung von Advice für vordefinierte Operatoren in AspectC++”[Sch15] vorgenommenen erfolgen, um Synergieeffekt weitest möglich zu nutzen und eine spätere Zusammenführung beider Varianten mit dem Hauptzweig der Entwicklung zu ermöglichen.

3. Grundlegende Advice für Datenzugriffe

Ziel dieses ersten Schrittes ist nun Advice zu ermöglichen, die die Anforderungen des Observer-Pattern abdecken. Es muss also ermöglicht werden, auf zwei grundlegende Ereignisse im Programmablauf zu reagieren: Nutzung und Veränderung von Daten.

Dazu muss Entwicklung in zwei wesentlichen Bereichen stattfinden. Zunächst muss auf der Sprachebene eine genaue Definition der gewünschten Joinpoints erfolgen, auf dieser aufbauend können dann die notwendigen Erweiterungen bzw. Anpassungen der Joinpoint-API sowie von Syntax und Semantik der Pointcut-Ausdrücke erfolgen. Da zunächst die Alias-Problematik explizit ausgeklammert wird, sind keine Laufzeitkomponenten notwendig, so dass anschließend nur noch auf der Ebene der Werkzeuge die Erweiterung des AspectC++ Webers um die Unterstützung der neuen Sprachfeatures erfolgen muss.

Diese erste Erweiterung wird sich in einigen Teil auf die Unterstützung nur eines Teils des C++ Sprachumfanges beschränken. Dies erfolgt auch mit dem Hintergrund den Umfang des ersten Schritts zu reduzieren um so schneller zu komplexeren Kernpunkten wie der Alias-Problematik vorstoßen zu können. Es werden daher jeweils nur wichtige Teile exemplarisch behandelt. Dies begünstigt zudem eine größere Klarheit bei der Findung der grundlegenden Definitionen. Die so erzielten Ergebnisse sollten dann ohne größere Probleme auf die ausgelassenen Teile übertragbar sein oder ein solides Gerüst für die Erweiterung bieten. Daher muss darauf geachtet werden, dass eine Erweiterung um die ausgelassenen Sprachelemente nicht durch getroffene Designentscheidungen behindert wird, so dass es möglich ist diese in späteren Schritten zu ergänzen.

3.1. Grundlagen: Daten in C++

Daten als wichtiges Programmelement sind in C++ in Form von Variablen (sowie Konstanten und Literalen, die aber ähnlich genug sind, als dass wir weiter auf sie eingehen müssen) enthalten. Da die verwendeten Begriffe stets etwas variieren, zunächst ein kurzer Überblick und die Begriffsdefinition für 4 Bezeichnungen, die im weiteren wichtig sind. Insbesondere die Unterscheidung zwischen diesen wird eine gewichtige Rolle spielen:

Datentyp: Jede Variable hat einen Typ, der sie definiert. Er bestimmt die wesentlichen Eigenschaften und Verwendungsmöglichkeiten.

Datenobjekt: Jede Variable stellt ein Datenobjekt dar, auf dem durch den Typ definierte Operationen ausgeführt werden können. Bei Konstanten und Literalen ergeben

sich hier Einschränkungen auf Grund ihrer Natur. Zur Laufzeit ist diesem ein fester Speicherbereich zugewiesen. Auf Grund der Art des Speichers lassen sich die Variablen in Kategorien aufteilen, die bestimmte Eigenschaften teilen.

Wert: Variablen haben einen Wert, der sich in der Regel im Laufe des Programm durch Operationen ändert.

Verweise: Es existieren ein oder mehrere Verweise auf das Datenobjekt, über die dieses angesprochen werden kann. Der einfachste ist ein Programmsymbol bzw. ein Name im Quellcode. Referenztypen stellen eine weitere Möglichkeit dar, auf die wir später eingehen werden.

Zur Veranschaulichung ein kurzes Beispiel:

```
1 int global = 4; // Variable 1
2
3 void main() {
4   bool local = false; // Variable 2
5 }
```

Es wurden 2 Variablen definiert, die sich in allen 4 Punkten unterscheiden:

- Sie unterscheiden sich im Datentyp: Variable 1 hat den Typ “int”, während Variable 2 ein “bool” ist.
- Die Datenobjekte fallen in verschiedene Kategorien: Variable 1 ist global definiert und es wird daher bereits zur Übersetzungszeit Speicher für ihr Datenobjekt reserviert. Die zweite Variable dagegen wurde innerhalb einer Funktion definiert, so dass ihr Datenobjekt zur Laufzeit auf dem Stack Platz findet.
- Sie erhalten die initialen Werte 4 bzw. false.
- Variable 1 ist im gesamten Programm über das Symbol “global” ansprechbar, während das Symbol “local” der zweiten Variablen nur innerhalb der Funktion “main” gültig ist.

Insbesondere die klare Unterscheidung zwischen dem Datenobjekt, den Verweisen über die selbiges erreicht werden kann und dem Wert sind im folgenden wichtig, da sich z.B. aus dem Unterschied der ersten beiden die Alias-Problematik ergibt.

3.1.1. Datentypen

C++ kennt mehrere Gruppen von Datentypen. Grundlage bilden die einfachen Typen: bool, char, diverse Integer- sowie Floatingpointvarianten. Aufbauend darauf existieren die Referenztypen, welche sich durch einen Verweis auf ein fremdes Datenobjekt als Wert auszeichnen. Die Pointertypen handhaben diesen Verweis explizit, während Referenzen diesen verbergen und sich so transparent nutzen lassen.

Die letzte Gruppe bilden die zusammengesetzten Typen. Hier besteht eine größere Diversität. Allen gemein ist, dass Datenobjekte dieser Typen, mehrere Unterobjekte

enthalten, auf denen separate Operationen ausgeführt werden können. Arrays sind die einfachste Variante: Sie enthalten eine definierte Anzahl Datenobjekte des selben Typs und adressieren diese über einen numerischen Index.

Structs hingegen gehören zu den explizit zu definierenden Typen. Hier wird die Zusammensetzung vom Programmierer festgelegt. Sie enthalten Datenobjekte beliebigen Typs, die jeweils über ein, nur in Verbindung mit dem umfassenden Typ, gültiges Symbol adressierbar sind. Unions und Klassen stellen jeweils Sonderfälle von Structs dar, die sich in einigen Punkten unterscheiden, welche für uns aber weitestgehend irrelevant sind. Für Klassen ist jedoch zu beachten, dass durch die Möglichkeit der Operatorüberladung dem Programmieren die Möglichkeit gegeben wird, eine eigene Semantik für Operationen auf dem Gesamtobjekt zu definieren.

3.1.2. Kategorien der Datenobjekte

Das primären Kriterien zur Kategorisierung der Datenobjekte sind ihre Allozierungsparameter, d.h. Art, Zeitpunkt und Ort der Reservierung des benötigten Speicherplatzes. Aus diesen ergeben sich die Lebensdauer der Variablen sowie die Gültigkeitszeit der Verweise auf ihr Datenobjekt.

Die erste Gruppe bilden statisch allozierte Variablen. Für diese wird zur Übersetzungszeit vom Linker Speicher reserviert. Dieser befindet sich in einer der Datensektionen des ausführbaren Programms und steht die ganze Ausführungszeit genau für diese Variable zur Verfügung. In diese Gruppe fallen globale Variablen, aber auch statische Member von Klassen sowie "static" definierte Variablen in Funktionen. Die Lebensdauer erstreckt sich über das gesamte Programm, so dass die Gültigkeit von Verweisen nicht beschränkt ist.

Die nächste Gruppe bilden die lokalen Variablen innerhalb der Funktionen. Diesen wird zur Laufzeit Speicherplatz auf dem Programmstack zugewiesen. Dies beschränkt ihre Lebensdauer auf die Laufzeit der Funktion deren Teil sie sind. Daraus folgt, dass etwaige Verweise die diese Zeit überleben, ihre Gültigkeit verlieren und insbesondere, dass der selbe Speicher zur Laufzeit des Programms durch verschiedene Variablen belegt werden kann. Aber auch dass mehrere Instanzen der selben Variablen gleichzeitig existieren können.

Aus der Möglichkeit des Programmierers Speicher explizit zur Laufzeit anzufordern, ergibt sich eine weitere Kategorie. Für diese wird auf dem Heap Speicher vorgehalten. Diese Art von Variablen zeichnet sich durch das Fehlen eines Symbols als Verweis aus. Es existieren nur Verweise in Variablen mit Referenztyp. Die Lebensdauer dieser Variablen ist generell sehr unterschiedlich und schwer zur Übersetzungszeit zu bestimmen, da sie durch konkrete Operationen (new / delete) beschränkt wird und so vom Ablauf des Programms abhängen.

Einen Sonderfall bilden die Member von Structs / Klassen. Sie haben gemeinsam, dass sie fester Bestandteil eines anderen Datenobjekts sind. Sie erben daher von diesem ihre Allozierungsparameter und Lebensdauer. Obwohl sie nur als Teil existieren können und ein Symbolverweis nur in Verbindung mit dem umgebenen Objekt Gültigkeit besitzt, besteht die Möglichkeit Verweise in Referenztypen zu erzeugen, die diese Ab-

hängigkeit nicht besitzen. Für einen solchen Verweis ist somit nicht bekannt, ob er auf ein selbständiges Datenobjekt zeigt, oder nur auf den Teil eines größeren Objektes.

3.1.3. Symbole und andere Verweise

In einem Programm existiert zu jeder Variablen mindestens ein Verweis[1], der auf das zugehörige Datenobjekt zeigt. In der Regel ist dies ein Programmsymbol, dass mit der Definition der Variablen einher geht. Dieses Symbol kann auf einen bestimmten Scope oder Namensraum beschränkt sein, aber auch für das gesamte Programm Gültigkeit besitzen. Symbole von Membervariablen sind nur in Verbindung mit einem Elternobjekt gültig. Das Symbol ist aber in jedem Fall zur Übersetzungszeit bekannt und auflösbar. Verweise in Form der Werte eines Referenztyps unterliegen diesen Beschränkungen nicht und können wie jeder andere Wert auch in beliebige Datenobjekte eines passenden Typs wandern. Sie können erst zur Laufzeit aufgelöst werden.

3.1.4. Datenwerte

Die Datenwerte sind durch den Typ beschränkt, können aber untereinander konvertiert werden. Im folgenden sind sie nicht von besonderer Relevanz, es ist nur die Unterscheidung von den anderen Begriffen zu beachten.

3.1.5. Einschränkungen für den ersten Umsetzungsschritt

Im folgenden werden wir unsere Betrachtungen auf eine Teilmenge der C++-Sprachelemente beschränken. Globale Variablen einfacher Datentypen stellen einen klar definierten Bereich dar, der es dennoch erlauben wird grundlegende Definitionen zu treffen und ein solides Fundament zu legen. Die spezifischen Gründe zunächst bestimmte Teile auszunehmen werden jeweils in den folgenden Abschnitten erläutert, die sich mit dem begründenden Thema auseinandersetzen. Hier jedoch ein kurzer Überblick:

Referenztypen stellen den Kern der Alias-Problematik dar, und werden daher im Kapitel 4 im Rahmen dessen behandelt. Zusammengesetzte Datentypen erfordern besonderen Aufwand bei der Semantikdefinition in Verbindung mit Joinpoints und werden daher zurückstellt um möglichst aus einer soliden Basis bei den anderen Datentypen eine Ableitung möglicher Definitionen tätigen zu können.

Für lokale Variablen sowie für explizit dynamisch allozierte Variablen bestehen Probleme bei der Umsetzung der Pointcut-Ausdrücke, da diese in Teilen auf den Symbolen basieren, die hier entweder fehlen oder einige Besonderheiten aufweisen. Somit werden auch diese im Rahmen späterer noch zu tätigender Schritte behandeln werden müssen.

Membervariablen weisen einige Besonderheiten auf, so dass wir erst am Ende dieses Kapitels auf sie eingehen werden. Es wird sich jedoch herausstellen, dass eine Unterstützung nur mit sehr geringfügigen Problemen verbunden ist.

[1] Variablen ohne noch existierenden Verweis können nur bei expliziter dynamischer Allozierung auftreten. In diesem Fall wäre die Variable in keinster Weise mehr nutzbar. Allgemein wird das Vorkommen eines solchen Zustandes als Fehler angesehen.

3.2. Joinpointdefinitionen

Grundlegend für jeden Advice sind die Joinpoints, an denen ein Eingriff erfolgen soll. Wir müssen also zunächst die Definition bestehender Joinpoints so erweitern, dass sie die neuen Anwendungsfälle abdecken oder hierzu neue Typen von Joinpoints definieren.

Für Anwendungsfälle, die dem Observer-Pattern folgen, besteht Interesse an zwei Arten von Ereignissen im Programmablauf:

- Verwendung des Wertes einer Variablen (Lesezugriff)
- Veränderung des Wertes (Schreibzugriff)

Diese beiden lassen sich mit den bestehenden Möglichkeiten nicht bzw. nur sehr unzureichend abdecken. Bisher werden Variablen in AspectC++ nicht berücksichtigt. Es besteht also nur die Möglichkeit auf diese Ereignisse näherungsweise zu reagieren, indem Advice für diejenigen Call- oder sonstigen Codejoinpoints definiert werden, welche den gewünschten Ereignissen im Programmablauf möglichst nahe liegen. Hierbei ergeben sich zwei Probleme: Erstens sind diese Joinpoints nur für definierte Funktionen existent, was nur eine gute Näherung erlaubt, wenn eine Funktion existiert, die im wesentlichen nur diesen einen Zugriff enthält (z.B. Getter/Setter). Zweitens ist es schwierig die Parameter und den Kontext des eigentlichen Ereignisses zu berücksichtigen oder zu beeinflussen, da diese möglicherweise nicht am jeweiligen gewählten Joinpoint zur Verfügung stehen, da für diesen nicht relevant. Abhilfe kann hier der Programmierer durch entsprechende Codemodifikationen schaffen. Dies bedeutet aber einen erheblichen Aufwand und widerspricht der Kapselung und Austauschbarkeit der jeweiligen Komponenten.

Es besteht also Bedarf an neuen Mitteln. Grob bestehen 3 Optionen, deren Vor- und Nachteile wir im folgenden kurz besprechen wollen:

3.2.1. Option 1: rhs / lhs

Eine enge Anlehnung an die Syntax von C++ verspricht klare, etablierte Definitionen und weniger Aufwand sowohl im Entwurf als auch in der Umsetzung. Joinpoints für die Verwendung einer Variablen als rhs- bzw. lhs-Ausdruck (im C++-Standard häufig auch RValue bzw. LValue) erscheinen auf den ersten Blick sinnvoll. Es wäre keine weitere Semantikdefinition nötig, aber hier liegt die große Schwachstelle. Die Semantik dieser Joinpoints wäre nur sehr schwach definiert, da sie sich an Syntaxelementen orientiert, deren Bedeutung sich in der Regel erst aus dem Kontext ergibt. So ist eine Variable gemäß C++-Standard immer ein LValue und wird erst bei Bedarf in ein RValue umgewandelt. Es wäre zwar möglich die Definition hier etwas zu dehnen und an diesem Punkt anzusetzen um das erste geforderte Ereignis abzudecken, aber insbesondere das zweite stellt ein Problem dar. Anschaulich wird dies im folgenden Beispiel:

```
1  void call_by_ref1( const int &r ) { global += r; }
2  void call_by_ref2( int &r ) { r = global; }
3
4  void main() {
5    int a;
6    a = 5;
7    int *ptr = &a;
8    call_by_ref1( a );
9    call_by_ref2( a );
10 }
```

Alle 4 Vorkommen von “a” erfordern ein LValue, unterscheiden sich aber erheblich in der Bedeutung bzw. in der zu erwartenden Verwendungen. Obwohl die Zuweisung(Zeile 6) sowie die Adressbildung(Zeile 7) beide zwingend ein LValue erfordern, ist erstere der prototypische Fall eines Schreibzugriffs, während bei letzterer zunächst keiner der beiden definierten Fälle vorliegt und die Art möglicher folgender Zugriffe unklar ist. Auch bei der Parameterübergabe per Referenz zeigt sich, dass die Erwartungen nicht mit der vorgeschlagenen Definition der Joinpoints übereinstimmen. Beide Referenz erfordern einen LValue, doch die const-Referenz schließt im Prinzip Schreibzugriffe aus.

Die Alias-Problematik wäre nur eingeschränkt vorhanden, denn LValues ließen sich nicht an Hand ihres Ursprunges unterscheiden. Der Programmierer stände hier somit alleine auf weiter Flur, denn er könnte möglicherweise an Hand der Signaturen der beteiligten Typen Rückschlüsse ziehen, aber dies wäre sowohl aufwendig als auch in der Erwartung wenig zuverlässig. Eine Möglichkeit der Unterstützung auf einer solideren Basis wäre hier wünschenswert.

Ferner ist unklar in weit der Programmierer mit den syntaktischen Details der Sprache vertraut ist. Die Unterscheidung der Joinpoints anhand ihrer syntaktischen Eigenheiten erscheint daher wenig intuitiv und bei näherer Betrachtung nicht unbedingt geeignet für die weitere Entwicklung.

3.2.2. Option 2: Call für vordefinierte Operatoren

Wir haben gesehen, dass die Möglichkeit besteht, dem gewünschten durch Call-Joinpoints bereits nahe zu kommen, sowie dass die Bedeutung des Erscheinens einer Variablen im Quellcode stark vom Kontext abhängt. Daher erscheint die Möglichkeit, mittels der von Simon Schröder[Sch15] geschaffenen zusätzlichen Call-Joinpoints basierend auf vordefinieren Operatoren, die am Anfang des Kapitels geschilderte Problematik soweit abzumildern, dass eine sinnvolle Nutzung möglich wird, vielversprechend. Denn genau diese Operatoren stellen häufig den Kontext dar, der über die Bedeutung der Verwendung einer Variablen entscheidet. Ein Problem stellt hier die Verwendung außerhalb eines Operatorkontextes dar. Beispiele sind hier die verschiedenen C++-Schlüsselworte: “while (a) {}” / “return a;” etc. Dieses könnte aber durch entsprechende Advice für eben diese gelöst werden.

Um dem Programmierer zu erlauben seine Advice auf bestimmte Variablen zu beschränken wäre es nötig, die bestehenden Pointcut-Funktionen für Argumente und Zie-

lobjekt so zu erweitern, dass diese nicht nur eine Filterung nach Typ, sondern auch nach der Variablen selbst erlauben. Dies wurde auch bereits in Abschnitt 2.2 als mögliches eigenständiges Feature angesprochen. Insbesondere für das Zielobjekt könnte dies zu Problemen führen, da dieses nicht mehr eindeutig definiert wäre. Zu jedem Argument ließe sich im Falle einer Membervariablen ein Objekt finden, dass der Definition entspricht.

Der größte Kritikpunk an dieser Variante besteht aber im Umfang der zu erwartenden Pointcut-Ausdrücke um die beiden gegebenen Anwendungsfälle abzudecken und somit in der großen Fehleranfälligkeit. Um alle Änderungen einer Zielvariablen zu erfassen, müsste der Programmierer in diesen alle Operatoren auflisten, die eine Änderung bewirken könnten. Diese Menge ist zwar noch überschaubar, birgt aber trotzdem die Gefahr von Fehlern durch Auslassung. Für einen Lesezugriff würde die Menge sogar noch größer, da sie fast alle Operatoren umfassen müsste. Es bestünde die Möglichkeit AspectC++ so zu erweitern, dass dem Programmierer die Arbeit durch vordefinierte Pointcut-Funktion entweder abgenommen oder zumindest erleichtert wird, doch dies löst das Problem nur bedingt.

Denn auch bei der Joinpoint-API bzw. im Advice-Code selbst stellt sich ein ähnliches Problem. Die einzelnen Operatoren unterscheiden sich erheblich in ihren Parametern. Ein einfaches Beispiel geben hier die Arithmetischen Operatoren: Beide Argumente werden gelesen, aber für eine Filterung nach Namen der Variablen müssten entweder zwei Advice geschrieben werden (für jedes Argument einer) oder es müsste mit Wildcards gearbeitet werden und dann innerhalb des Advice eine Prüfung erfolgen. Beides ist zusätzlicher Aufwand, der vom eigentlich Zweck ablenkt und neue Fehlerquellen schafft. Ein anderes Beispiel lässt sich bei den Varianten des Zuweisungsoperators finden. Die einfache Variante erhält als zweites Argument den neuen Wert, der der als erstes Argument übergebenen Variablen zugewiesen werden soll. Die Verbundvarianten dagegen berechnen den neuen Wert erst aus diesem zweitem Argument. Bei einer Gültigkeitsprüfung vor der Zuweisung müsste also hier eine Fallunterscheidung eingebaut werden, die gegebenenfalls die Operation emuliert und so den neuen Wert schon zur Prüfung bereitstellt.

Auch wenn diese Möglichkeit der Joinpointdefinition wenig Aufwand an neuen Definitionen verspricht und gleichzeitig eine große Mächtigkeit, Flexibilität und auch bei Bedarf hohe Präzision in der Joinpointselektion erwarten lässt, stellen das Fehlen an Abstraktion und damit die Notwendigkeit an Boilerplate-Code für die allgemeinen Fälle einen erheblichen Mangel dar. Daher kann diese Möglichkeit nicht die erste Wahl für eine alleinige Implementierung sein, auch wenn sie durchaus Potenzial als spätere Erweiterung besitzt.

3.2.3. Option 3: get / set

Führt man den Gedanken der Abstraktion von konkreten Sprachelementen hin zu einer stärker semantisch geprägten Definition fort, landet man fast zwangsläufig bei den zwei Konzepten wie sie in Gettern und Settern bereits angesprochen wurden. Diese beiden decken nicht nur ziemlich genau die Anwendungsfälle des Observer-Patterns ab, sondern

sind in der Objekt-Orientierung ebenfalls bereits etablierte Konzepte. Auch ist die Definition basierend auf diesen Konzepten aus anderen Aspekt-orientierten Sprachen bzw. Umsetzungen bekannt. So unterstützt sowohl AspectJ[KHH+01, Abschnitte 3.1, 3.2] für Java als auch die AspectC++ Erweiterung von Magnusson[Mag06], wie sie in [AÖ07] beschrieben wird, Joinpoints nach einer vergleichbaren Definition. Zu beachten ist, dass AspectJ keine Unterscheidung zwischen dem Datenobjekt und den Verweisen treffen muss, da in Java nur Verweise in Form des Symbols existieren, während die Umsetzung von Magnusson sich explizit auf Zugriffe über das Symbol beschränkt.

Die Definition eines Get-Joinpoints als lesenden Zugriff auf ein Datenobjekt unabhängig vom verwendeten Verweis ist semantisch wohldefiniert, aber auch in der C++-Syntax hinreichend verankert. Ist sie doch mit dem oben angesprochenen leicht gedehnten Begriff eines RHS-Ausdruckes für praktische Zwecke identisch. Weiterhin stellt sie genau die Abstraktion dar, welche als Verbesserung der Operatorjoinpoints vorgeschlagen wurde, bietet aber als eigenständige Joinpointart im Gegensatz zu einer reinen Pointcut-Funktion die Möglichkeit eines einheitlichen Joinpoint-APIs.

Analog dazu lässt sich nun ein Set-Joinpoint als schreibender Zugriff definieren. Im Vergleich zu den Operatoren findet hier ebenfalls die notwendige Abstraktion statt, während sie die Menge der möglichen LHS-Joinpoints hinreichend auf eine konkrete Verwendung einschränkt, um somit eine sinnvolle und klar definierte Nutzung zu ermöglichen. Die anderen Verwendungen eines LHS-Ausdruckes lassen bei Bedarf über weitere Joinpointarten abdecken.

Auch wenn die Konzepte dahinter intuitiv erfassbar sind, müsste die genaue Definition noch explizit erfolgen, da einige Randfälle wie die Verwendung eines einfachen Ausdrucks als Statement: “a;” oder die genaue Semantik für alle Datentypen, insbesondere der zusammengesetzten, eine genauere Betrachtung erfordern.

Diese Möglichkeit der Definition scheint gut erweiterbar zu sein, bietet aber gleichzeitig eine solide Grundlage für eine weitere Entwicklung. Die Alias-Problematik kann zunächst eine untergeordnete Rolle spielen, da die Definition nicht von der Art des Verweises über den ein Zugriff erfolgt abhängt. Sie kann also bereits in der Sprache verankert werden, während der Weber zunächst nur Zugriffe über die Symbole der Variablendefinition unterstützt und eine Erweiterung zur Lösung der Alias-Problematik kann zu einem späteren Zeitpunkt nachgereicht werden. Auch fügt sie sich gut in bestehende Strukturen ein, denn es besteht eine große Ähnlichkeit zu den bestehenden Call-Joinpoints für Funktionen, so dass sich beispielsweise die Semantik dieser leicht auf Funktionspointer übertragen ließe.

3.2.4. Übersicht & Vergleich

Die Vor- und Nachteile der 3 Optionen lassen sich wie folgt zusammenfassen:

- RHS / LHS
  - ⊕ einfache Definition basierend auf Syntax
  - ⊕ Umfassend und abschließend

- ⊖ zu wenig Präzision in der Auswahl der gewünschten Ereignisse möglich
- ⊖ wenig Unterstützung für den Programmierer

- Call für eingebaute Operatoren
  - ⊕ aufbauend auf bestehenden Definitionen
  - ⊕ mächtig und gleichzeitig flexibel
  - ⊖ wenig Abstraktion, daher komplexe Pointcuts und uneinheitliche API

- Get / Set
  - ⊕ einfache Pointcuts / einfache API
  - ⊕ intuitive / bereits etablierte Konzepte
  - ⊕ gute Erweiterbarkeit
  - ⊖ erfordert nicht wenige neue Definitionen um alle Bereich abzudecken

Die erste Option (RHS/LHS) steht im Prinzip nicht zur Wahl, da sie die gewünschte Funktionalität nicht oder nur sehr schlecht erfüllen würde. Die Option der Call-Joinpoints ist zwar vielversprechend und sollte definitiv für spätere Erweiterungen in Betracht gezogen werden, erweist sich in der Diskussion aber der dritten Option (auf das Problem zugeschnittene Joinpointarten) als unterlegen, da sie den Programmierer auf Grund der fehlenden Abstraktion nicht so gut unterstützen kann.

In diesem ersten Schritt werden also Joinpoints für Get und Set geschaffen, für die im folgenden eine genaue Definition festgelegt werden muss. Die Alias-Problematik kann zunächst ausgeklammert werden, da es möglich ist, diese in einem späteren Schritt zu behandeln.

3.2.5. Definitionen

Der Kern der Definitionen liegt bereits vor, doch es sind noch einige Randfälle unklar und zu überdenken. Auch wenn es wünschenswert wäre, muss eine erste Definition noch nicht vollumfänglich sein und kann bewusste Lücken aufweisen, die späteren Schritten überlassen werden. Sie muss aber so vollständig sein, dass spätere Schritte sie nur erweitern und sich konsistent integrieren lassen, so dass keine Widersprüche geschaffen werden.

Eine Verwendung einer Variablen als RHS-Ausdruck im Rahmen eines Operators, C++-Schlüsselwortes oder eines Funktionsaufrufs entspricht dem Kern der informell gegebenen Definition. Doch einige Fälle müssen näher betrachtet werden:

```
1 int a, b;
2
3 if( a ) {}       // Fall 1
4 while( ! a ) {} // Fall 2
5 a + b;          // Fall 3
6 a;              // Fall 4
```

Die Fälle 1 + 2 sind recht klar. Die Variable wird im Kontext eines Schlüsselwortes bzw. unären Operators verwendet. In beiden Fällen wird nur ein RValue gefordert, so dass sie in ein RValue passenden Typs umgewandelt wird. Im Zuge dessen wird lesend auf diese zugegriffen. Für den Compiler besteht nun zwar die Möglichkeit diesen Zugriff auszulassen, wenn der Wert an dieser Stelle bereits bekannt ist (statisch ermittelbar oder von einem vorherigen Zugriff), aber die Definition muss diese Fälle trotzdem umfassen, da sie unabhängig von den Optimierungsfähigkeiten des konkreten Compilers zu sein hat. Andernfalls wäre die Anwendung der Definition auf konkreten Code auch nicht nachvollziehbar. Die beiden letzten Fälle sind schwieriger.

In Fall 3 wird aus den beiden Variablen ein neuer Wert errechnet, dieser jedoch offensichtlich nicht verwendet. Es ist also davon auszugehen, dass hier kein Code vom Compiler generiert wird, doch aus Gründen der Konsistenz sollte die Definition hier zwei Get-Joinpoints vorsehen. Dass das Endergebnis nicht verwendet wird, spielt im konkreten Kontext des jeweiligen Joinpoints keine Rolle, denn wie in den anderen Fällen auch erfolgt die Umwandlung in ein RValue und somit in der Regel eine Ladeoperation.

Im letzten Fall 4 erfolgt genau die Umwandlung mangels Bedarf nicht. Im Interesse der Klarheit der Definition erscheint es nicht sinnvoll hier eine Ausnahme zu machen, daher sollte in diesem Fall kein Joinpoint vorliegen. Es könnte zwar argumentiert werden, dass es eine Möglichkeit schaffen würde einen Joinpoint explizit im Code zu verankern, der garantiert keinen weiteren Effekt besitzt. Die Existenz eines sinnvollen Anwendungsfalls ist hier jedoch fraglich und kann gegebenenfalls auch durch Verwendung anderer Möglichkeiten geschaffen werden (Fall 1 z.B. hat ebenfalls auf Grund des leeren Body-Statements keinen Effekt.)

Doch auch im Rahmen von LHS-Ausdrücken können Get-Joinpoints auftreten:

```
1 a += 5;          // Fall 1
2 --a;             // Fall 2
3 if( a = 3 ) {} // Fall 3
4 b = 4;           // Fall 4
```

Neben den zu erwartenden Set-Joinpoints ist in den ersten drei Fällen jeweils noch ein Get-Joinpoint vorhanden. In den ersten beiden Fällen ist es nötig den alten Wert zu kennen um den neuen zu berechnen, daher muss hier ein Ladevorgang erfolgen und folglich ein Get-Joinpoint berücksichtigt werden. Im 3. Fall liegt der Get-Joinpoint nach der Zuweisung. Der Rückgabewert der Zuweisung ist ein LValue, welches vor der Auswertung im Rahmen der If-Bedingung in RValue umgewandelt wird. Es kann zwar argumentiert werden, dass hier im Normalfall kein Ladevorgang stattfindet, da der Wert auf jeden Fall bereits bekannt ist, doch die Konsistenz spricht auch hier für einen definierten Joinpoint. Im letzten Fall entfällt diese Umwandlung, daher ebenso der Get-Joinpoint.

Für Set-Joinpoints gibt es neben den analog explizit einzuschließenden Joinpoints, die möglicherweise mangels Effekt (Verbundzuweisung mit neutralem Element etc.) oder aus Optimierungsgründen entfallen könnten, nur einen Randfall, der zu klären ist.

Die Initialisierung einer Variablen kann große Ähnlichkeiten mit einer Zuweisung aufweisen. (Je nach verwendeter Syntax optisch mehr oder weniger deutlich.) Es erscheint also plausibel diese ebenfalls in die Definition mit aufzunehmen. Doch erscheint auch

die Schaffung eines eigenen Typs von Joinpoint denkbar, der analog zu der Beziehung zwischen Call und Execution einen Kontrapunkt zu den bestehenden Construction-Joinpoints setzen könnte. Mangels eines konkreten Anwendungsfalls, der die eine oder die andere Möglichkeit erfordert, ist es sinnvoll diese Entscheidung zunächst offen zu lassen um die spätere Entwicklung nicht zu beeinträchtigen, ohne auf eine fundierte Grundlage verweisen zu können.

Es ergeben sich also die folgenden vorläufigen Definitionen:

Definition 1. Ein Get-Joinpoint bildet einen lesenden Zugriff auf das Datenobjekt einer Variablen ab. Er liegt an den Punkten der Umwandlung des LValues (gemäß C++-Standard) einer Variablen in das zugehörige RValue explizit vor. Das Vorliegen berücksichtigt explizit nicht mögliche Optimierungen oder sonstige Ergebnisse einer Analyse zur Übersetzungszeit, die zu einem Entfallen des Zugriffs führen könnten.

und

Definition 2. Ein Set-Joinpoint bildet einen schreibenden Zugriff auf das Datenobjekt einer Variablen ab. Mögliche Optimierungen oder sonstige Ergebnisse einer Analyse zur Übersetzungszeit, die zum Entfallen des Zugriffs führen könnten, finden explizit keine Berücksichtigung.

Diese beiden Definition decken die Anwendungsfälle des Observer-Pattern ab und können bei Bedarf um weitere ergänzt werden. Da sie explizit auf das Datenobjekt abzielen und somit unabhängig vom Verweis sind, sind sie sowohl in diesem ersten Schritt nutzbar, in dem nur Zugriffe über das Symbol erfasst werden sollen, als auch für eine Erweiterung im Rahmen der Alias-Problematik.

3.2.6. Probleme in Verbindung mit zusammengesetzten Typen

Bei der Anwendung der obigen Definitionen auf Variablen eines zusammensetzten Typs ergeben sich mehrere Probleme. Die Semantik der Joinpoints ist bei einfachen Datentypen klar zu erkennen. Für Structs ist gemäß C++-Standard nur der Zuweisungs- sowie der Adressbildungsoperator definiert. Der Zuweisungsoperator sowie die Möglichkeit Structs als Parameter oder Rückgabewerte zu verwenden würden ausreichen um Joinpoints gemäß den obigen Definitionen zu erzeugen, doch ungeklärt ist die gewünschte Semantik bei Zugriffen, die nicht das gesamte Datenobjekt umfassen:

```
1  struct S {
2    int a;
3    float b;
4  };
5
6  S s1;
7  S s2;
8
9  s1 = s2;   // Fall 1
10 s1.a = 5; // Fall 2
```

Fall 1 ist eindeutig ein lesender Zugriff auf s2, sowie ein schreibenden Zugriff auf s1. Doch Fall 2 verändert nur einen Teil des Datenobjekt von s1. Hier ist die gewünschte Semantik noch nicht klar und diese Teilzugriffsproblematik muss anhand konkreter Anwendungsfälle weiter untersucht werden. In der Zwischenzeit kann durch Advice für den Zuweisungsoperator sowie für die einzelnen Membervariablen jeweils genau der im konkreten Fall gewünschte Effekt erzielt werden. Auf eine Abstraktion wie sie für einfache Datentypen auf Grund der Vielzahl an Operatoren notwendig scheint, kann hier im Zweifel verzichtet werden.

Klassen weisen als erweiterte Structs die selbe Problematik auf, besitzen jedoch in Bezug auf die Semantik bei definierten Operatoren eine weitere Besonderheit. Dem Programmierer ist es möglich den bestehenden Operator zu überschreiben und neue zu definieren. Auch wenn es dem guten Stil entspricht nicht sehr weit von der Semantik für einfache Typen abzuweichen, steht es dem Programmierer frei. Somit kann diese nicht mehr als verlässliche Basis für Definitionen dienen. Ein Beispiel ist der Operator << für Standard-IO-Streams. Als Bitschiebeoperation auf normalen Datentypen stellt er die verwendeten Variablen eindeutig in den Kontext eines Get-Joinpoints, da deren Werte zur Berechnung verwendet werden und das Datenobjekt dabei nicht verändert wird. Auf IO-Streams ist die Bedeutung nun eine ganz andere und kann auch das Datenobjekt verändern.

Für Arrays sind gar keine Operationen auf dem Gesamtobjekt definiert. Es bleiben also nur die Zugriffe auf die einzelnen Teile. Ein Angehen der Teilzugriffsproblematik sowie der Besonderheiten von Klassen bedarf also noch weiterer Untersuchungen und muss daher auf spätere Schritte verschoben werden. Obwohl aus heutiger Sicht keine einheitliche Lösung erwartet werden kann, sollte in der Zukunft eine Reevaluation basierend auf einer breiteren Erfahrungsbasis erfolgen.

3.3. Joinpoint-API

Nachdem nun die Joinpoints an sich hinreichend spezifiziert sind, ist es daran eine Erweiterung bzw. Anpassung der bisherigen API zu entwerfen, die ein Advice nutzen kann. Ziel hierbei muss es sein, dem Programmierer die notwendigen und nützlichen Informationen zur Verfügung zu stellen und dabei eine weitgehende Kohärenz mit der bestehen API zu schaffen um ein einheitliches Interface zu erhalten. Die bestehende API ist in [SL07, Abschnitt 4.4 sowie Fig. 1(g)] dokumentiert, wir können uns also auf die notwendigen Anpassungen konzentrieren.

Viele Teile können ohne weiteres von den anderen Code-Joinpoints übernommen werden. Informationen wie die Signatur der Funktion bzw. Variablen sowie Zeilennummern, Joinpointart oder IDs werden für alle Joinpoint identisch zur Verfügung gestellt. Hier besteht nicht der geringste Grund zu einer Abweichung. Auf Grund der großen Ähnlichkeit zu Call-Joinpoints bietet es sich an, sich bei den vom Joinpoint abhängigen Kernfunktionen der API an diesen zu orientieren. Für Funktionen sind dies im wesentlichen die beteiligten Objekte und die Aufrufinformationen (Argumente und Ergebnis). Zu diesen sind jeweils die beteiligten Typen sowie zur Laufzeit die konkreten Zeiger auf

die Objekte bzw. Werte verfügbar. Die Funktion selbst ist nicht direkt ansprechbar. Nur im Falle von around-Advice besteht die Möglichkeit die Ausführung des Aufrufs explizit fortzusetzen. In den anderen Fällen geschieht dies automatisch.

Element	Funktions-API
Zielfunktion	[nur indirekt via proceed()]
Ausgangsobjekt	that()
Zielobjekt	target()
Argumente	args()
Rückgabewert	result()

Tabelle 3.1.: Joinpoint-API für Call-Advice

Für Datenzugriffe sind nun 2 bzw. 3 Elemente besonders wichtig:

- Die Variable selbst
- Für Get-Joinpoints: das Ergebnis des Lesezugriffs
- Für Set-Joinpoints: der neue Wert der Variablen

Das Zielobjekt bzw. zutreffender das umgebene Datenobjekt spielt für Zugriffe auf Membervariablen eine bedeutenden Rolle. Es ist zwar zu diesem Zeitpunkt noch nicht erforderlich, sollte allerdings mit berücksichtigt werden um eine spätere Unterstützung von Membervariablen zu ermöglichen.

Folgt man nun der Analogie der Call-Joinpoints und dem schon bei Definition der Joinpoints verwendeten Getter/Setter Konzepts, ergeben sich 2 Möglichkeiten. Zum einen kann man die Variablen jeweils als eigenständiges Objekt sehen und die Analogie zu einem dem Joinpoint entsprechenden Memberfunktionsaufruf ziehen:

```
1 int a;
2
3 b = a.get();
4 a.set( 5 );
```

Daraus ergibt sich, dass die Variable über den target()-Teil der API verfügbar wäre und Ergebnis bzw. Rückgabewert analog zu entsprechenden Funktionen gehandhabt werden. Dies würde für den Fall eines Set-Joinpoints große Parallelen zu einem Advice für den Zuweisungsoperator auf einem Objekt aufweisen. Das umgebene Datenobjekt im Falle von Membervariablen wäre allerdings nicht mehr verfügbar und gegebenenfalls müsste hier eine Alternative geschaffen werden.

Zum anderen besteht die Möglichkeit sich stärker an der Idee der Getter und Setter zu orientieren und alle Variablen vergleichbar zu Membern zu behandeln. Nicht Member könnten sich bezüglich der target()-Funktion wie statische Funktionen verhalten:

```
1 b = get_a();
2 set_a( 5 );
```

Damit stünde zwar die target()-Funktion bei Membern für das umgebene Objekt zur Verfügung, aber es bestünde die Notwendigkeit die Variable anderweitig verfügbar zu machen. Möglich wäre diese als Referenzparameter einzuführen, doch die zentrale Bedeutung für den Joinpoint, lässt auch eine eigene neue API-Funktion sinnvoll erscheinen.

Ausschlaggebendes Argument in der Diskussion ist hier die größtmögliche Einheitlichkeit der API. Daher wird die 2. Option favorisiert. Auf Grund der Bedeutung der Variablen als zentralem Element des jeweiligen Joinpoints soll diese über ein neues API-Element verfügbar gemacht werden. Da auch Anwendungsfälle existieren, die bei Call-Joinpoints die Zielfunktion explizit und nicht nur in Form einer Fortsetzung erfordern, wurde für dieses neue Element der allgemeine Name entity gewählt. Dies ermöglicht es, dem Programmierer das zentrale Programmelement eines Joinpoints auf eine einheitliche Weise zur Verfügung zu stellen. Zusammen mit der Orientierung an den üblichen Konventionen für Getter/Setter in Bezug auf Argumente und Rückgabewert ergibt sich der folgende API-Kern:

Element	Funktions-API	Daten-API
Zielentität (Funktion/Variable)	entity() [indirekt via proceed()]	entity()
Ausgangsobjekt	that()	that()
Zielobjekt / umgebenes Objekt	target()	target()
Argumente / neuer Wert	args()	get: - set: args()
Rückgabewert / Ergebnis	result()	get: result() set: -

Tabelle 3.2.: Joinpoint-API für Datenbasierte Advice

Wie deutlich zu erkennen ist sind die Unterschiede der beiden APIs minimal. Datenjoinpoints können bezüglich Argumenten und Rückgabewert analog zu Getter/Setter Funktionen behandelt werden (Get: Rückgabewert und keine Argumente, Set: neuer Wert als einziges Argument und kein Rückgabewert). Dies trägt sicherlich zur einfachen und intuitiven Verwendbarkeit bei. Das neue API-Element fügt sich auch in die Funktions-API nahtlos ein und ist auch für diese wohldefiniert. Da es sich nur um eine Erweiterung und keine Redefinition handelt sind Kompatibilitätsprobleme mit bestehendem Advice-Code nicht zu erwarten. Auch ist zu erwarten, dass die angewandten Prinzipien für den Entwurf bei späterer Erweiterung um neue Joinpoints ebenfalls greifen, so dass dort keine oder nur sehr minimal abweichende APIs genutzt werden können.

3.4. Pointcut-Ausdrücke

Neben der API stellen die Pointcut-Ausdrücke den wesentlichen Teil von AspectC++ auf der Sprachebene dar[SL07, Fig. 1(d-e)]. Sie dienen dazu für einen Advice die Menge der Joinpoints zu bestimmen, an denen dieser eingewoben werden soll.

3.4.1. Grundlegende MatchExpressions

Die Basis hierfür bilden die MatchExpressions. Diese dienen der Selektion der benannten Programmelemente basierend auf deren Signatur. Sie erlauben Wildcards sowohl bei den Typen der Signatur also auch als Teil von Symbolen. Die Signatur für globale Variablen ist einfach anzugeben und besteht aus dem Datentyp sowie dem Symbol ergänzt um etwaige Namensräume. Genau wie für Funktionen entspricht sie also der im Quellcode existierenden Deklaration.

Hier sind daher keine weiteren Definitionen zu treffen, da sich die MatchExpressions für Variablen analog zu jenen von Funktionen ergeben, mit dem geringfügigen Unterschied, dass die Argumenttypen entfallen.

3.4.2. Probleme bei der Erweiterung

Für nicht globale Variablen jedoch ergeben sich Probleme mit dieser Spezifikation. Die Symbole lokaler Variablen sind nur innerhalb der definierenden Funktion gültig. Verzichtet man auf eine weitere Qualifizierung, sind sie höchst mehrdeutig. Es besteht die Möglichkeit sich bei einer Qualifizierung an der Symbolhandhabung des Linkers zu orientieren. Dies beschränkt die Mehrdeutigkeit auf gleiche Symbole innerhalb einer einzigen Funktion (eine Funktion kann mehrere Scopes enthalten). Für den Linker sind nur statisch lokale Variablen relevant, aber eine Übertragung auf nicht statische Variablen stellt kein Problem dar. Es bedarf hier allerdings noch einer genaueren Prüfung bezüglich der Namensraum-äquivalent verwendeten Funktionssignatur, da auch cv-Qualifier berücksichtigt werden sollten, wie es auch bei der Auflösung überladener Funktionen geschieht.

Dynamisch auf dem Heap allozierte Variablen verfügen über gar kein Symbol, das zum Matching herangezogen werden kann. Hier muss also ein anderer Ansatz gefunden werden, bevor diese unterstützt werden können. Ähnlich verhält es sich mit den Elementen von Arrays. Diese sind zwar zur Laufzeit über einen Index spezifizierbar, doch um diesen Ansatz auf die MatchExpr zu übertragen wären einige Änderungen notwendig, so dass vorher die Sinnhaftigkeit intensiv geprüft werden muss. So wäre es insbesondere nicht mehr möglich den Ausdruck zur Übersetzungszeit zu prüfen.

3.4.3. Pointcut-Funktionen

Neben den MatchExpressions sind die eingebauten Pointcut-Funktionen wichtiger Bestandteil der Ausdrücke. Sie ermöglichen es Ableitungen von bestehenden Ausdrücken zu bilden und erlauben so die Selektion der gewünschten Joinpoints. Die Call-Pointcutfunktion zum Beispiel liefert zu einer gegebenen Menge an Funktionen alle Call-Joinpoints, die ein Element der Ursprungsmenge als Ziel haben. Andere existieren für einschränkende Verknüpfungen, so liefert die args()-Funktion nur Joinpoints die eine bestimmte Parametersignatur aufweisen.

Im Zusammenhang mit Advice für Datenzugriffe sind also eine get() und eine set() Pointcut-Funktion notwendig, die analog zum Verhalten der call()-Funktion, eine als Argument übergebene Variablen Menge auf die Menge der entsprechenden Joinpoints

abbilden. Der Ausdruck “call(“int %()”)” liefert so alle Call-Joinpoints, die eine beliebig benannte Funktion im globalen Namensraum aufrufen, die keine Parameter erwartet und einen int liefert. Weitere Ergänzungen sind nicht notwendig, es bleibt nur zu erwähnen, dass auch wenn sich ihre Definition nicht ändert, die anderen Pointcut-Funktionen natürlich im Rahmen, der für die Joinpoint-API genutzten Entsprechungen, auch für die neuen Joinpointtypen nutzbar sein sollten.

3.4.4. Kurzformen

Die Pointcutsprache erlaubt im Rahmen des Matchings von Funktionen eine Kurzform der Art “someNameSpace” welche äquivalent zu der Wildcardform “% someNameSpace::...::%(...)” ist und somit alle Funktionen, die unterhalb des gegebenen Namensraumes definiert wurden, liefert. Eine Entsprechung für Variablen wäre wünschenswert, es ergäbe sich aber eine mögliche Doppeldeutigkeit. Da eine Erweiterung derjenigen Pointcut-Funktionen, welche zur Zeit nur nach Typ auswählen (args(),that(),target()), um die Unterstützung der Selektion eines spezifischen Objekts (z.B. target(“someClass obj1”)) angedacht ist, erscheint auch die Kurzform “someType” -> “someType %” als Ableitung aus der alten und der erweiterten Syntax sinnvoll. Klassen stellen nun sowohl eine Art Namespace als auch einen Typ dar, so dass hier nicht eindeutig aufgelöst werden kann.

Eine genauere Untersuchung steht hier noch aus. Es besteht die Möglichkeit, dass durch den Verwendungskontext der jeweiligen Kurzform die Auflösung eindeutig möglich wird. Dann stände dieser Kurzform nur noch ein Kohärenzargument im Weg, denn es wäre wünschenswert, dass Pointcut-Ausdrücke wenig vom Kontext abhängig sind um sie dem Programmierer verständlicher zu machen. Diese Abwägung wird auf Grund der noch fehlenden praktischen Erfahrungen auf einen späteren Zeitpunk verschoben.

3.5. Erweiterung des Aspekt-C++-Webers

Um die nun definierten Spracherweiterungen im Weber umzusetzen sind drei Gruppen von Modifikationen notwendig. Zunächst muss das Verarbeiten des AST[2] modifiziert werden um das interne Joinpoint-Modell zu füllen, auf dem dann das Auswerten der erweiterten Pointcut-Ausdrücke und das Einweben von Advice an den neuen Joinpoints aufsetzen kann.

3.5.1. Modellbildung

Zunächst musste die bestehende interne Modellstruktur um Klassen für die neuen Joinpoints ergänzt werden und die Klassen für Variablen so modifiziert, dass sie die nun nötigen zusätzlichen Information aufnehmen können, da Variablen bisher noch keine eigene Bedeutung hatten. Sie existierten nur als Container für Joinpoints in ihrem Initialisierungsausdruck. Da große Ähnlichkeiten zu Call-Joinpoints bestehen, wurden die zugehörigen Klassen des Modells aufgespalten. Es entstanden neben den bestehenden, generalisierte Oberklassen, die auch für die Umsetzung der neuen Anforderungen Verwendung finden konnten. Diese enthalten die gemeinsamen Informationen und dienen als Sammelpunkt für Funktionalität, die sowohl für Daten- als auch für Call-Joinpoints benötigt wird.

So konnte einiger bereits bestehender Code nach der Umstrukturierung wiederverwendet werden. Dies begrenzte den Aufwand für die Realisierung der neuen Modellelemente auf ein beherrschbares Maß. Ebenso können zukünftige Erweiterungen von diesen Strukturen profitieren. Bei der Verarbeitung des AST konnten diese Synergieeffekte jedoch nur sehr begrenzt genutzt werden. Die Informationen, die zur korrekten Erkennung und Modellierung eines Joinpoints notwendig sind, verteilen sich auf mehrere Knoten im Syntaxbaum. Zum einen jenen Knoten, welcher den genutzten Verweis auf die Variable enthält, zum anderen denjenigen, welcher den Kontext für den Joinpoint bestimmt und damit seinen Typ festlegt. Für Funktionen ist diese Beziehung zwangsläufig eine fast direkte Eltern-Kind-Beziehung, da auf Funktionen nur ein Aufruf, keine anderen Operationen gestattet sind. Für Variablen können die dazwischen liegenden Knoten deutlich komplexer werden und erschweren hierdurch die Zuordnung.

In dem folgenden kurzen Beispiel wird dies für die Bedingung der If-Anweisung bereits deutlich. (Listing 3.1 enthält den Programmquelltext, Listing 3.2 die leicht gekürzte Ausgabe des Clang-AST-Parsers, Abbildung 3.1 deren grafische Repräsentation)

```
1 int test() {
2   int a,b;
3   if( ( b *= 5, ++a ) ) {}
4 }
```

Listing 3.1: Quelltext für AST-Beispiel

[2] Abstract Syntax Tree

```
1  TranslationUnitDecl
2  `-FunctionDecl test 'int (void)'
3    `-CompoundStmt
4      |-DeclStmt
5      | |-VarDecl used a 'int'
6      | `-VarDecl used b 'int'
7      `-IfStmt
8        |-ImplicitCastExpr '_Bool' <IntegralToBoolean>
9        | `-ImplicitCastExpr 'int' <LValueToRValue>
10       |   `-ParenExpr 'int' lvalue
11       |     `-BinaryOperator 'int' lvalue ','
12       |       |-CompoundAssignOperator 'int' lvalue '*='
13       |       | |     ComputeLHSTy='int' ComputeResultTy='int'
14       |       | |-DeclRefExpr 'int' lvalue Var 'b' 'int'
15       |       | `-IntegerLiteral 'int' 5
16       |       `-UnaryOperator 'int' lvalue prefix '++'
17       |         `-DeclRefExpr 'int' lvalue Var 'a' 'int'
18       |-CompoundStmt
```

Listing 3.2: Parserausgabe für AST-Beispiel

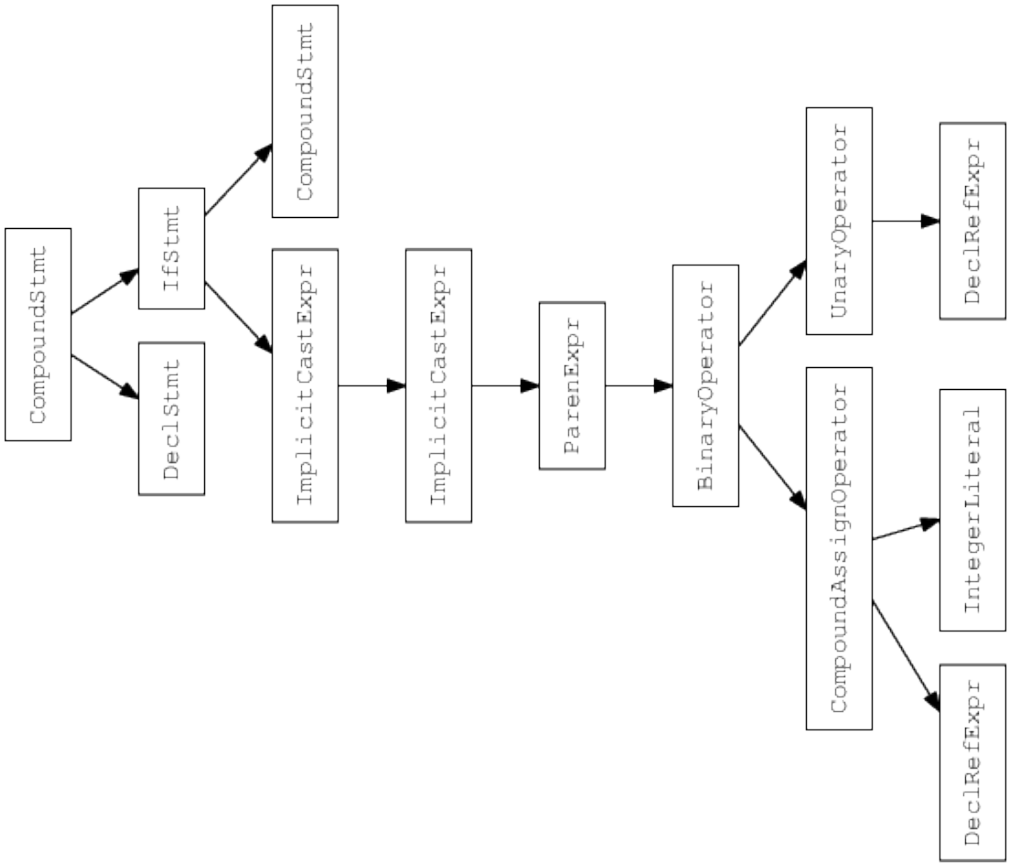

Abbildung 3.1.: Aufbau des AST im Beispiel

Es ist zu erkennen, dass Clang, welches auch im Weber als Parser verwendet wird, den Übergang von einem LValue zu einem RValue explizit modelliert (Zeile 9). Es stellt daher kein Problem dar, den Get-Joinpoint zu erkennen. Doch die Variable, um die es an diesem Joinpoint geht, ist schwerer zu finden. Sie findet sich sobald man dem Baum basierend auf der Semantik der einzelnen Knoten folgt (letzte DeclRefExpr in Zeile 17, in der Grafik unten rechts).

Diese Informationen konnten über einen Stack zusammengeführt werden, der während der Traversierung des Syntaxbaumes die notwendigen Informationen vorhält, so dass, sobald diese vollständig sind, der Joinpoint im Modell erzeugt werden kann. Dies wurde notwendig, da keine Möglichkeit besteht andernfalls auf die Elternknoten im Syntaxbaum zu schließen. Auch wenn es hier am Beispiel eines Get-Joinpoints veranschaulicht wurde, eignet sich dieses Vorgehen nicht nur gleichfalls für die anderen Joinpointtypen, sondern ist genauso erforderlich. Generell können die Joinpoint-Definitionen sehr gut auf Knoten im AST abgebildet werden. Somit kann die Verarbeitung des AST mit Hilfe der geschaffenen Datenstruktur problemlos erfolgen. Einzige Ausnahme bildet hier der Conditional Operator. Folgender Code ist nicht zur Übersetzungszeit auflösbar, da er vom Laufzeitwert von “var” abhängt.

```
1 int a,b;
2
3 ( var ? a : b ) = 4;
```

Der vorliegende Set-Joinpoint kann daher nicht korrekt erkannt und behandelt werden. In einzelnen Fällen ist dies durch tiefer gehende Analyse oder eine Transformation in zwei (oder auch mehrere) Joinpoints handhabbar, doch im allgemeinen ist dies nicht möglich.

3.5.2. Pointcut Ausdrücke

Die Umsetzung der neuen Pointcut-Ausdrücke stellte kein Problem dar. Auf Grund der bereits bestehenden, gut erweiterbaren Strukturen, die hier zum Einsatz kommen, konnte die Umsetzung mit minimalen Eingriffen im Parser und einer einfachen Ergänzung, der für die Auswertung zuständigen Klassen erfolgen. Bei den grundlegenden MatchExpressions musste nur die Unterstützung des static Schlüsselwortes auf Variablen adaptiert werden. Die neuen Pointcut-Funktionen konnten auf Grund ihrer identischen Struktur leicht analog zu bestehenden umgesetzt werden.

3.5.3. Einweben der Advice

Auch das Einweben der Advice konnte zu großen Teilen durch Generalisierung der bestehenden Funktionalität für Call-Joinpoints erreicht werden. In allen Fällen wird die eigentliche Operation (sei es ein Funktionsaufruf, ein Lese- oder Schreibzugriff etc.) durch den Aufruf einer Wrapperfunktion am Joinpoint ersetzt. Diese initialisiert dann das TJP Objekt für die Advice und ruft diese auf, bevor bzw. nach dem der Fortsetzungs-Code,

welcher die ursprüngliche Operation umsetzt, zur Ausführung kommt. So ergab sich neben der Generalisierung nur die Notwendigkeit für geringe Anpassungen um die neuen Joinpointtypen zu unterstützen. Zum einen musste die Transformation der Parameter der ursprünglichen Operation zu denen des Wrapperaufrufs leicht angepasst werden. Zum anderen musste natürlich der Fortsetzungs-Code um die neuen Arten von Operationen erweitert werden.

Die Unterstützung für die neue Funktionalität der Joinpoint-API (entity) konnte durch entsprechende Modifizierung der generierten TJP Struktur und deren Initialisierung ohne besondere Probleme erreicht werden. Bei den Tests für verschiedene Szenarien ergaben sich jedoch zwei bedeutende Probleme bei der Codetransformation.

Die Reihenfolge, in der die einzelnen Transformationsschritte durchgeführt werden, spielt eine große Rolle, wenn es darum geht sicherzustellen, dass der erzeugte Quellcode auch korrekt ist. Dies trifft insbesondere auf geschachtelte Ausdrücke zu, denn es müssen Einfügungen an der selben Zeichenposition im Quelltext erfolgen. Wird hier die Reihenfolge nicht beachtet, kann es zu einer Vermischung der Teile der einzelnen Wrapperaufrufe kommen. Der so entstehende Quellcode ist in der Regel nicht mehr korrekt.

Fast alle Teile des Weber erhalten hier die Reihenfolge der Joinpoints, wie sie im Quelltext auftauchen. Es wurden jedoch bei Testläufen im generierten Quellcode Fehler festgestellt, die auf genau dieser Vermischung beruhen. Die Ursache hierfür konnte nicht wie erwartet in der Ausführung des Webvorgangs an sich gefunden werden. Dieser erwies sich als korrekt. In der Planungsphase, d.h. dem Abgleich mit und der Zuordnung der Advice zu den einzelnen Joinpoints, kommt es jedoch zu einer Änderung der Joinpointreihenfolge. Diese beruht darauf, dass die Joinpoints gruppiert nach ihrem Typ verarbeitet werden. Eine einfache Anpassung konnte hier nicht erfolgen, da die notwendigen Strukturen für die Planung recht komplex sind. Vorläufig konnte das Problem gelöst werden, in dem durch nachträgliche Sortierung die ursprüngliche Reihenfolge zwar nicht wiederhergestellt werden konnte, aber eine vorhersagbare Ordnung erreicht wird, welche korrektes Weben ermöglicht.

Das andere Problem ist zwar ähnlich gelagert, musste aber strukturell anders gelöst werden. Auf Grund der Joinpointdefinitionen ergeben sich zum Beispiel für das Inkrement einer Variablen (“a++”) zwei Joinpoints (jeweils ein Get und ein Set), die an exakt der selben Stelle eingewoben werden müssen. Dieses Problem lässt sich nicht über die Reihenfolge der Webvorgänge lösen, denn um eine korrekte Modifikation zu erreichen muss beim Weben des einen Joinpoints bekannt sein, ob bei dem anderen ebenfalls ein Webvorgang stattfindet oder bereits stattfand. Diese Beziehung lässt sich aber aus dem bestehenden Modell nicht ableiten. Da zu erwarten war, dass dieses Problem durch die Zusammenführung mit der parallel von Simon Schröder entwickelten Version für Advice an vordefinierten Operatoren noch verstärkt wird, musste eine allgemeine Lösung gefunden werden.

3.5.4. Integration vordefinierter Operatoren: geschachtelte Joinpoints

Das Modell und die AST Verarbeitung wurden so angepasst, dass diese Beziehung explizit modelliert werden kann. Es erwies sich als vorteilhaft einen Joinpoint aus dieser Gruppe hervorzuheben und im Rahmen dessen Einwebvorganges, alle notwendigen Transformationen durchzuführen. Hier bot sich der Call-Joinpoint für den Operator an, da dieser immer die anderen Joinpoints impliziert. So liegt bei einem Inkrement immer ein Get sowie ein Set vor, an einer Zuweisung nur ein Set etc. Bei einer anderen Wahl ist diese Zuordnung nicht so eindeutig. Um keinen doppelten Entwicklungsaufwand zu betreiben und ein späteres Zusammenführen nicht zu verkomplizieren, wurde daher die Version von Simon Schröder bereits zu diesem Zeitpunk integriert. Dies erwies sich als aufwendiger als erwartet, da es durch die parallele Entwicklung an den Kernkomponenten und insbesondere durch die umfangreichen Codeänderungen im Rahmen der Generalisierung zu erheblichen Konflikten bei der Zusammenführung kam. Anpassungen für die Behandlung der vordefinierten Operatoren mussten in die generalisierte Version übernommen, sowie neue Generalisierungen geschaffen werden. Dies gelang jedoch, so dass eine einheitliche Basis für die Lösung des eigentlichen Problems geschaffen werden konnte.

Durch die Unterordnung der weiteren Joinpoints unter den Call-Joinpoint bei der Modellierung ist es nun möglich den Wrapper für diesen einzuweben und im Fortsetzungs-Code des Joinpoints die Ausführung der Advice für die implizierten Joinpoints sicherzustellen. Die Operatoren werden dazu in eine generische Form mit Zwischenwerten übertragen um so alle Advice einweben zu können. Deutlich wird dies im folgenden Pseudocode-Beispiel. Der Ausdruck “a++” mit a vom Typ “int” wird transformiert zu:

```
1  int __wrap_get() {
2    // Init TJP for get
3    __advice_before_get;
4    result = a;
5    __advice_after_get;
6    return result;
7  }
8  void __wrap_set( int newValue ) {
9    // Init TJP for set
10   __advice_before_set;
11   a = newValue;
12   __advice_after_set;
13   return result;
14 }
15 int __wrap_operator++( int &arg ) {
16   // Init TJP for operator call
17   __advice_before_call;
18   int tmp = __wrap_get();
19   result = tmp;
20   tmp++;
21   __wrap_set( tmp );
22   __advice_after_call;
```

```
23    return result;
24  }
25  __wrap_operator++( a );
```

Der Pseudocode wurde hier auf das wesentliche beschränkt, da der generierte Code auf Grund der notwendigen Templates, der Eindeutigkeit der Namen sowie anderen Details erheblich unübersichtlicher und schwerer verständlich ist. Es ist jedoch gut zu erkennen, dass der Einfachheit und klaren Trennung der Joinpoints halber die bestehenden Wrapper weiterverwendet werden. Der Fortsetzungs-Code(Zeilen 18-21) wurde in eine Version mit Zwischenergebnissen überführt, um so die geschachtelten Joinpoints integrieren zu können.

Die so geschaffene Struktur zur Modellbildung und zum Weben hat sich als tauglich erwiesen, das vorliegende konkrete Problem zu lösen, verspricht aber auch die leichte Integration späterer Erweiterungen. Die klare Gliederung entspricht den Strukturen des abstrahierten Quelltextes und fügt sich nahtlos in bestehendes ein. Auch wird durch die explizite Umsetzung im Modell, die Beziehung zwischen den Joinpoints besser deutlich, was für eine externe Verwendung der Modellinformationen wichtig ist. Daher stellt der verwendete Ansatz eine nützliche konzeptionelle Erweiterung der jeweiligen Komponenten des Webers dar.

3.6. Erweiterungen für Membervariablen

Membervariablen wurden in den bisherigen Betrachtungen bewusst ausgeklammert, da sie einige Besonderheiten aufweisen. Sie stellen jedoch einen wichtigen Teil moderner Softwarearchitektur dar, so dass es im folgenden um die notwendigen Anpassungen zu ihrer Unterstützung bei Datenzugriffen gehen wird.

3.6.1. Joinpointdefinition, API & Matching

Der Kernpunkt bisher, die Definitionen der Joinpoints, ist für Member genau so verwendbar wie für globale Variablen, da diese auf das Datenobjekt abzielen. Das Datenobjekt von Membervariablen ist zwar Teil einer größeren Struktur und ist daher an diese gebunden. Es ist aber auch eigenes gekapseltes Objekt und kann daher als hinreichend eigenständig bezeichnet werden um unseren Definitionen zu entsprechen. Bestes Argument hierfür ist die Möglichkeit Verweise auf selbiges zu erzeugen, die unabhängig vom Elternobjekt sind, und wie jeder andere Verweis des entsprechenden Typs verwendet werden können.

Für Membervariablen kommt in der Joinpoint-API die target()-Funktion ins Spiel. Diese bildet analog zu ihrer Verwendung bei Member-Funktionen in Call-Joinpoints das (umgebene) Bezugsobjekt ab. Neben der entity()-Funktion, die keiner Einschränkung in der Nutzbarkeit unterliegen sollte, bietet es sich an dem Programmierer einen Memberpointer zur Verfügung zu stellen, mit dem in Verbindung mit dem Target-Pointer ebenfalls das Datenobjekt referenziert werden kann. Es wurde entschieden dies im Rahmen

einer neuen member()-Funktion in der API zu realisieren, so dass beide Wege offenstehen und je nach den Erfordernissen der günstigere gewählt werden kann.

Da die Signatur einer Membervariablen identisch zu der einer globalen Variablen im entsprechenden Namespace ist, sind für die Verwendung der MatchExpressions keine weiteren Definitionen erforderlich. Dies trifft genauso auf die nun bedeutungsvolle target()-Pointcutfunktion zu, deren Definition nicht angepasst werden muss, da sie 1:1 auch auf die Datenjoinpoints für Membervariablen zutrifft.

3.6.2. Anpassungen im Weber

Auch die notwendigen Änderungen im Weber sind überschaubar. Für die Modellbildung und das Matching müssen nur die Informationen über die Zielklasse an den Joinpoints ergänzt werden. Dies erforderte auf Grund der bereits erfolgten Generalisierung des bestehen Codes nur minimalen Aufwand.

Der Vorgang des Einwebens an sich musste ebenfalls nicht wesentlich verändert werden. Da der Zugriff auf nicht-öffentliche Member nicht direkt erfolgen kann, muss hier in jedem Fall auf die Referenz ausgewichen werden, welche dem Wrapper übergeben wird. Es ergibt sich jedoch ein gewichtiges Problem beim Einweben des Wrapper-Aufrufs.

Damit diese für das TJP-Objekt und damit die Joinpoint-API zu Verfügung stehen, müssen sowohl die Referenz auf die Variable wie bisher, als auch eine Referenz auf das Zielobjekt übergeben werden. Diese sind jedoch beide Teil eines gemeinsamen Ausdrucks. Für Memberfunktionen ist dieser wenig komplex und kann daher aufgespalten werden um so die Referenz auf das Zielobjekt übergeben zu können. Bei Variablen jedoch kann dieser eine größere Komplexität entfalten. Dies verhindert ein analoges Vorgehen, da sonst Operationen in diesem Ausdruck entfallen könnten und hierdurch die Semantik des Programms geändert würde. Eine andere Möglichkeit wäre es den Teilausdruck, der das Zielobjekt bestimmt zu kopieren und separat zu übergeben. Dem entgegen steht die Tatsache, dass dieser Seiteneffekte haben kann. Im Falle einer Kopie kommen diese doppelt zum tragen, wodurch ebenfalls die Programmsemantik geändert würde.

Es konnte jedoch eine Möglichkeit gefunden werden, die es zwar nicht erlaubt beide Referenzen zu übergeben, jedoch die Referenz auf das Zielobjekt wiederherzustellen. Wird diese im Rahmen der TJP-Struktur benötigt, kann sie aus der Variablenreferenz mit Hilfe weiterer statisch bekannter Informationen gewonnen werden, da beide in einer engen und klar definierten Beziehung zu einander stehen. Für Member virtueller Basisklassen ist diese Bindung aber nur zu der sie deklarierenden Klasse eng genug, so dass in diesen Fällen von der bestehenden Konvention abgewichen werden musste. In diesen seltenen Fällen steht daher nur das Zielobjekt der Basisklasse zur Verfügung, nicht jedoch, wie im allgemeinen Fall, das im Original-Code verwendete.

3.7. Zusammenfassung & Bewertung

In diesen ersten beiden Schritten wurden die beiden neuen Joinpoint-Arten Get und Set definiert und die Möglichkeit geschaffen diese in Verbindung mit einer grundlegenden

Auswahl an Variablen zu nutzten. Auch wenn damit noch nicht der volle Sprachumfang von C++ unterstützt wird, lassen sich schon jetzt viele Anwendungsfälle abdecken, insbesondere jene, welchen das Observer-Pattern zu Grunde liegt.

Ein Beispiel bildet hier die General Objekt Protection[BSS13]. Wie im folgenden Aspekt zu sehen ist, konnte die Absicherung auch auf die öffentlichen Variablen ausgedehnt werden, so dass auf die Abwägung zwischen Entfallen der Absicherung für diese und einer umfangreichen Code-Modifikation im Zielprogramm verzichtet werden kann. (Der Advice kann hier aus Gründen der Übersichtlichkeit nur in einer gekürzten Version wiedergegeben werden. Das Beispiel wurde daher auf die wesentlichen neuen Sprachelemente und ihre bisherigen Entsprechungen beschränkt.)

```
1  #include <iostream>
2  using namespace std;
3
4  //-----------------------------------------------------
5  // compare two pointers
6  __attribute__((always_inline)) inline
7  const bool EqualPointers
8      ( const void* const a, const void* const b ) {
9    return a == b;
10 }
11
12 // compare two types
13 template<typename T, typename U>
14 struct TypeTest { enum { EQUAL=0 }; };
15 template<typename T>
16 struct TypeTest<T,T> { enum { EQUAL=1 }; };
17 //-----------------------------------------------------
18
19 aspect GenericObjectProtection {
20   pointcut critical() = "A" || "B";
21   // the the shortcut above is not yet implemented for vars,
22   // as it is likely ambiguous
23   pointcut criticalV() = "% A::%" || "% B::%";
24
25   // static function calls
26   // (from outside of the particular class)
27   advice
28     call( critical() ) && call( "static % ...::%(...)" )
29   : before() {
30     // static checks (different types only)
31     if( ! TypeTest<TJP::That, TJP::Target>::EQUAL ) {
32       cout << "static check: " << tjp->signature() << endl;
33     }
34   }
35
36   // non-static calls
37   // (from outside of the particular class)
38   advice
39     call( critical() ) && ! call( "static % ...::%(...)" )
40   : before() {
```

```
41      // static checks (different types or different objects)
42      if( ! TypeTest<TJP::That, TJP::Target>::EQUAL ||
43          ! EqualPointers(tjp->that(), tjp->target())  ) {
44        cout << "check: " << tjp->signature() << endl;
45      }
46    }
47
48  // get advice (set advice similarly):
49
50    // static member access
51    // (from outside of the particular class)
52    advice
53      get( criticalV() ) && get( "static % ...::%")
54    : before() {
55      // static checks (different types only)
56      if( ! TypeTest<TJP::That, TJP::Target>::EQUAL ) {
57        cout << "static check (get): " << tjp->signature() << endl;
58      }
59    }
60
61    // non-static member access
62    // (from outside of the particular class)
63    advice
64      get( criticalV() ) && ! get( "static % ...::%" )
65    : before() {
66      // static checks (different types or different objects)
67      if( ! TypeTest<TJP::That, TJP::Target>::EQUAL ||
68          ! EqualPointers(tjp->that(), tjp->target()) ) {
69        cout << "check (get): " << tjp->signature() << endl;
70      }
71    }
72  };
```

Listing 3.3: Aspekt für die General Objekt Protection

Wir erkennen das die bestehenden Advice direkt auf die neue Funktionalität übertragen werden konnten. Insbesondere da sich die Strukturen und Interfaces nicht ändern. Lediglich bei der selbst definierten Pointcut-Funktion in Zeile 20 war dies nicht möglich, da die Kurzform der Ausdrücke noch nicht unterstützt wird.

Aus Anwendersicht haben daher sowohl die Definitionen auf der Sprachebene als auch deren Umsetzung im AspectC++-Weber ihre erste Bewährungsprobe bestanden. Es ist zu erwarten, dass sich bei der Umsetzung anderer Anwendungsfälle ein ähnliches Bild ergibt. Aber auch aus der Sicht der Softwareentwicklung können die Änderungen am Weber bestehen. Bestehender Code wurde generalisiert, so dass ein hoher Grad an Wiederverwendung erreicht werden konnte aber gleichzeitig auch eine gute Basis für Erweiterungen geboten wird, wie am Beispiel der Unterstützung von Membervariablen zu sehen war.

Ein wichtiges Sprachelement von C++ sind jedoch die Zugriffe über Zeiger und Referenzen. Die mangelnde bisherige Unterstützung stellt daher einen gewichtigen Nachteil dar, so dass es naheliegend ist, sich mit dieser Problematik als nächstes zu beschäftigen. Hierbei kann sowohl auf die neuen Definitionen der Sprachebene als auch auf die ver-

besserten Strukturen der Weberumsetzung zurückgegriffen werden, so dass eine Lösung nicht mehr so fern wie zu Beginn erscheint.

4. Die Alias-Problematik

Durch die Existenz von Referenztypen in C++ entsteht ein Problemfeld, dass im allgemeinen Alias-Problem oder auch Points-To-Problem genannt wird. Bei der Analyse und Verarbeitung von C++-Code stellt sich häufig die Frage, ob ein Pointer oder eine Referenz auf eine bestimmte Variable oder auf die selbe Variable wie ein anderer Verweis zeigen. Auf Grund der dynamischen Natur dieser Verweise, ist diese Frage an einem konkreten Punkt der Programmlaufzeit zwar eindeutig beantwortbar, aber im allgemeinen zur Übersetzungszeit nicht entscheidbar[Hin01, Kapitel 1]. Durch Analyse des Programms können zwar Erkenntnisse gewonnen werden, welche die Menge der möglichen Ziele eines Verweis einschränken, doch kann in vielen Fällen keine Eindeutigkeit erreicht werden. Dies betrifft häufig nur Optimierungsentscheidungen, wirkt sich allerdings für den AspectC++-Weber auf bedeutende Weise aus.

Das Einweben der Advice erfordert es zur Übersetzungszeit eine Entscheidung über die Erfüllung des gegebenen Pointcut-Ausdrucks zu treffen, oder alternativ aus diesem eine zur Laufzeit prüfbare Bedingung abzuleiten und den Advice nur bedingt auszuführen. Für einen Datenjoinpoint dessen Ziel durch einen Verweis bestimmt wird, kann der Weber nun diese Entscheidung nicht treffen, da die Zielvariable noch nicht bekannt ist. Zur Laufzeit ist die Entscheidung ebenfalls problematisch. Es liegt zwar der Verweis auf das Datenobjekt vor, doch fehlen die Informationen über die Variable. Diese notwendigen Informationen können auch nicht ohne weiteren Aufwand aus dem Verweis abgeleitet werden. Doch auch die Joinpoint-API ist betroffen, da sie normalerweise Informationen bereit stellt, die im konkreten Fall am Joinpoint selbst nicht verfügbar sind. Ein Beispiel ist hier die Signatur, welche normalerweise statisch bestimmt werden kann. Ein anderes ist das Zielobjekt. Der Typ lässt sich ebenfalls statisch bestimmen und die Referenz kann zur Laufzeit ermittelt werden. Basiert der Joinpoint nun aber auf einem Verweis, entfällt diese Informationen sowohl für den Typ als auch die Referenz.

Die Problematik äußert sich also primär in der Nichtverfügbarkeit von Informationen an den entscheidenden Stellen. Generell sind diese Informationen zu anderen Zeitpunkten verfügbar. So stünden sie beispielsweise beim Erzeugen des Verweis, genauso wie an anderen Joinpoints, zur Verfügung.

4.1. mögliche Lösungen

Die wünschenswerteste Lösung wäre natürlich eine vollständige Auflösung zur Übersetzungszeit. Dies ist aber nicht in allen Fällen möglich, da der Programmablauf von Eingaben und sonstigen Laufzeitparameter abhängt. Ziel einer Lösung muss es also sein, die notwendigen Information an einem Punkt ihrer Verfügbarkeit zu sammeln und sie

einer Laufzeitprüfung am eigentlichen Joinpoint zur Verfügung zu stellen. Als Quellort bietet sich hier die Erzeugung des Verweises an, denn an diesem wären die Information analog zu anderen Joinpoints verfügbar und es ist garantiert, dass dieser Punkt vor dem Bedarf an den Informationen durchlaufen wird. Es bleibt also die Frage wie diese Information am Joinpoint zur Verfügung gestellt werden können.

Eine Möglichkeit wäre die Informationen zusammen mit dem Verweis vom der Erzeugung zum Joinpoint durchzureichen. Innerhalb der bisherigen Verweisdaten wäre dies nicht möglich, da diese keine garantiert ungenutzten Bereich erhalten. Es müsste also der Umfang und damit der Datentyp angepasst werden. Dies würde sehr umfangreiche Modifikationen des Zielquellcodes erfordern, da sämtliche Zugriffe auf den modifizierten Verweis angepasst werden müssten.

Die Speicherung der Informationen in einer Mappping-Datenstruktur an der Quelle und anschließende Suche in dieser am Joinpoint verspricht hier ein deutlich besseres Aufwand / Nutzen Verhältnis. Als Zuordnungsschlüssel kann hier der Verweis selbst dienen. Die Code-Modifikationen würden sich auf zwei klar definierte Punkte beschränken. Es entsteht zwar ein gewisser Overhead durch die Operationen auf der Datenstruktur, doch dieser dürfte sich im beherrschbaren Rahmen bewegen, wenn die Struktur entsprechend gewählt wird. Von daher fällt auch diese Abwägung zu Gunsten einer Store-and-Lookup-Lösung aus, denn auch das Durchreichen der Informationen würde auf Grund der erhöhten Datenmenge einen erhöhten Aufwand bedeuten. Dieser wäre sogar deutlich schwerer abzuschätzen, da der Overhead von der Zahl der modifizierten Punkte abhängt.

Da noch unklar ist wie eine solche Lösung im Detail aussehen kann und im Interesse der frühen Verfügbarkeit von Features zu Testzwecken, bietet es sich an als nächsten Schritt die notwendigen Sprachelemente einzuführen und ihre Unterstützung im Weber umzusetzen. Dies ermöglicht es verschiedene Konzepte zu testen, da zu erwarten ist, dass eine hohe Anpassung an den Advice sowie das Zielprogramm erfolgen kann. In einem zukünftigen Schritt kann dann basierend auf den hiermit gemachten Erfahrungen eine Integration in den Weber sowie die Nutzung von Ergebnissen aus einer statischen Analyse untersucht werden.

4.2. Notwendige Erweiterungen auf der Sprachebene

Die erste notwendige Erweiterung stellt ein Joinpoint an der Stelle der Erzeugung eines Verweises dar, um so die notwendigen Informationen hinterlegen zu können. Hierbei kann auf Grund der großen Ähnlichkeit zu einem Get-Joinpoint dieser wie folgt definiert werden:

Definition 3. Ein Ref-Joinpoint bildet die Erzeugung eines Verweises auf das Datenobjekt einer Variablen ab. Dabei berücksichtigt er sowohl die explizite Erzeugung von Pointern, als auch die implizite Erzeugung bei der Initialisierung einer Referenz.

Diese Definition abstrahiert von den beiden Arten an Verweisen um eine einheitliche Nutzung zu ermöglichen und gliedert sich sinnvoll in die Menge der bestehenden Joinpoints ein, da sie ähnlich wie get und set eine Klasse von Operationen auf einer Variablen

abdeckt. Die Joinpoint-API und die Pointcut-Ausdrücke können ebenfalls analog vom Get-Joinpoint übernommen werden.

Als weiteres Erfordernis ergibt sich, jene Joinpoints auswählen zu können, welche auf einem Verweis basieren, der potentiell auf eine Variable von Interesse zeigt. Hierzu bietet sich eine Pointcut-Funktion an, die analog zu den Vererbungsfunktionen base() und derived() die gewünschte Menge liefert.

Definition 4. Die alias()-Pointcutfunktion liefert im Kontext eines Datenjoinpoints die Menge aller Verweise auf die gegebene Menge an Variablen. Sie dehnt somit die Menge der ausgewählten Datenjoinpoints um jene aus, welche einen Verweis verwenden, der potentiell eine der gegebenen Variablen zum Ziel hat.

Diese Definition fügt sich in die bestehende Struktur gut ein und ermöglicht so eine einfache Anpassung an die neue Funktionalität. Sie greift in sofern etwas vor, als dass die Auswahl der Joinpoints auch direkt durch den Programmierer, basierend auf den in Frage kommenden Referenztypen, erfolgen könnte. Sie ist jedoch vorzuziehen, da sie das größere Erweiterungspotential besitzt. So würden sich eine spätere automatische Unterstützung der Alias-Auflösung sowie eine Einschränkung der möglichen Kandidaten durch statische Analyse beides ohne weitere Anpassungen auf der Sprachebene umsetzen lassen. Gleichzeit bietet die explizite Anforderung der Ausdehnung dem Programmierer die Möglichkeit auf diese zu Verzichten, falls dies erforderlich sein sollte.

4.3. Unterstützung durch den Weber

Die Unterstützung für den Ref-Joinpoint konnte auf Grund der Ähnlichkeit zum Get-Joinpoint und der bereits geschaffenen strukturellen Basis schnell und einfach umgesetzt werden. Einzig die Tatsache, dass zwar die explizite Form am AddrOf-Operator ansetzen kann, aber für die implizite Form keine solche Entsprechung gefunden werden konnte, hat den Aufwand etwas erhöht. Doch auch für die implizite Bildung eines Verweises bei der Initialisierung einer Referenz konnte schließlich die Menge der zu betrachtenden AST-Knoten auf eine handvoll reduziert werden.

Für die Unterstützung von auf Verweisen basierenden Daten-Joinpoints waren größere Anpassungen erforderlich. Diese betrafen weniger die Verarbeitung des AST, denn hier erwiesen sich die geschaffenen Strukturen als tauglich, sondern die Modellierung. Auf Grund von Umfang und Art der verfügbaren Informationen, sowie der Ableitung selbiger aus unterschiedlichen Quellen, war es erforderlich Varianten der existierenden Modellklassen zu schaffen. Diese teilen sich zwar eine gemeinsame Basis, aber bei Verweisbasierten muss beispielsweise der Typ aus dem des Verweises abgeleitet werden, während er bei direkten Joinpoints zusammen mit allen anderen Informationen der Deklaration entnommen werden kann. Weiterhin war es notwendig die fehlenden Informationen über die Zielvariable an allen Stellen ihrer Verwendung zu berücksichtigen bzw. auf Dummy-Werte zurückzugreifen. So wird beispielsweise bei der Generierung der Signatur der Typ wie bisher verwendet, doch das Symbol durch ein “<unknown>” ersetzt.

Mit diesen Änderungen war es dann möglich die Auswertung der Pointcut-Ausdrücke um die alias()-Funktionaliät zu erweitern. Hier ist zu beachten, dass in dieser ersten Umsetzung die Zuordnung bisher rein auf Typ-Basis erfolgen kann, da noch keine weiteren Informationen vorliegen. Es bleibt also einer späteren Erweiterung überlassen diese Zuordnung zu präzisieren und gegebenenfalls Laufzeitbedingungen zu generieren.

4.4. Konzepte für alias-gewahre Advice

Mit den erfolgten Änderungen ist es dem Programmieren nun möglich seine Advice um eine Berücksichtigung der Alias-Problematik zu erweitern. Es soll nun um Konzepte gehen, die er dazu nutzen kann, um so eine gewisse Orientierungsmöglichkeit zu schaffen. Hierbei werden wir uns zunächst den Informationen widmen, die verfügbar gemacht werden müssen und wie man diese speichern kann. Anschließend wird ein einfacher Advice prototypisch angepasst um eine bessere Vorstellung von den Erfordernissen zu erhalten.

4.4.1. Informationen von Interesse und ihre Repräsentation

In Frage kommen alle Information, die in Zusammenhang mit der Zielvariable stehen und sich nicht aus dem Joinpoint ergeben. Im wesentlichen sind dies die Signatur und die Informationen im Zusammenhang mit Membern. Die Repräsentation hängt nun von den wirklich benötigten Informationen und deren Verwendung ab.

Wird nur die Signatur benötigt, kann diese gespeichert werden und so bei Bedarf verwendet werden. Sind die Informationen nur für die bedingte Ausführung des Joinpoints notwendig, hängt es davon ab ob für die Prüfung sowohl Information über die Variable als auch den Joinpoint notwendig sind. Besteht nur Bedarf an ersteren kann die Prüfung bereits bei der Erzeugung des Verweises erfolgen und es reicht, dass Ergebnis vorzuhalten. Ist dies nicht der Fall müssen die Informationen in Form einer ID oder ähnlichem codiert werden, um so die Prüfung am Joinpoint zu ermöglichen.

Die Informationen in Verbindung mit Membern sind hier kritischer und bedeuten einen erhöhten Aufwand, da sie es erfordern Typen zu übertragen, die in C++ keine Laufzeitdarstellung besitzen. Die Referenz auf das umgebene Objekt kann gespeichert werden, der Typ dessen jedoch nicht ohne weiteres. Werden nur Informationen über den Typ benötigt reicht es diese passend zu kodieren. Wird jedoch im Advice der Typ selbst benötigt, muss durch Templates oder ähnliche Methoden eine Abstraktion sowie eine Fallunterscheidung über alle in Frage kommenden Typen erfolgen. Ähnlich verhält es sich mit dem Memberpointer.

4.4.2. Prototypische Advice-Adaption

Als Beispiel soll hier ein einfacher Tracing-Aspekt, der alle Zugriffe auf eine Auswahl an Variablen protokolliert, genutzt werden. Neben den eigentlichen Inhalten muss ein

Aspekt um alias-basierende Joinpoints zu unterstützen, Aufgaben aus zwei Bereichen erfüllen. Erstens an den primären Joinpoints die fehlenden Informationen wiederherstellen. Zweitens erfordert dies die notwendigen Datenstrukturen und Informationen zu pflegen.

```
1  #ifndef __TRACING_ALIAS_AH__
2  #define __TRACING_ALIAS_AH__
3
4  #include <iostream>
5  #include <map>
6
7  using namespace std;
8
9  class T1;
10
11 template<typename C>
12 const unsigned int classID() { return 0; };
13 template<> const unsigned int classID<T1>() { return 1; };
14
15 template<typename C>
16 const char *className() { return "<unknown>"; };
17 template<> const char *className<T1>() { return "T1"; };
18
19 class EntityMap {
20 public:
21   struct Entry {
22     void *ent;
23     const char *sig;
24     void *target;
25     unsigned int target_type;
26
27     Entry( void *_ent, const char *_sig,
28         void *_dst = 0, unsigned int _dst_type = 0 ) :
29       ent( _ent ),
30       sig( _sig ),
31       target( _dst ),
32       target_type( _dst_type )
33       {};
34   };
35 private:
36   typedef map<void *, Entry *> Map;
37   Map _entmap;
38 public:
39   void remember( const Entry &e ) {
40     Entry *entry = new Entry( e );
41     _entmap.insert( Map::value_type( entry->ent, entry ) );
42   }
43
44   void clear( void *start, size_t length ) {
45     for( Map::iterator it = _entmap.begin();
46         it != _entmap.end(); it++ ) {
47       void *ent = it->first;
48       if( ent >= start
49         && ent < ((void *) ( (size_t)start + length ) ) ) {
```

```
50            delete it->second;
51          _entmap.erase( it );
52        }
53      }
54    }
55
56    Entry *lookup( void *ent ) {
57      Map::iterator it = _entmap.find( ent );
58      if( it != _entmap.end() )
59        return it->second;
60      else
61        return 0;
62    }
63  } entmap;
64
65  aspect VarTracer {
66    pointcut vars() = "% ...::%";
67
68    advice ref( vars() ) && ! within( "EntityMap" ) : after() {
69      entmap.remember( EntityMap::Entry(
70        tjp->entity(),
71        tjp->signature(),
72        tjp->target(),
73        classID<JoinPoint::Target>() ) );
74    }
75
76    advice destruction( "T1" ) : after () {
77      entmap.clear( tjp->that(), sizeof( JoinPoint::That ) );
78    }
79
80    template<typename TJP, typename Target>
81    void handle_get( TJP *tjp, const char *signature,
82        Target *target ) {
83      cout << "Get: " << signature;
84      if( target ) {
85        cout << " (" << className<Target>()
86             << ": " << target << ")";
87      }
88      cout << " = " << *tjp->result() << endl;
89    }
90
91    template<typename TJP, typename Target>
92    void handle_set( TJP *tjp, const char *signature,
93        Target *target ) {
94      cout << "Set: " << signature;
95      if( target ) {
96        cout << " (" << className<Target>()
97             << ": " << target << ")";
98      }
99      cout << " = " << *tjp->template arg<0>() << endl;
100   }
101
```

```
102   advice get( vars() ) && ! within( "EntityMap" ) : after() {
103     tjp->result(); // generate functions
104     handle_get( tjp, tjp->signature(), tjp->target() );
105   }
106
107   advice get( alias( vars() ) ) && ! within( "EntityMap" ) : after() {
108     tjp->result(); // generate functions
109     EntityMap::Entry *info = entmap.lookup( tjp->entity() );
110     if( info ) {
111       if( info->target_type == 0 )
112         handle_get( tjp, info->sig,
113           static_cast<void *>( info->target ) );
114       else if( info->target_type == classID<T1>() )
115         handle_get( tjp, info->sig,
116           static_cast<T1 *>( info->target ) );
117     }
118   }
119
120   advice set( vars() ) && ! within( "EntityMap" ) : after() {
121     tjp->arg<0>(); // generate functions
122     handle_set( tjp, tjp->signature(), tjp->target() );
123   }
124
125   advice set( alias( vars() ) ) && ! within( "EntityMap" ) : after() {
126     tjp->arg<0>(); // generate functions
127     EntityMap::Entry *info = entmap.lookup( tjp->entity() );
128     if( info ) {
129       if( info->target_type == 0 )
130         handle_set( tjp, info->sig,
131           static_cast<void *>( info->target ) );
132       else if( info->target_type == classID<T1>() )
133         handle_set( tjp, info->sig,
134           static_cast<T1 *>( info->target ) );
135     }
136   }
137 };
138 #endif
```

Listing 4.1: Alias-aware Tracing

Als Komponenten sind zu erkennen:

- Templates zum Mapping von Typen auf IDs bzw. Namen (Z. 11-17)
- Die verwendete Datenstruktur (Z.19-63) mit
  - der Definition eines einzeln Eintrags (Z. 21-34)
  - den notwendigen Operationen (Z. 39-62)
- Der adaptierte Aspekt (Z. 65-137) mit
  - der Definition der Variablenauswahl

- den Advice zur Pflege der Daten (Z. 68-78)
- Funktionen für die eigentlichen Aufgaben (Z. 80-100)
- Advice für Zugriffe ohne alias (Z. 102-105,120-123)
- Die Advice für alias-Zugriffe (z. 107-118, 125-136)

Nun zu der Betrachtung der Komponenten im Detail. Die Mapping-Templates werden benötigt, um Typinformationen auch zur Laufzeit verarbeiten zu können. Das Name-Mapping ist durch den ursprünglichen Code bedingt, damit diesem der Name eines Typs zur Verfügung steht. Das ID-Mapping wird jedoch für die Verarbeitung der Target-Informationen zwingend benötigt. Das Beispiel kennt nur eine Klasse "T1". In der Praxis müsste dieses Mapping für alle existierenden Klassen erfolgen.

Die Datenstruktur ist soweit selbsterklärend. Wichtig sind hier die vom Aspekt benötigten Operationen remember und clear zum pflegen der Informationen, sowie lookup zum Auffinden selbiger. Hier kommt als Basis eine einfache Map der STL zu Einsatz. Für andere Anwendungsfälle müssen hier entsprechende Abwägungen getroffen werden. Zum einen steht die STL gerade im Bereich der eingebetteten Systeme nicht unbedingt zur Verfügung. Zum anderen besteht durch Anpassung an die konkreten Erfordernisse Potential zur Steigerung der Effizienz und damit der Verringerung, des durch den Aspekt erzeugten Overheads. Dem entgegen steht der zu tätigende Aufwand. Auf die Überlegungen zur Laufzeitdatenstruktur werden wir im folgenden Abschnitt 4.4.3 noch ausführlicher eingehen.

Den Kern bildet der adaptierte Aspekt. Die Auswahl der Variablen kann übernommen werden, so dass bei Aspekten, die als Vorlage entwickelt wurden, nur der Basisaspekt angepasst werden muss und die abgeleiteten Aspekte, welche die konkrete Definition treffen, keiner Veränderung bedürfen. Die ursprünglichen Advice, mussten in Template-Funktionen verlagert werden, damit ihnen in jedem Fall die benötigten Informationen zur Verfügung gestellt werden können. In den Advice für Zugriffe ohne involvierten Verweis, verbleibt damit nur der Aufruf der passenden Funktion. Die Dummy-Funktionsaufrufe am Anfang haben keinen wirklichen Zweck. Sie sind nur erforderlich, da der Weber die Funktionen des tjp-Objekts nur bei Bedarf generiert und dabei nur den direkten Advice-Code berücksichtigt.

Es bleiben also noch die neuen Advice. Zunächst sind dies diejenigen zur Pflege der Informationsdatenstruktur. Der Ref-Advice bietet sich an die jeweiligen Informationen der Struktur bekannt zu machen, da hier nicht nur die fehlenden Informationen vorliegen, sondern auch garantiert ist, dass der jeweilige Joinpoint vor dem Bedarf an selbigen durchlaufen wird. Vor dem Durchlaufen des Joinpoints, existiert der jeweilige Verweis noch nicht, kann also auch noch nicht verwendet werden und somit besteht noch kein Bedarf an den Informationen. Genau so wichtig ist jedoch die Informationen wieder aus der Struktur zur entfernen, wenn die Lebensdauer der jeweiligen Variablen endet. Da der Speicherplatz tendenziell wieder verwendet wird, besteht sonst die Möglichkeit einer falschen Zuordnung. Hier kann sich nicht auf die Annahme beschränkt werden, dass die Informationen mit denen dann gültigen überschrieben werden. Denn es ist nicht

sichergestellt, dass für die neue Variable auch Informationen hinterlegt werden, da diese nicht zwangsläufig vom Aspekt berücksichtigt wird.

Für globale Variablen kann dies entfallen, da das Ende ihrer Lebensdauer mit dem des Programms zusammenfällt. Für dynamisch allozierte Variablen muss dies jedoch erfolgen. Explizit werden zwar noch keine dynamischen Variablen unterstützt, doch da Membervariablen ihre Lebensdauer von ihren Elternobjekten erben, muss dieser spezielle Fall berücksichtigt werden. Wie in Zeile 76-78 zu sehen ist, wurde hier ein Advice für die Objektzerstörung verwendet. Es ist zwar nicht unbedingt gegeben, dass auch Verweise auf Teile des jeweiligen Objekt in der Struktur enthalten sind, doch ist das Entfernen des kompletten Speicherbereichs des Objekts aus der Datenstruktur die einzige Möglichkeit. Dies ergibt sich aus der Tatsache, dass noch kein Mechanismus existiert, der die Lebensdauer einer konkreten Variablen überwachen könnte. Sobald dynamische Variablen anderer Art unterstützt werden, für die dieser Weg nicht möglich ist, muss hier ein anderer Ansatz gefunden und berücksichtigt werden.

Interessant wird es noch einmal bei den Get- und Set-Advice für Verweise. Wie zu erkennen ist, beschränkt sich der erforderliche Aufwand nicht nur auf das Auslesen der Informationen (Z. 109,127). Die Prüfung ob überhaupt ein Verweis auf eine überwachte Variable vorliegt erfolgt hier einfach durch das Vorhanden sein der Informationen in der Datenstruktur (Z.108,128). Würde diese für mehrere Advice mit unterschiedlichen Kriterien genutzt, müsste hier auf ein entsprechendes Flag geprüft werden. Zu beachten ist hier, dass die Spezifikation der Variablen in der Pointcut-Funktion nicht ausreicht, da diese bisher nur eine Typ-Kompatibilität prüfen kann.

Einen besonderen Aufwand stellt das Wiederherstellen des Target-Typs dar. Um hier den Typ für den Aufruf der Template Funktion verwenden zu können, ist eine entsprechende Fallunterscheidung notwendig. Insbesondere die Tatsache, dass alle in Frage kommenden Typen explizit aufgeführt und behandelt werden müssen, führt hier zu einem erheblichen Umfang und einer potentiellen Fehlerquelle. Für jede mögliche Klasse muss hier der Abgleich und der folgende Funktionsaufruf samt Cast erfolgen. Mit jeder neuen Klasse muss hier eine Anpassung erfolgen und die Zeilen 114-116 bzw. 132-134 multiplizieren sich.

Der vorliegende Tracing-Aspekt könnte zwar durch weitere Anpassungen auf diese Typ-Wiederherstellung verzichten, da sie nur zur Ausgabe benötigt wird, andere Aspekte können dies jedoch nicht. Die General Object Protection zum Beispiel muss auf das Target-Objekt zugreifen um eine Prüfsumme zu berechnen. Der Aspekt demonstriert also durchaus praktische Probleme und bietet Ansätze zu ihrer Lösung.

4.4.3. Betrachtungen zur Laufzeitdatenstruktur

Grundlegende Anforderung an die Datenstruktur ist das Speichern der Informationen und die Zuordnung zu einem gegebenen Verweis. Um den Overhead so klein wie möglich zu halten, ist hier Effizienz von großer Bedeutung. Weiterhin ist für den allgemeinen Fall eine Unabhängigkeit von extern Bibliotheken wünschenswert.

Sinnvoll wäre beispielsweise den Typ des Verweis mit einzubeziehen, da dieser sowohl beim Eintragen als auch bei der Suche zur Verfügung steht, und somit einen Teil des

Vorgangs zur Übersetzungszeit optimieren zu können. Weiterhin kann die begründete Annahme getroffen werden, dass die verändernden Operation im Vergleich zur Suche seltener sind, so dass hier weitere Optimierungen getroffen werden könnten. Auch ist zu erwarten, dass die Schlüssel sich nicht gleichmäßig auf den Schlüsselraum verteilen: Global wird das Verhältnis von vorhanden zu möglichen Schlüsseln sehr gering sein. Lokal dürfte es jedoch ein Clustering in bestimmten Bereichen aufweisen (je nach Programm: Heap und/oder Stack). Einen Suchbaum wird dies weniger betreffen, Hashbasierte Verfahren sollten dies jedoch bei der Wahl der Hashfunktion berücksichtigen.

Auch das Suchverhalten kann optimiert werden. Häufig wird der Fall auftreten, dass nur ein kleine Menge an Variablen von Interesse ist, aber die Menge der zu prüfenden Verweise hoch, da keine Vorauswahl getroffen werden kann. Hier bieten sich Konzepte an, die es ermöglichen schnell das Vorhandensein der Informationen in der Struktur auszuschließen ohne eine vollständige Suche durchführen zu müssen. Eins dieser Konzepte sind Bloomfilter[Blo70]. So wäre es durch den Vergleich eines Verweises mit der zwischengespeicherten Veroderung aller gespeicherten Verweise möglich mit wenig Aufwand festzustellen, dass der Verweis nicht enthalten sein kann. Unter Einbeziehung weiterer Informationen (alignment der Daten, Unterscheidung Heap und Stack, etc.) wäre es sicherlich möglich dies noch weiter zu verfeinern.

4.5. Bewertung & Ausblick

Die zur Behandlung der Alias-Problematik geschaffenen Erweiterungen von AspectC++ fügen sich ebenfalls nahtlos in die bisherigen ein. Sie ermöglichen zwar keine Auflösung der selbigen, doch schaffen sie Ansatzpunkte um diese zu umgehen. In dem prototypischen Aspekt wird neben der generellen Umsetzbarkeit jedoch auch der notwendige Aufwand deutlich. Insbesondere die Notwendigkeit der manuellen Berücksichtigung aller Klassen stellt ein großes Hindernis dar. Sie ist eine bedeutende Fehlerquelle und erfordert eine konstante Adaption der Advice bei Veränderungen am Zielprogramm. Ferner verhindert sie die Entwicklung von universell verwendbaren Aspekten, da die Advice stark von den im Zielprogramm verwendeten Typen abhängig sind.

Um den Aufwand für den Programmierer zu reduzieren wäre daher eine Integration in den Weber eine Option. Zunächst müssen hierfür jedoch die vorgestellten Konzepte verfeinert und in Verbindung mit anderen Aspekten erprobt werden. Dies würde dem Programmierer zwar den Aufwand abnehmen gleichzeitig aber eine Anpassung an die konkreten Eigenschaften des Aspekts bzw. des Zielprogramms erschweren oder gänzlich unmöglich machen.

Für eine zukünftige Weiterentwicklung ist auch eine im Weber integrierte statische Programmanalyse in Betracht zu ziehen. Diese würde es ermöglichen die jeweiligen Kandidatenmengen sowohl für die Typen als auch für die Variablen selbst erheblich zu reduzieren. Im Zuge dessen wäre zu Evaluieren in wie weit es möglich ist, die Ergebnisse dem Advice-Programmierer zur Verfügung zu stellen, so dass er sie automatisiert nutzen kann. Der Weber selbst wird diese nur nutzen können, wenn er die Konzepte selbst umsetzt und so die Kontrolle über die Datenstruktur besitzt, da er andernfalls keine validen

Annahmen über die Semantik der Datenstruktur tätigen kann.

Zusammenfassend ist also festzuhalten, dass zwar eine nutzbare erste Möglichkeit besteht aber noch großes Potential für weitere Entwicklungen vorhanden ist.

5. Evaluation

Um die neuen Sprachfeatures für AspectC++ richtig einschätzen zu können, sowie den weiteren Entwicklungsprozess zu verbessern, müssen sich die bisher erzielten Ergebnisse einer Bewertung auf mehreren Ebenen unterziehen. Zunächst stellt sich die Frage nach dem Nutzen und der Qualität der Spracherweiterungen. Dies kann sowohl in Bezug zu Anwendungsfällen beurteilt werden, als auch im Vergleich zu anderen ähnlichen Arbeiten. Weiterhin kann die Eignung der getroffenen Entscheidungen zum Entwicklungsprozess und zur Umsetzung betrachtet werden. Schließlich ist es möglich anhand der empirischen Untersuchung von Testsamples die vorhergehenden Betrachtungen zu stützen oder einzuschränken.

Allgemein ist jedoch festzuhalten, dass zum jetzigen Zeitpunkt auf Grund der noch dünnen und nicht repräsentativen Erfahrungslage eine abschließende Beurteilung schwierig ist. Hier ist stark zu erwarten, dass durch die Nutzung in der Zukunft in deutlichem Umfang Erfahrungen gewonnen werden können, die dann in der Rückschau eine wesentlich bessere Beurteilung erlauben.

5.1. Spracherweiterungen: Nutzen und Vergleich

Die Spracherweiterungen sind mit Blick auf eine Klasse von Anwendungsfällen hin entwickelt worden. Anhand eines Beispiels konnte gezeigt werden, dass sie diese auch grundlegend abdecken. In wie weit die getroffenen Einschränkung bezüglich des C++-Sprachumfangs bedeutende Auswirkungen auf die Nutzbarkeit haben, lässt sich aber erst in der Praxis feststellen, da dies stark von den Strukturen des Zielprogramms abhängt. Das Ziel einer ersten Nutzungsmöglichkeit kann aber als erreicht betrachtet werden.

Die getroffenen Definitionen haben sich also soweit als tauglich erwiesen. Bei Betrachtung auf mehr theoretischer Basis, fügen sie sich gut in die Konzepte und Ideen der bisherigen Sprache AspectC++ ein. Die beim Entwurf zu Grunde gelegten Konzepte haben sich als robust erwiesen, aber auch als offen für Erweiterungen, wie sich im Rahmen der Unterstützung von Membervariablen(3.6) und der Alias-Problematik(4.2 + 4.3) gezeigt hat. Zukünftigen Schritten, die für neue Anwendungsfälle weitere Joinpointtypen schaffen müssen oder die Unterstützung in Bezug auf die nutzbaren Variablen und Typen erweitern, steht also von dieser Seite nichts entgegen. Die fehlenden Teile sind daher gesamt gesehen als Mangel zu bewerten, dieser sollte aber in der Zukunft behebbar sein.

Dem Vergleich mit AspectJ[KHH+01] können die Sprachfeatures ebenfalls standhalten. Im Grunde besteht eine große Ähnlichkeit in der Ausgestaltung. Die Möglichkeiten und die Sprache selbst sind dabei fast identisch. Unterschiede ergeben sich zu großen Teilen aus den Unterschieden der Basissprachen. So sind viele der in AspectC++ noch

offenen Probleme auf Grund dieser Unterschiede für AspectJ nicht relevant. Einzig die noch fehlende Unterstützung von Arrays hat AspectJ auf konzeptioneller Ebene hier voraus[CC07].

Die Umsetzung von Magnusson[Mag06] ist hier besser vergleichbar. Im wesentlichen hat sie die gleichen Designentscheidungen getroffen. Sie setzt aber auf eine reine Zuordnung über die Symbole und fordert den Programmierer auf, die Alias-Problematik nicht entstehen zu lassen. Diese Entscheidung entspricht nicht dem Geist der C++ Entwicklung. Der Programmierer sollte sich zwar der Problematik bewusst sein, eine strikte Vermeidung ist aber in der Regel nicht möglich und auch nicht optimal. Daher bieten die begonnenen Maßnahmen hier deutliche Vorteile, da sie es ermöglichen Anwendungsspezifisch mit der Problematik umzugehen.

Eine andere Abweichung stellt die Joinpoint-API dar. Magnusson verwendet hier die separaten Funktionen src und dst. Die aktuelle Lösung, wie sie im Übrigen auch in AspectJ Verwendung findet, dies in die bestehen Funktionen für Argumente und Resultate zu integrieren fügt sich hier deutlich besser in bestehendes ein und bietet ein einheitlicheres Interface, während sie keinen erkennbaren Nachteil aufweist.

Über die Unterstützung der aktuell noch fehlenden Teile durch Magnussons Version liegen leider keine Informationen vor. Aus den Beispielen in [AÖ07] lässt sich eine Unterstützung für Membervariablen und vermutlich auch globale Variablen ableiten. In wie weit aber lokale oder dynamisch erzeugte Variablen unterstützt werden, lässt sich leider nicht erkennen. Für die dynamisch erzeugten ist dies jedoch unwahrscheinlich, da explizit auf die Zugriffe über das jeweilige Symbol abgezielt wird, dass bei dieser Variablenart nicht verfügbar ist. Bei den Datentypen hat sich Magnusson explizit auf die einfachen Typen beschränkt, so das hier keine Unterschiede bestehen.

In der Praxis wird aber der wichtigste Unterschied sein, dass die aktuelle Version gegenüber der von Magnusson in den bestehenden aktuellen AspectC++-Weber integriert ist, so dass sie in der Praxis Anwendung finden kann.

5.2. Entwicklungsprozess und Umsetzung

Die Entscheidung für einen stark inkrementell geprägten Entwicklungsprozess hat sich als richtig erwiesen. Es ist gelungen zumindest erste Teile in hinreichend kleine Schritte aufzuspalten. Dies hat es trotz des an einigen Stellen unerwartet hohen Aufwandes, der zur Umstrukturierungen oder Behandlung von Problemfällen nötig war, ermöglicht erste nutzbare Resultate zu erzielen. Der vorläufige Verzicht auf Features zu Gunsten einer Betrachtung der Alias-Problematik war aus zwei Gründen sinnvoll. Zum einen besteht nun die Möglichkeit der ersten Nutzung und Erprobung der Konzepte, andererseits hat dies die noch zu meisternden Schwierigkeiten verdeutlicht, so dass diese für den weiteren Entwicklungsprozess berücksichtigt werden können. Jedoch haben die Probleme bei der Integration der Version von Simon Schröder[Sch15] gezeigt, dass es noch Verbesserungsbedarf in der Koordination paralleler Entwicklungsschritte gibt. Durch die frühe Integration wurde das Problem vorweggenommen, aber auch bei einer regulären Übernahme der Änderungen in den Hauptentwicklungszweig hätten diese Probleme be-

standen. Hier sollte für die weitere Entwicklung eine bessere Koordination angestrebt werden. Es ist aber zu erwarten, dass dies durch die Entwicklerstruktur des Projekts Beschränkungen unterworfen ist.

Bei der Entwicklung der einzelnen Weberkomponenten konnten in allen Bereichen generalisierte Strukturen geschaffen bzw. existierende ausgebaut werden. Daher wurde auch den Zielen der Modularisierung und Erweiterbarkeit entsprochen. Auf Grund der gemachten Erfahrungen ist eine gute Unterstützung und hohe Wiederverwendbarkeit für die weitere Entwicklung zu erwarten. Die Funktionsfähigkeit und Korrektheit des erzeugten Quellcodes konnte anhand von ausgewählten Beispielen und den entwickelten Regressionstests generell nachgewiesen werden. Da diese begrenzte Menge aber noch nicht als repräsentativ angesehen werden kann, ist eine weitere kontinuierliche Prüfung und Fehlerbehebung zwingend erforderlich.

5.3. Die Alias-Problematik

Hier besteht ganz klar noch Verbesserungsbedarf. Die vorgestellten Konzepte scheinen zwar als erster Ansatz geeignet zu sein, doch kann die Entwicklung hier nicht enden, da sie für einen produktiven Einsatz noch zu aufwändig und fehleranfällig sind. Auch ist ihre Einsetzbarkeit im Bereich der eingebetteten Systeme zumindest in Frage zu stellen. Dies beruht zwar auf einer bewussten Designentscheidung, sich zunächst weitgehend auf Laufzeitkomponenten zu verlassen, doch insbesondere das zu erwartende lineare Wachstum der Codegröße in Bezug auf die Anzahl der existenten Typen in einigen Teilen des erforderlichen Codes, kann schnell zu Problemen beim Speicherbedarf führen. Die Laufzeitstrukturen selbst sind zwar auch noch verbesserungswürdig, es wurden hier jedoch mögliche Potentiale aufgezeigt.

5.4. Empirische Untersuchung

Um eine bessere Einschätzung zu ermöglichen wurden die Änderungen anhand der Qt-Beispielprogramme evaluiert. Erster Schritt war hier die Prüfung auf korrekte Funktionsfähigkeit. Eine rein quantitative Analyse ergab eine syntaktische Korrektheit der durch das Einweben eines Tracing-Aspekts erzeugten Programme. Die korrekte Aufführung der Programme konnte zumindest soweit geprüft werden, dass kein fehlerhaftes Verhalten beobachtet werden konnte.

Um eine Vorstellung von den Auswirkungen der Änderungen zu erhalten wurde anschließend die benötigte Übersetzungszeit der einzelnen Beispiele gemessen. Zum Einsatz kam hier ein System mit einer Intel® Core™ i5-2520M CPU @ 3GHz CPU und 8GB RAM. Der Test fand jeweils auf einem Prozessorkern unter Linux statt. Verwendung fand ein gcc in der Version 4.9.2 und der jeweilige Entwicklungsstand des AspectC++ Webers in Form eines Release-Builds ohne Assertions und mit Optimierung.

Bei den Tests wurden 4 Varianten des Webers unterschieden und eine Vergleichsmessreihe ohne Weber durchgeführt:

	alias	**getset**	**operator**	**trunk**	**gnu**
Joinpoints (sum)	71248	64069	53417	40876	-
Joinpoints (avg)	334,5	300,8	250,8	191,9	-
Laufzeit (sum, s)	2494,54	2447,92	2134,86	2405,11	762,27
Laufzeit (avg, s)	11,711	11,493	10,023	11,292	3,579
Incr. Rel. Trunk	+3,72%	+1,78%	−11,24%	–	–

Tabelle 5.2.: Übersicht Übersetzungszeiten

1. g++ ohne Weber, als Vergleichsreferenz (gnu)
2. ag++/ac++ in der Variante des Hauptzweigs (trunk)
3. die Variante von Simon Schröder[Sch15] (operator)
4. die Version mit Datenjoinpoints aber ohne alias Unterstützung (getset)
5. die aktuelle Version mit alias Unterstützung (alias)

Hier der Übersichtlichkeit halber nur die zusammengefassten Ergebnisse (Tabelle 5.2). Die vollständigen Messreihen können im Anhang A gefunden werden.

Auffällig ist hier die Abweichung der Messreihe 3 (operator). Diese blieb auch bei Wiederholung der Messung bestehen. Eine Erklärung konnte nicht gefunden werden, so dass die Vergleichbarkeit leicht eingeschränkt ist.

Für die neuen Varianten ist nur eine kleine Erhöhung der Laufzeiten gegenüber der normalen Version zu beobachten, obwohl die Anzahl der Joinpoints stark gestiegen ist. Dies entspricht den Ergebnissen in [Sch15, Abschnitt 5.5]. Der starke Anstieg der in [Sch15, Abschnitt 5.4.2] gefunden wurde, deutet jedoch auf eine starke Abhängigkeit des zusätzlichen Aufwandes von den Zielprogrammen an.

Allgemein wird der erhebliche Mehraufwand durch die Verwendung den AspectC++-Webers als vorgeschaltete Source-to-Source Transformation deutlich. Hier liegt grob eine Verdreifachung der Laufzeit vor.

Setzt man für einzelne Testprogramme die Anzahl der Joinpoints und die Übersetzungszeit in Relation (Abbildung 5.1), kann man zumindest einen schwachen Zusammenhang finden. Auch hier lassen sich große Unterschiede zwischen den Programmen finden.

Dies entspricht den Erwartungen. Auch die starke Vergrößerung (Tabelle 5.4) des Projectmodel in XML-Form verwundert nicht. Durch die gestiegene Anzahl der Joinpoints und die anderen neuen Informationen lässt sich dieser leicht erklären. Die Steigerung um ca. 50% ist also nicht bedenklich, zumal die Größe generell weitab eines kritischen Bereichs ist.

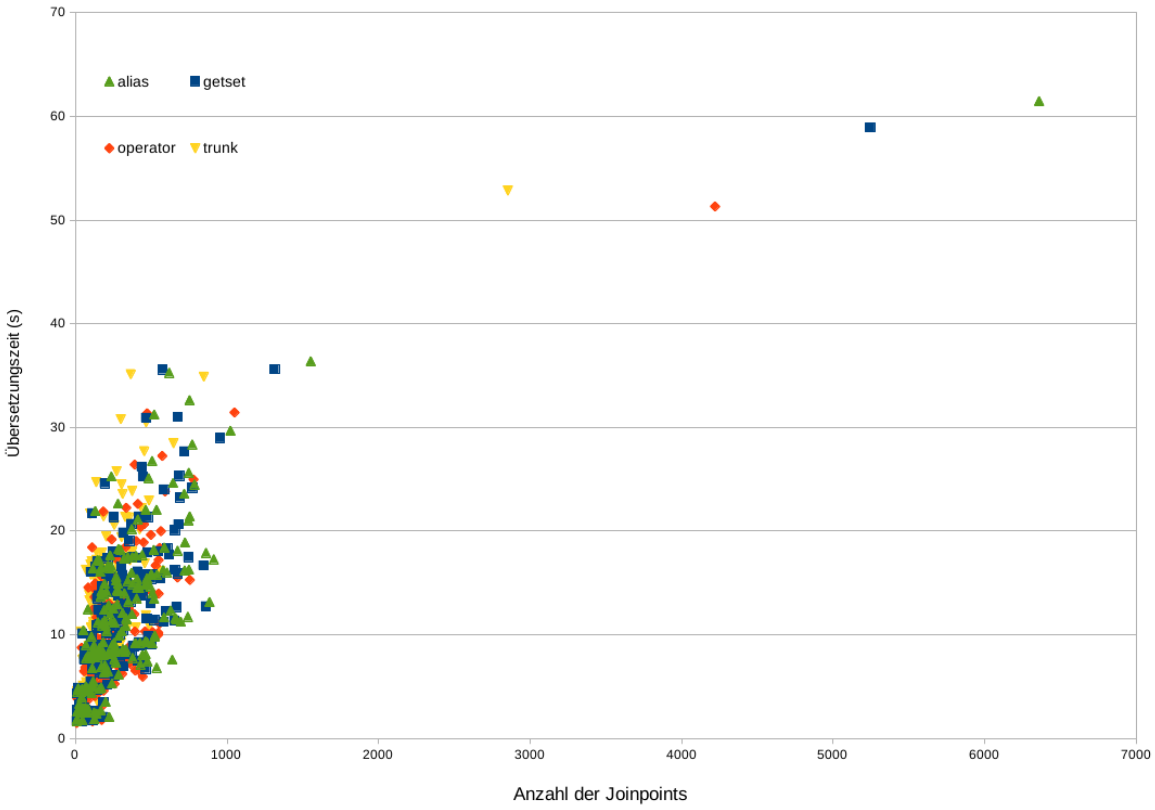

Abbildung 5.1.: Zusammenhang Joinpointzahl / Übersetzungszeit

	alias	*getset*	*operator*	*trunk*
Filesize (sum, byte)	36538680	35267017	29085491	23670961
Filesize (avg, byte)	171543	165573	136552	111131
Incr. Rel. Trunk	$+54,36\%$	$+48,99\%$	$+22,87\%$	–

Tabelle 5.4.: Übersicht Projectmodel

Neben der Übersetzungszeit wurde die Größe der entstandenen Programme untersucht, da diese im Bereich der eingebetteten Systeme kritisch sein kann. Eine Untersuchung der Programmlaufzeit konnte nicht erfolgen, da die Qt-Beispiele auf Grund ihrer Interaktivität nicht für eine Messung geeignet sind.

Die Übersicht in Tabelle 5.6 zeigt, dass sich der Overhead pro Joinpoint im wesentlichen nicht ändert. Die Reduktion hin zu der alias-Variante lässt sich durch einen initialen Offset erklären, der mit steigender Joinpointzahl stärker verteilt wird.

Betrachtet man dieses Verhältnis für die einzelnen Testprogramme bestätigt sich dies (Abbildung 5.2). Die Korrelation ist hier deutlich stärker ausgeprägt als für die Übersetzungszeit.

	alias	getset	operator	trunk	gnu
Joinpoints (sum)	71248	64069	53417	40876	-
Joinpoints (avg)	334, 5	300, 8	250, 8	191, 9	-
Size (sum, bytes)	8328264	8267630	7953498	7649042	5712595
Size (avg, bytes)	39100	38815	37340	35911	26820
Size Incr / JP (avg, bytes)	38, 72	40, 93	42, 72	46, 85	–

Tabelle 5.6.: Übersicht Codegröße

Anhand dieser Messungen lässt sich also vermuten, das kein Unterschied zwischen den einzelnen Joinpointtypen besteht. Die umgesetzten Erweiterungen bewegen sich also im Rahmen des Bekannten und erscheinen daher auch aus dieser Sicht als geeignet.

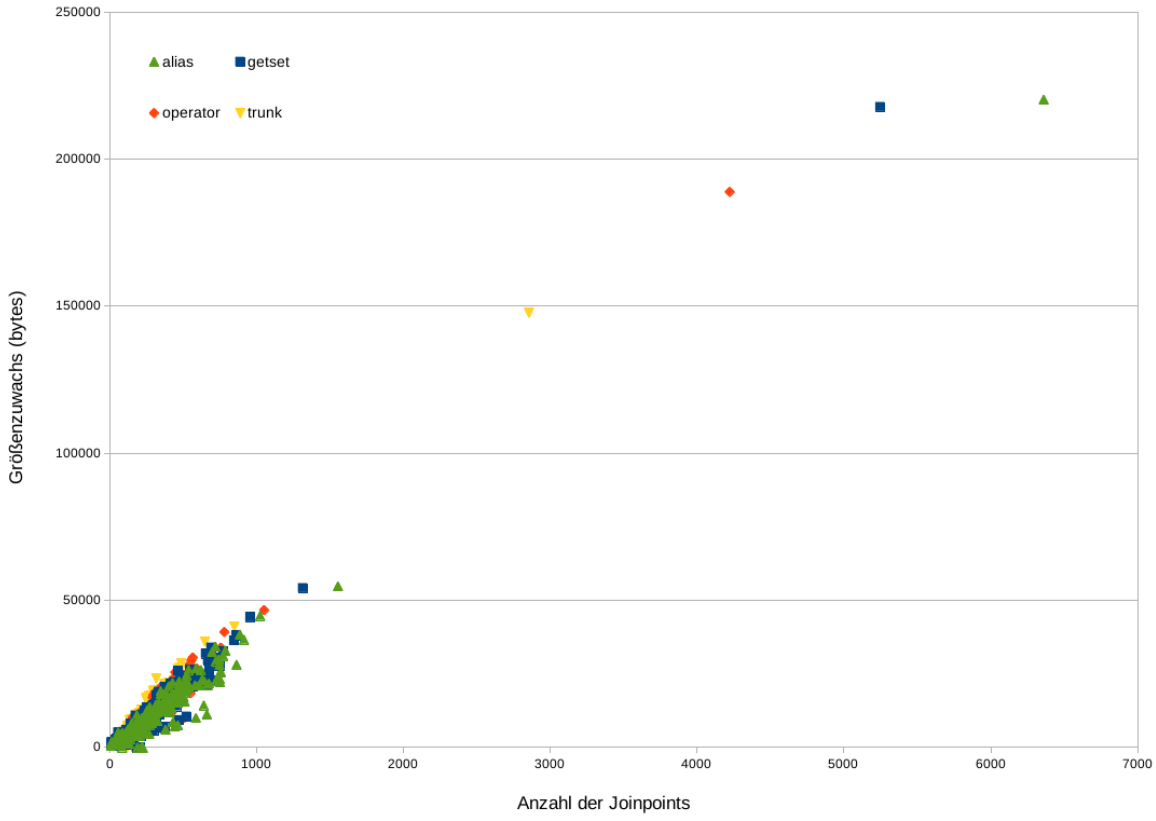

Abbildung 5.2.: Zusammenhang Joinpointzahl / Größenzuwachs

6. Schluss

6.1. Zusammenfassung

Im Rahmen der Arbeit wurden erfolgreich die ersten Schritte im Bereich der Daten basierenden Joinpoints umgesetzt. Die entwickelten neuen Sprachelemente für AspectC++ decken eine wichtige Klasse von Anwendungsfällen ab. Die hierzu getroffenen Definition konnten sich im Rahmen der Umsetzung und den Tests mit Beispielen aus der zur Orientierung genutzten Klasse von Anwendungsfällen bewähren. Den Kern stellen hier die über Datenzugriffe definierten Get- und Setjoinpoints dar.

Zur Behandlung der Alias-Problematik konnten weitere Sprachelemente eingeführt werden. Diese erlauben eine Reaktion auf die sich in diesem Zusammenhang bietenden Herausforderungen. Dieser Ansatz ist noch keine vollständige Lösung, aber ein möglicher Weg konnte gangbar gemacht werden.

Bei der Planung und Ausführung wurde großer Wert auf Erweiterbarkeit gelegt. Bei den notwendigen Anpassungen im AspectC++-Weber konnten viele generalisierte Strukturen geschaffen werden, so dass die aktuellen sowie mögliche zukünftige Erweiterungen schnell und einfach integriert werden können. Auch der Entwicklungsprozess selbst wurde im Sinne der inkrementellen Entwicklung in kleine klar abgrenzte Module zerlegt und somit standen bereits früh Testmöglichkeiten zur Verfügung.

Für einige der noch offenen Fragen konnten bereits Lösungsansätze vorgeschlagen werden. Für andere konnte ein Rahmen für die zukünftigen notwendigen genaueren Betrachtungen abgesteckt werden. Insgesamt wurden somit solide Grundlagen für eine weitere Entwicklung in diesem Bereich geschaffen.

6.2. Ausblick

Denkt man an weitere Entwicklungsschritte fallen einem natürlich sofort die ausgelassenen C++-Sprachfeatures ein. Einige werden verhältnismäßig einfach umzusetzen sein, für andere Bedarf es gründlicher Überlegung. Gemein ist ihnen jedoch, für alle der Nutzen anhand von konkreten Anwendungsfällen noch bestimmt werden muss. Die Unterstützung von lokalen Variablen wird recht einfach zu realisieren sein. Offen ist hier nur die Definition ihrer Signatur. Wie bereits erwähnt, könnte man sich dafür an der Handhabung des Linker orientieren. Alles weitere sollte sich unter nur minimalen Anpassungen aus dem Bestehenden ergeben. Anderes verhält es sich für die Erweiterung um Heapallozierte Variablen, da hier das Symbol gänzlich fehlt, muss eine Alternative gefunden werden. Ein möglicher Ansatzpunkt wäre der Ort ihrer Allozierung im Quellcode, da die-

ser halbwegs einfach beschrieben werden kann. Die Details und die Tauglichkeit müssten dabei noch untersucht werden.

Aber auch weitergehende Features versprechen einen Nutzen. Wie in der Analyse erwähnt, wäre es möglich die Pointcut-Funktionen für Argumente oder das Zielobjekt zu erweitern, so dass sie auch Informationen über das Objekt bzw. die Variablen selbst und nicht nur den Typ berücksichtigen. Dies würde es ermöglichen bestehende Aspekte besser auf ein gegebenes Zielprogramm hin maßzuschneidern. Hier kann davon profitiert werden, dass es die Aspekte in die Lage versetzt ihren Wirkungsbereich optimaler zu wählen. In der Umsetzung wären die notwendigen Komponenten quasi schon vorhanden, da die AST-Verarbeitung bereits allgemein die Verwendung von Variablen bestimmen kann. Auch in den anderen Komponenten kann auf bereits genutzten Code zurückgegriffen werden, es bliebe also nur das interne Modell um die neuen Informationen zu erweitern, die Pointcut-Auswertung analog zu den bestehenden Funktionen anzupassen sowie die Einzelkomponenten zusammenzufügen.

Einen großen Bereich stell weiterhin die Alias-Problematik dar. Hier bietet sich sowohl ein hoher Bedarf an als auch großes Potential für weitere Entwicklungsschritte. Die bisherigen Ergebnisse sind zwar als erster Schritt nutzbar, bedeuten jedoch noch einen viel zu hohen Aufwand. Es müssen Verbesserungen für die vorgeschlagenen oder neue Konzepte gefunden werden um diesen zu reduzieren. Insbesondere das Problem der Codevervielfachung wird hier besonderer Aufmerksamkeit bedürfen. Auch die Laufzeitstrukturen können noch verfeinert werden. Einige Möglichkeiten dazu wurden bereits vorgeschlagen, andere werden sich bei der Analyse von Anwendungsfällen ergeben.

Das größte Potential verspricht aber eine statische Analyse, der in Frage kommenden Zeiger und Typen. Gelingt es hier die jeweiligen Kandidatenmengen zur Übersetzungszeit erheblich zu reduzieren, entschärft dies die anderen Bereiche erheblich. Hier kann auf ein breites Feld an Erfahrung aus der Forschung zur Compileroptimierung zurückgegriffen werden. Es wird also darum gehen, geeignete Algorithmen zu identifizieren und sie auf das neue Gebiet zu übertragen. Gelingt dies, ist ein hoher Gewinn an Effizienz bei Verwendung der Verweis basierten Joinpoints zu erwarten.

Allgemein sind die erfolgten ersten Schritte aber nur der Start in einen neuen Bereich der Möglichkeiten für AspectC++ und viele zukünftige Anwendungsfälle sind noch nicht abzusehen.

A. Messreihen Qt-Beispiele

Hinweis: Die Tabellen auf Grund des Formats erst auf der folgenden Seite.

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
animation/animatedtiles	182	83	2	0	16	2	285
animation/easing	315	105	132	21	26	5	604
animation/moveblocks	120	39	5	2	4	0	170
dbus/chat	378	106	128	24	38	0	674
dbus/complexpingpong	89	114	8	1	12	0	224
dbus/listnames	51	5	0	0	0	0	56
dbus/pingpong	44	34	0	0	4	0	82
dbus/remotecontrolledcar/car	139	34	15	4	14	0	206
dbus/remotecontrolledcar/controller	146	35	31	7	10	0	229
declarative/cppextensions/imageprovider	57	11	0	0	4	0	72
declarative/cppextensions/networkaccessmanagerfactory	47	13	2	1	7	0	70
declarative/cppextensions/plugins	128	56	8	5	24	2	223
declarative/cppextensions/qgraphicslayouts/layoutitem	20	2	0	0	0	0	22
declarative/cppextensions/qgraphicslayouts/qgraphicsgridlayout	255	172	38	36	96	42	639
declarative/cppextensions/qgraphicslayouts/qgraphicslinearlayout	195	126	21	7	60	20	429
declarative/cppextensions/qwidgets	79	37	7	1	14	1	139
declarative/cppextensions/referenceexamples/adding	62	29	2	2	10	2	107
declarative/cppextensions/referenceexamples/attached	241	123	7	7	56	8	442
declarative/cppextensions/referenceexamples/binding	356	180	12	8	96	8	660
declarative/cppextensions/referenceexamples/coercion	153	66	5	4	34	5	267
declarative/cppextensions/referenceexamples/default	153	66	5	4	34	5	267
declarative/cppextensions/referenceexamples/extended	79	71	16	4	22	8	200

Tabelle A.2.: Joinpoints 1/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
declarative/cppextensions/referenceexamples/grouped	204	102	7	7	50	8	378
declarative/cppextensions/referenceexamples/properties	101	58	5	4	22	5	195
declarative/cppextensions/referenceexamples/signal	253	130	7	7	58	8	463
declarative/cppextensions/referenceexamples/valuesource	317	165	9	8	78	8	585
declarative/imageprovider	57	11	0	0	4	0	72
declarative/objectlistmodel	75	30	0	0	12	0	117
declarative/plugins	128	56	8	5	24	2	223
designer/calculatorbuilder	64	18	6	3	8	2	101
designer/calculatorform	159	28	75	14	8	2	286
designer/containerextension	230	84	39	7	38	9	407
designer/taskmenuextension	319	108	19	15	22	0	483
designer/worldtimeclockbuilder	17	1	0	0	0	0	18
desktop/screenshot	130	24	44	11	8	0	217
desktop/systray	189	39	76	21	8	2	335
dialogs/classwizard	506	58	82	25	24	0	695
dialogs/configdialog	172	16	19	2	8	2	219
dialogs/extension	64	15	23	11	4	0	117
dialogs/findfiles	180	34	35	10	10	2	271
dialogs/licensewizard	317	53	49	19	24	0	462
dialogs/standarddialogs	282	52	67	17	8	0	426
dialogs/tabdialog	188	25	7	2	16	0	238
dialogs/trivialwizard	38	1	0	0	0	0	39

Tabelle A.4.: Joinpoints 2/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
draganddrop/draggableicons	102	9	0	0	0	0	111
draganddrop/draggabletext	118	22	2	0	4	2	148
draganddrop/dropsite	185	40	26	6	19	2	278
draganddrop/fridgemagnets	163	24	1	0	4	1	193
draganddrop/puzzle	338	73	14	6	32	1	464
effects/lighting	91	37	7	2	12	2	151
gestures/imagegestures	209	68	29	16	50	0	372
graphicsview/anchorlayout	47	23	0	0	0	0	70
graphicsview/collidingmice	112	87	43	10	4	1	257
graphicsview/diagramscene	862	203	254	47	150	37	1553
graphicsview/dragdroprobot	233	26	3	5	8	0	275
graphicsview/elasticnodes	337	121	57	5	50	9	579
graphicsview/padnavigator	475	107	96	18	46	11	753
graphicsview/simpleanchorlayout	47	1	1	0	8	1	58
graphicsview/weatheranchorlayout	150	30	6	1	12	0	199
help/contextsensitivehelp	279	51	142	33	12	2	519
help/remotecontrol	268	55	157	31	10	3	524
help/simpletextviewer	289	60	86	20	16	1	472
ipc/localfortuneclient	89	21	31	8	10	1	160
ipc/localfortuneserver	89	15	9	3	6	0	122
ipc/sharedmemory	118	18	25	4	6	1	172
itemviews/addressbook	382	108	82	21	26	0	619

Tabelle A.6.: Joinpoints 3/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
itemviews/basicsortfiltermodel	135	22	55	13	4	0	229
itemviews/chart	566	195	85	17	38	12	913
itemviews/coloreditorfactory	105	32	0	0	8	0	145
itemviews/combowidgetmapper	95	28	37	11	6	1	178
itemviews/customsortfiltermodel	210	34	52	13	12	0	321
itemviews/dirview	11	1	0	0	0	0	12
itemviews/editabletreemodel	429	133	117	19	36	12	746
itemviews/fetchmore	112	38	13	3	18	3	187
itemviews/pixelator	265	76	35	5	28	3	412
itemviews/puzzle	365	95	24	7	40	4	535
itemviews/simpledommodel	146	42	15	8	24	3	238
itemviews/simpletreemodel	133	34	17	2	22	7	215
itemviews/simplewidgetmapper	88	27	35	10	6	1	167
itemviews/spinboxdelegate	52	13	0	0	4	0	69
itemviews/stardelegate	192	54	9	4	22	0	281
layouts/basiclayouts	80	30	18	9	4	0	141
layouts/borderlayout	154	52	15	4	18	7	250
layouts/dynamiclayouts	122	37	49	11	16	6	241
layouts/flowlayout	119	51	7	0	10	3	190
linguist/arrowpad	75	16	9	7	8	0	115
linguist/hellotr	7	1	0	0	0	0	8
linguist/trollprint	111	23	19	10	8	0	171

Tabelle A.8.: Joinpoints 4/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
mainwindows/application	249	40	78	18	12	2	399
mainwindows/dockwidgets	273	42	58	16	4	0	393
mainwindows/mdi	447	72	138	29	30	5	721
mainwindows/menus	296	34	147	26	4	0	507
mainwindows/recentfiles	198	47	42	11	14	2	314
mainwindows/sdi	287	52	87	21	22	3	472
network/blockingfortuneclient	175	58	36	10	30	2	311
network/broadcastreceiver	56	11	9	3	4	0	83
network/broadcastsender	58	16	16	8	4	0	102
network/download	98	17	1	0	8	1	125
network/downloadmanager	149	43	20	9	16	4	241
network/fortuneclient	150	31	51	11	10	1	254
network/fortuneserver	137	20	16	4	6	0	183
network/googlesuggest	181	37	57	3	18	2	298
network/http	249	59	111	25	22	5	471
network/loopback	130	40	51	14	22	3	260
network/qftp	304	49	121	15	26	7	522
network/securesocketclient	471	89	267	42	14	2	885
network/threadedfortuneserver	157	28	7	2	20	1	215
network/torrent	2935	1368	974	202	731	151	6361
opengl/2dpainting	146	47	6	4	34	0	237
opengl/framebufferobject	248	141	59	21	26	6	501

Tabelle A.10.: Joinpoints 5/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
opengl/framebufferobject2	134	56	11	5	0	0	206
opengl/grabber	357	215	88	32	44	13	749
opengl/hellogl	393	154	76	17	66	17	723
opengl/overpainting	463	216	78	13	74	18	862
opengl/pbuffers	348	136	90	24	56	12	666
opengl/pbuffers2	249	140	57	20	28	6	500
opengl/samplebuffers	81	21	3	1	0	0	106
opengl/textures	159	87	9	8	20	1	284
painting/basicdrawing	317	58	114	22	24	3	538
painting/concentriccircles	98	62	8	10	8	0	186
painting/fontsampler	401	149	115	20	42	15	742
painting/imagecomposition	164	27	19	5	28	0	243
painting/painterpaths	221	96	56	16	28	3	420
painting/svgviewer	240	54	68	10	28	4	404
painting/transformations	197	49	10	2	22	2	282
qtconcurrent/imagescaling	94	22	37	6	10	2	171
qtconcurrent/map	10	2	0	0	0	0	12
qtconcurrent/progressdialog	27	13	2	0	2	1	45
qtconcurrent/runfunction	11	0	0	0	0	0	11
qtestlib/tutorial1	34	10	0	0	4	0	48
qtestlib/tutorial2	51	12	0	0	4	0	67
qtestlib/tutorial3	35	11	0	0	4	0	50

Tabelle A.12.: Joinpoints 6/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
qtestlib/tutorial4	52	13	0	0	4	0	69
qtestlib/tutorial5	84	24	3	0	10	3	124
richtext/calendar	144	29	6	3	16	2	200
richtext/orderform	283	43	27	9	12	1	375
richtext/syntaxhighlighter	185	40	6	2	36	0	269
richtext/textobject	113	23	10	4	8	0	158
script/calculator	25	1	0	0	0	0	26
script/customclass	295	103	17	10	40	11	476
script/defaultprototypes	125	50	0	0	8	0	183
script/marshal	31	8	1	0	2	1	43
script/qscript	318	129	20	12	44	13	536
sql/cachedtable	103	15	24	5	4	0	151
sql/drilldown	309	92	65	15	34	5	520
sql/masterdetail	491	79	110	19	16	3	718
sql/querymodel	153	31	0	0	8	0	192
sql/relationaltablemodel	71	4	0	0	0	0	75
sql/sqlwidgetmapper	96	25	46	11	6	1	185
sql/tablemodel	58	7	0	0	0	0	65
statemachine/eventtransitions	24	0	0	0	0	0	24
statemachine/factorial	80	52	10	4	16	5	167
statemachine/pingpong	42	2	0	0	0	0	44
statemachine/rogue	137	88	26	8	10	0	269

Tabelle A.14.: Joinpoints 7/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
statemachine/trafficlight	97	35	10	5	10	2	159
statemachine/twowaybutton	14	4	0	0	0	0	18
threads/mandelbrot	195	201	39	18	52	4	509
threads/semaphores	26	11	5	0	0	0	42
threads/waitconditions	36	15	9	2	4	0	66
tools/codecs	220	57	51	13	30	5	376
tools/completer	212	36	62	8	10	3	331
tools/contiguouscache	57	24	11	0	10	2	104
tools/customcompleter	196	36	33	3	8	0	276
tools/echoplugin	108	24	19	5	8	0	164
tools/i18n	175	49	24	12	18	3	281
tools/inputpanel	381	79	209	28	46	5	748
tools/plugandpaintplugins/basictools	156	52	2	0	4	0	214
tools/plugandpaintplugins/extrafilters	120	64	0	0	4	0	188
tools/regexp	145	44	56	14	4	0	263
tools/settingseditor	659	132	152	31	42	7	1023
tools/styleplugin	106	13	0	0	12	0	131
tools/treemodelcompleter	223	71	42	8	24	5	373
tools/undoframework	278	64	84	22	54	4	506
touch/dials	64	13	34	9	2	1	123
touch/fingerpaint	267	66	38	16	26	3	416
touch/knobs	39	2	0	0	0	0	41

Tabelle A.16.: Joinpoints 8/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
touch/pinchzoom	182	97	46	11	12	1	349
tutorials/addressbook/part1	38	6	2	2	4	0	52
tutorials/addressbook/part2	106	14	39	5	12	0	176
tutorials/addressbook/part3	158	20	61	7	12	0	258
tutorials/addressbook/part4	205	27	68	10	26	0	336
tutorials/addressbook/part5	271	41	83	14	32	0	441
tutorials/addressbook/part6	321	48	96	16	36	0	517
tutorials/addressbook/part7	376	54	104	17	36	0	587
uitools/multipleinheritance	168	29	83	15	8	2	305
uitools/textfinder	84	21	6	5	4	0	120
widgets/analogclock	68	20	1	0	6	1	96
widgets/calculator	276	82	51	32	30	4	475
widgets/calendarwidget	332	64	179	32	16	6	629
widgets/charactermap	250	109	85	15	28	7	494
widgets/codeeditor	121	30	11	2	10	3	177
widgets/digitalclock	50	10	0	0	10	0	70
widgets/groupbox	119	4	0	0	4	0	127
widgets/icons	466	137	108	24	32	4	771
widgets/imageviewer	188	41	79	17	8	0	333
widgets/lineedits	136	23	32	5	14	5	215
widgets/movie	164	35	101	15	10	1	326
widgets/scribble	275	73	54	23	36	3	464

Tabelle A.18.: Joinpoints 9/10

Name	Joinpoints						
	Functions	**Operators**	**Get**	**Set**	**Ref**	**alias**	**Total**
widgets/shapedclock	105	27	2	0	12	2	148
widgets/sliders	149	44	86	16	22	6	323
widgets/spinboxes	186	20	37	8	6	1	258
widgets/stylesheet	442	62	232	42	8	0	786
widgets/tablet	320	119	119	39	44	3	644
widgets/tetrix	318	245	86	35	56	14	754
widgets/tooltips	241	60	12	6	46	5	370
widgets/wiggly	109	36	2	2	17	0	166
widgets/windowflags	264	49	59	27	8	0	407
xml/dombookmarks	253	43	23	8	16	2	345
xml/htmlinfo	69	11	0	0	4	0	84
xml/rsslisting	137	29	28	5	33	3	235
xml/saxbookmarks	268	44	49	15	24	0	400
xml/streambookmarks	267	31	29	8	11	0	346
xml/xmlstreamlint	39	7	0	0	0	0	46

Tabelle A.20.: Joinpoints 10/10

Name	Kompilierzeit (s)				
	alias	*getset*	*operator*	*trunk*	*gnu*
animation/animatedtiles	6,14	6,07	5,26	5,87	1,91
animation/easing	16,00	15,40	13,51	14,70	4,44
animation/moveblocks	4,90	4,88	4,44	4,69	1,56
dbus/chat	18,10	17,72	15,33	17,17	5,12
dbus/complexpingpong	16,44	16,08	14,34	16,16	16,67
dbus/listnames	2,91	2,98	2,57	2,87	0,99
dbus/pingpong	12,44	12,42	11,13	12,55	12,54
dbus/remotecontrolledcar/car	14,12	13,88	12,64	13,95	4,16
dbus/remotecontrolledcar/controller	17,65	17,37	15,58	17,22	5,28
declarative/cppextensions/imageprovider	2,61	2,52	2,33	2,50	0,83
declarative/cppextensions/networkaccessmanagerfactory	2,69	2,72	2,43	2,64	0,90
declarative/cppextensions/plugins	2,06	2,05	1,82	2,06	0,73
declarative/cppextensions/qgraphicslayouts/layoutitem	2,65	2,60	2,34	2,56	0,81
declarative/cppextensions/qgraphicslayouts/qgraphicsgridlayout	7,57	7,41	6,51	7,19	2,26
declarative/cppextensions/qgraphicslayouts/qgraphicslinearlayout	7,13	6,93	6,22	6,93	2,23
declarative/cppextensions/qwidgets	2,38	2,33	2,20	2,34	0,85
declarative/cppextensions/referenceexamples/adding	4,33	4,32	3,77	4,29	1,29
declarative/cppextensions/referenceexamples/attached	8,04	8,09	7,42	8,05	2,55
declarative/cppextensions/referenceexamples/binding	11,61	11,42	10,25	11,30	3,65
declarative/cppextensions/referenceexamples/coercion	7,28	7,38	6,38	7,20	2,26
declarative/cppextensions/referenceexamples/default	7,25	7,17	6,42	7,18	2,24
declarative/cppextensions/referenceexamples/extended	6,45	6,38	5,83	6,22	1,95

Tabelle A.22.: Übersetzungszeit 1/10

Name	Kompilierzeit (s)				
	alias	*getset*	*operator*	*trunk*	*gnu*
declarative/cppextensions/referenceexamples/grouped	7,76	7,85	6,73	7,62	2,41
declarative/cppextensions/referenceexamples/properties	7,05	6,99	6,14	7,01	2,18
declarative/cppextensions/referenceexamples/signal	8,11	8,22	7,17	7,98	2,57
declarative/cppextensions/referenceexamples/valuesource	11,65	11,55	10,30	11,36	3,69
declarative/imageprovider	2,58	2,55	2,30	2,58	0,84
declarative/objectlistmodel	5,50	5,48	4,83	5,41	1,67
declarative/plugins	2,10	2,01	1,77	2,07	0,70
designer/calculatorbuilder	8,03	8,00	7,08	7,92	2,41
designer/calculatorform	9,89	9,70	8,32	9,22	2,72
designer/containerextension	15,37	15,29	13,27	15,06	4,43
designer/taskmenuextension	25,10	25,24	22,61	24,48	7,48
designer/worldtimeclockbuilder	4,83	4,86	4,40	4,82	1,50
desktop/screenshot	8,15	7,85	7,24	7,73	2,43
desktop/systray	11,59	11,51	10,09	11,17	3,46
dialogs/classwizard	11,26	11,37	10,00	10,83	3,44
dialogs/configdialog	12,82	12,64	11,10	12,68	3,87
dialogs/extension	7,78	7,55	6,83	7,64	2,29
dialogs/findfiles	8,51	8,27	7,45	8,26	2,55
dialogs/licensewizard	9,35	9,26	8,79	9,21	2,78
dialogs/standarddialogs	9,21	9,09	7,93	8,65	2,68
dialogs/tabdialog	8,18	7,91	7,05	7,95	2,48
dialogs/trivialwizard	4,50	4,41	3,89	4,55	1,43

Tabelle A.24.: Übersetzungszeit 2/10

Name	Kompilierzeit (s)				
	alias	*getset*	*operator*	*trunk*	*gnu*
draganddrop/draggableicons	6,76	6,63	5,96	6,55	2,05
draganddrop/draggabletext	11,20	10,90	9,67	11,22	3,42
draganddrop/dropsite	14,24	14,10	12,33	13,89	4,37
draganddrop/fridgemagnets	11,39	11,18	9,83	11,16	3,41
draganddrop/puzzle	22,04	21,39	19,03	21,35	6,62
effects/lighting	8,25	8,14	7,19	8,12	2,52
gestures/imagegestures	20,19	19,84	17,29	19,46	6,12
graphicsview/anchorlayout	4,66	4,53	4,01	4,53	1,43
graphicsview/collidingmice	7,63	7,52	6,73	7,26	2,33
graphicsview/diagramscene	36,38	35,60	31,43	34,89	10,60
graphicsview/dragdroprobot	14,31	14,15	12,31	14,06	4,67
graphicsview/elasticnodes	16,22	15,79	14,01	15,45	4,81
graphicsview/padnavigator	32,63	30,99	27,24	30,48	9,33
graphicsview/simpleanchorlayout	4,54	4,47	4,11	4,67	1,42
graphicsview/weatheranchorlayout	3,51	3,48	3,13	3,47	1,08
help/contextsensitivehelp	13,45	13,05	11,22	12,11	3,56
help/remotecontrol	9,92	9,39	7,64	8,30	2,43
help/simpletextviewer	15,44	15,26	13,59	15,09	4,50
ipc/localfortuneclient	8,75	8,45	7,67	8,58	2,65
ipc/localfortuneserver	8,94	8,82	7,84	8,90	2,74
ipc/sharedmemory	6,89	6,99	6,12	6,92	2,03
itemviews/addressbook	35,28	35,59	31,35	35,13	10,85

Tabelle A.26.: Übersetzungszeit 3/10

Name	**Kompilierzeit (s)**				
	alias	*getset*	*operator*	*trunk*	*gnu*
itemviews/basicsortfiltermodel	11,93	10,94	9,53	10,68	3,31
itemviews/chart	17,27	16,67	15,28	16,14	4,84
itemviews/coloreditorfactory	17,19	16,56	14,94	16,75	5,20
itemviews/combowidgetmapper	7,99	7,88	6,79	7,69	2,42
itemviews/customsortfiltermodel	17,33	16,58	14,40	16,54	4,99
itemviews/dirview	4,50	4,33	3,82	4,54	1,43
itemviews/editabletreemodel	21,01	20,63	18,37	20,15	6,11
itemviews/fetchmore	11,06	10,63	9,43	10,63	3,23
itemviews/pixelator	21,12	20,68	18,46	20,59	6,19
itemviews/puzzle	22,05	21,33	18,90	21,32	6,67
itemviews/simpledommodel	16,55	15,83	14,02	15,93	4,81
itemviews/simpletreemodel	12,65	12,08	10,81	12,16	3,82
itemviews/simplewidgetmapper	8,19	7,70	6,79	7,86	2,44
itemviews/spinboxdelegate	8,99	8,36	7,47	8,43	2,57
itemviews/stardelegate	22,63	21,34	19,19	21,42	6,72
layouts/basiclayouts	8,01	7,71	6,86	7,71	2,35
layouts/borderlayout	12,51	12,19	10,89	12,09	3,73
layouts/dynamiclayouts	8,12	7,94	7,04	7,88	2,43
layouts/flowlayout	12,38	12,11	10,67	11,91	3,74
linguist/arrowpad	16,36	16,03	14,54	16,26	4,90
linguist/hellotr	1,66	1,64	1,48	1,68	0,53
linguist/trollprint	16,49	16,27	14,35	16,15	4,99

Tabelle A.28.: Übersetzungszeit 4/10

Name	Kompilierzeit (s)				
	alias	*getset*	*operator*	*trunk*	*gnu*
mainwindows/application	9,17	8,91	7,88	8,65	2,75
mainwindows/dockwidgets	9,13	8,89	7,86	8,91	2,66
mainwindows/mdi	16,20	15,91	13,98	15,62	5,02
mainwindows/menus	9,15	9,05	7,72	8,49	2,54
mainwindows/recentfiles	8,61	8,36	7,37	8,20	2,58
mainwindows/sdi	9,28	9,06	7,90	9,14	2,72
network/blockingfortuneclient	12,08	11,70	10,28	11,66	3,53
network/broadcastreceiver	7,99	7,60	6,82	7,66	2,39
network/broadcastsender	7,92	7,72	6,81	7,68	2,36
network/download	1,85	1,77	1,62	1,78	0,62
network/downloadmanager	5,26	5,16	4,59	5,10	1,59
network/fortuneclient	9,20	8,91	7,77	8,84	2,72
network/fortuneserver	9,26	9,03	8,10	9,12	2,88
network/googlesuggest	18,19	17,95	15,72	17,92	5,45
network/http	9,26	8,99	7,83	8,70	2,74
network/loopback	8,89	8,71	7,65	8,68	2,67
network/qftp	9,85	9,82	8,39	9,25	2,93
network/securesocketclient	13,15	12,72	11,06	11,87	3,49
network/threadedfortuneserver	16,66	16,10	14,37	16,19	4,87
network/torrent	61,48	58,93	51,31	52,84	17,15
opengl/2dpainting	25,25	24,60	21,87	24,71	7,44
opengl/framebufferobject	15,82	15,43	13,61	15,20	4,57

Tabelle A.30.: Übersetzungszeit 5/10

Name	*Kompilierzeit(s)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
opengl/framebufferobject2	10,34	10,11	9,13	10,13	3,00
opengl/grabber	16,27	16,24	13,95	15,37	4,68
opengl/hellogl	18,89	18,34	16,66	18,05	5,47
opengl/overpainting	17,89	17,48	15,51	16,88	5,15
opengl/pbuffers	11,48	11,23	9,76	10,75	3,31
opengl/pbuffers2	15,78	15,80	13,47	15,10	4,74
opengl/samplebuffers	9,82	9,82	8,45	9,73	2,94
opengl/textures	18,27	18,03	15,77	17,84	5,49
painting/basicdrawing	15,78	15,39	13,21	15,68	4,55
painting/concentriccircles	13,90	13,86	12,05	13,32	4,12
painting/fontsampler	11,73	12,66	10,18	10,69	3,39
painting/imagecomposition	8,99	9,03	8,06	8,75	2,62
painting/painterpaths	14,52	14,15	12,56	13,94	4,28
painting/svgviewer	13,49	13,17	11,57	13,02	3,82
painting/transformations	14,27	14,18	12,56	13,91	4,31
qtconcurrent/imagescaling	14,15	13,79	12,39	13,81	4,70
qtconcurrent/map	2,61	2,71	2,41	2,59	1,28
qtconcurrent/progressdialog	4,76	4,66	4,37	4,67	1,59
qtconcurrent/runfunction	1,85	1,83	1,71	1,81	0,64
qtestlib/tutorial1	2,91	2,81	2,87	2,83	0,95
qtestlib/tutorial2	2,94	2,89	2,63	2,87	0,96
qtestlib/tutorial3	4,69	4,63	4,24	4,67	1,61

Tabelle A.32.: Übersetzungszeit 6/10

Name	*Kompilierzeit(s)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
qtestlib/tutorial4	5,00	4,85	4,78	5,08	1,74
qtestlib/tutorial5	4,98	4,82	4,34	4,76	1,60
richtext/calendar	10,85	11,17	9,34	10,71	3,29
richtext/orderform	17,48	19,06	14,84	17,26	5,29
richtext/syntaxhighlighter	15,23	15,65	12,99	14,95	4,65
richtext/textobject	17,16	17,05	14,46	17,06	5,17
script/calculator	3,32	3,23	2,78	3,23	1,04
script/customclass	7,41	7,50	6,53	7,45	2,12
script/defaultprototypes	8,31	8,33	7,11	8,21	2,56
script/marshal	2,75	2,71	2,36	2,69	0,96
script/qscript	6,82	6,71	5,94	6,63	1,93
sql/cachedtable	8,16	7,89	6,77	7,75	2,38
sql/drilldown	31,23	30,91	26,40	30,79	9,39
sql/masterdetail	23,58	23,24	19,98	22,94	7,11
sql/querymodel	14,90	14,68	12,47	14,59	4,39
sql/relationaltablemodel	4,85	4,71	4,14	4,82	1,58
sql/sqlwidgetmapper	8,07	7,93	6,75	7,88	2,50
sql/tablemodel	4,68	4,64	4,00	4,64	1,49
statemachine/eventtransitions	4,67	4,41	3,84	4,44	1,41
statemachine/factorial	2,67	2,62	2,33	2,50	0,81
statemachine/pingpong	2,48	2,46	2,09	2,38	0,73
statemachine/rogue	15,59	15,04	13,04	15,42	4,69

Tabelle A.34.: Übersetzungszeit 7/10

Name	*Kompilierzeit(s)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
statemachine/trafficlight	4,75	4,70	4,07	4,66	1,52
statemachine/twowaybutton	4,67	4,41	3,80	4,41	1,41
threads/mandelbrot	14,00	13,76	11,98	13,46	4,03
threads/semaphores	2,43	2,38	2,04	2,39	0,70
threads/waitconditions	2,47	2,42	2,07	2,46	0,71
tools/codecs	14,86	14,31	12,33	14,21	4,46
tools/completer	10,82	10,41	8,98	10,27	3,08
tools/contiguouscache	5,00	4,89	4,24	4,80	1,47
tools/customcompleter	12,93	12,51	10,78	12,47	3,79
tools/echoplugin	16,06	16,04	13,62	15,78	4,98
tools/i18n	14,91	14,66	12,60	14,77	4,54
tools/inputpanel	25,62	25,32	20,65	23,85	7,05
tools/plugandpaintplugins/basictools	6,50	6,27	5,51	6,29	1,89
tools/plugandpaintplugins/extrafilters	6,34	6,21	5,42	6,14	1,90
tools/regexp	8,30	8,24	6,94	7,94	2,45
tools/settingseditor	29,68	28,97	24,97	28,46	8,70
tools/styleplugin	21,93	21,71	18,42	21,65	6,67
tools/treemodelcompleter	11,95	12,03	10,14	11,89	3,49
tools/undoframework	26,76	26,19	22,24	25,75	8,49
touch/dials	2,49	2,54	2,08	2,33	0,76
touch/fingerpaint	14,71	14,62	12,46	14,45	4,46
touch/knobs	3,95	3,83	3,29	3,86	1,16

Tabelle A.36.: Übersetzungszeit 8/10

Name	*Kompilierzeit(s)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
touch/pinchzoom	13,18	12,95	11,26	12,74	3,85
tutorials/addressbook/part1	10,42	10,12	8,74	10,30	3,19
tutorials/addressbook/part2	10,89	10,70	9,29	10,71	3,29
tutorials/addressbook/part3	11,15	10,86	9,64	10,75	3,37
tutorials/addressbook/part4	11,45	11,32	9,85	10,99	3,45
tutorials/addressbook/part5	17,69	17,41	14,67	17,28	5,26
tutorials/addressbook/part6	18,18	17,93	14,95	17,48	5,45
tutorials/addressbook/part7	18,40	18,04	15,30	17,93	5,45
uitools/multipleinheritance	10,33	10,02	8,18	9,38	2,71
uitools/textfinder	8,27	8,42	6,95	8,15	2,44
widgets/analogclock	7,74	7,77	6,49	7,58	2,30
widgets/calculator	14,82	14,70	12,59	14,26	4,36
widgets/calendarwidget	12,33	12,28	10,30	11,26	3,54
widgets/charactermap	14,63	14,33	12,38	14,01	4,25
widgets/codeeditor	11,11	11,01	9,53	10,96	3,50
widgets/digitalclock	7,73	7,57	6,46	7,56	2,30
widgets/groupbox	7,95	7,73	6,75	7,92	2,37
widgets/icons	28,35	27,67	23,81	27,70	8,40
widgets/imageviewer	8,57	8,31	7,00	8,06	2,41
widgets/lineedits	8,16	7,95	6,93	8,05	2,44
widgets/movie	8,32	8,11	6,87	7,90	2,41
widgets/scribble	14,90	14,62	12,45	14,24	4,39

Tabelle A.38.: Übersetzungszeit 9/10

Name	*Kompilierzeit(s)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
widgets/shapedclock	7,85	7,76	6,65	7,73	2,37
widgets/sliders	14,00	13,74	11,69	13,60	4,14
widgets/spinboxes	8,39	8,10	7,03	8,03	2,49
widgets/stylesheet	24,49	24,17	19,62	22,22	6,63
widgets/tablet	24,61	24,00	20,39	23,57	7,07
widgets/tetrix	21,41	20,08	17,20	19,41	5,76
widgets/tooltips	15,97	15,78	13,71	15,86	4,85
widgets/wiggly	13,70	13,49	11,66	13,58	4,11
widgets/windowflags	14,52	14,19	12,17	14,14	4,17
xml/dombookmarks	15,18	14,98	13,08	15,01	4,65
xml/htmlinfo	2,84	2,67	2,42	2,60	0,82
xml/rsslisting	11,20	11,07	9,53	11,01	3,62
xml/saxbookmarks	17,60	17,47	15,14	17,38	5,27
xml/streambookmarks	17,49	17,38	14,89	17,08	5,29
xml/xmlstreamlint	1,74	1,65	1,62	1,65	0,50

Tabelle A.40.: Übersetzungszeit 10/10

Name	Codeumfang (bytes)				
	alias	*getset*	*operator*	*trunk*	*gnu*
animation/animatedtiles	116939	116699	116699	115019	108555
animation/easing	40882	40658	36642	33426	19922
animation/moveblocks	14009	13817	13481	12153	7705
dbus/chat	40881	40353	35857	33905	19185
dbus/complexpingpong	22386	21858	21570	21122	22386
dbus/listnames	14289	14289	14289	14105	9241
dbus/pingpong	10514	10258	10258	10050	10514
dbus/remotecontrolledcar/car	13129	12713	12169	11353	7225
dbus/remotecontrolledcar/controller	16857	16473	15321	14841	8457
declarative/cppextensions/imageprovider	4399	3991	3991	3774	1820
declarative/cppextensions/networkaccessmanagerfactory	10142	10142	10094	9630	6814
declarative/cppextensions/plugins	9463	8735	8143	7423	4551
declarative/cppextensions/qgraphicslayouts/layoutitem	5245	5245	5245	5069	3965
declarative/cppextensions/qgraphicslayouts/qgraphicsgridlayout	32836	32516	30468	28916	18596
declarative/cppextensions/qgraphicslayouts/qgraphicslinearlayout	27102	26718	26366	25246	18654
declarative/cppextensions/qwidgets	7067	6451	6211	6195	4219
declarative/cppextensions/referenceexamples/adding	9936	9680	9568	9504	7408
declarative/cppextensions/referenceexamples/attached	30935	30427	30155	29659	23787
declarative/cppextensions/referenceexamples/binding	42951	42283	41899	41035	31803
declarative/cppextensions/referenceexamples/coercion	20757	20325	20165	19781	16341
declarative/cppextensions/referenceexamples/default	20724	20292	20132	19748	16308
declarative/cppextensions/referenceexamples/extended	11983	11775	11519	10719	8175

Tabelle A.42.: Programmgröße 1/10

Name	Codeumfang (bytes)				
	alias	*getset*	*operator*	*trunk*	*gnu*
declarative/cppextensions/referenceexamples/grouped	26458	25978	25706	25130	20426
declarative/cppextensions/referenceexamples/properties	15968	15600	15440	15104	12096
declarative/cppextensions/referenceexamples/signal	31687	31147	30875	30363	24187
declarative/cppextensions/referenceexamples/valuesource	40982	40330	39930	39274	31050
declarative/imageprovider	4399	3991	3991	3774	1820
declarative/objectlistmodel	9685	9397	9397	9397	7029
declarative/plugins	9463	8735	8143	7423	4551
designer/calculatorbuilder	432735	432527	432175	431919	429615
designer/calculatorform	16833	16641	14017	13361	6689
designer/containerextension	20563	19763	18611	17715	8707
designer/taskmenuextension	26972	26140	24876	22636	10892
designer/worldtimeclockbuilder	427743	427743	427743	427727	426831
desktop/screenshot	16361	16105	14377	13449	6313
desktop/systray	34581	34325	31509	30181	20869
dialogs/classwizard	132887	132455	128919	127703	100535
dialogs/configdialog	39489	39233	38737	38401	29809
dialogs/extension	8993	8801	7585	6945	3857
dialogs/findfiles	24305	24113	22753	21793	10657
dialogs/licensewizard	71281	70785	68497	67729	56113
dialogs/standarddialogs	40305	40097	37617	36129	18417
dialogs/tabdialog	21313	20945	20657	20081	12321
dialogs/trivialwizard	4847	4847	4847	4783	2953

Tabelle A.44.: Programmgröße 2/10

Name	Codeumfang (bytes)				
	alias	*getset*	*operator*	*trunk*	*gnu*
draganddrop/draggableicons	19584	19584	19584	19184	14320
draganddrop/draggabletext	15524	15524	15524	14660	7396
draganddrop/dropsite	22105	21817	20825	20041	11001
draganddrop/fridgemagnets	18139	18139	18139	17083	8459
draganddrop/puzzle	78851	78355	77779	75971	61811
effects/lighting	11121	10865	10561	9313	5729
gestures/imagegestures	24601	24217	22921	20889	12297
graphicsview/anchorlayout	6105	6105	6105	5489	3241
graphicsview/collidingmice	21442	21330	19770	16698	10714
graphicsview/diagramscene	104038	103494	96006	90406	49350
graphicsview/dragdroprobot	45929	45833	45593	44553	33137
graphicsview/elasticnodes	37761	37473	36105	32265	16633
graphicsview/padnavigator	92560	92144	89376	86704	66864
graphicsview/simpleanchorlayout	6697	6697	6697	6601	3657
graphicsview/weatheranchorlayout	112167	111975	111719	110743	104855
help/contextsensitivehelp	41193	40953	35369	33785	18521
help/remotecontrol	38528	38208	32208	30368	16464
help/simpletextviewer	37225	36937	33721	32121	17049
ipc/localfortuneclient	12401	12145	10929	10209	5665
ipc/localfortuneserver	11897	11641	11241	10761	5833
ipc/sharedmemory	13425	13217	12417	11969	6385
itemviews/addressbook	53361	52721	49617	46817	27073

Tabelle A.46.: Programmgröße 3/10

Name	Codeumfang (bytes)				
	alias	*getset*	*operator*	*trunk*	*gnu*
itemviews/basicsortfiltermodel	20489	20281	18169	17497	10793
itemviews/chart	60021	59781	57349	52405	23429
itemviews/coloreditorfactory	13721	13481	13481	13321	9209
itemviews/combowidgetmapper	15033	14777	13353	12313	8329
itemviews/customsortfiltermodel	26553	26233	24265	23129	12985
itemviews/dirview	1745	1745	1745	1673	918
itemviews/editabletreemodel	52366	51998	48670	45486	24430
itemviews/fetchmore	11473	11137	10753	9905	5809
itemviews/pixelator	28085	27605	26677	24565	12245
itemviews/puzzle	84307	83811	82995	80467	64067
itemviews/simpledommodel	20113	19745	19121	17921	9425
itemviews/simpletreemodel	19118	18862	18526	17614	11150
itemviews/simplewidgetmapper	13497	13241	11945	10985	7641
itemviews/spinboxdelegate	5241	5033	5033	4633	2297
itemviews/stardelegate	25257	24857	24441	23193	13577
layouts/basiclayouts	11041	10785	9905	8625	5089
layouts/borderlayout	13921	13729	13361	11777	6465
layouts/dynamiclayouts	14545	14353	12737	11521	6529
layouts/flowlayout	13161	12969	12809	11289	6041
linguist/arrowpad	8353	8033	7425	6977	4705
linguist/hellotr	1681	1681	1681	1625	1093
linguist/trollprint	12489	12105	11017	10521	6345

Tabelle A.48.: Programmgröße 4/10

Name	Codeumfang (bytes)				
	alias	*getset*	*operator*	*trunk*	*gnu*
mainwindows/application	39363	39171	36547	35619	22403
mainwindows/dockwidgets	40132	39876	37652	36452	22772
mainwindows/mdi	59555	59255	54871	53047	30215
mainwindows/menus	36065	35873	30961	30193	15985
mainwindows/recentfiles	26913	26705	25217	23921	12385
mainwindows/sdi	43491	43303	40439	39079	23607
network/blockingfortuneclient	20697	20297	18937	17561	9289
network/broadcastreceiver	6425	6153	5769	5513	3465
network/broadcastsender	7617	7425	6673	6177	3793
network/download	12385	12193	12193	11825	6193
network/downloadmanager	19649	19361	18529	17121	9137
network/fortuneclient	22081	21825	20081	19041	9393
network/fortuneserver	20185	19993	19337	18601	9193
network/googlesuggest	19961	19657	18201	17177	8681
network/http	37257	37001	33321	31673	14825
network/loopback	17081	16593	14769	13441	6961
network/qftp	44955	44699	41147	39899	20379
network/securesocketclient	65395	65091	56595	53939	27027
network/threadedfortuneserver	17617	17233	16929	16289	8993
network/torrent	432025	429577	400697	359593	211865
opengl/2dpainting	13329	12833	12497	11521	7825
opengl/framebufferobject	95388	95149	93021	88381	78333

Tabelle A.50.: Programmgröße 5/10

Name	*Codeumfang(bytes)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
opengl/framebufferobject2	19893	19893	19365	17237	11365
opengl/grabber	35785	34969	32041	27065	13641
opengl/hellogl	40737	40241	38065	33345	17985
opengl/overpainting	47553	47265	45281	38801	19473
opengl/pbuffers	44357	44085	41189	37333	23109
opengl/pbuffers2	95052	94637	92637	88013	78141
opengl/samplebuffers	9385	9385	9289	8401	4758
opengl/textures	27062	26598	26118	23638	18326
painting/basicdrawing	47512	47096	43416	42296	26584
painting/concentriccircles	10905	10537	9929	8217	4969
painting/fontsampler	55425	55233	51761	47425	25537
painting/imagecomposition	67469	67277	66493	65821	56141
painting/painterpaths	27025	26721	24609	22449	13697
painting/svgviewer	27406	27166	25102	23854	12734
painting/transformations	23881	23641	23273	22009	14121
qtconcurrent/imagescaling	26793	26601	25433	24745	19609
qtconcurrent/map	22569	22569	22569	22473	20713
qtconcurrent/progressdialog	10377	10329	10249	9753	8297
qtconcurrent/runfunction	6361	6361	6361	6361	4809
qtestlib/tutorial1	3009	2753	2753	2641	2065
qtestlib/tutorial2	4369	4177	4177	4001	2817
qtestlib/tutorial3	5841	5585	5585	5441	4753

Tabelle A.52.: Programmgröße 6/10

Name	*Codeumfang(bytes)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
qtestlib/tutorial4	10161	9969	9969	9649	8033
qtestlib/tutorial5	7761	7505	7505	6801	4225
richtext/calendar	15681	15473	15249	14657	7953
richtext/orderform	35249	34929	33729	32337	17073
richtext/syntaxhighlighter	25521	25201	24897	23649	17393
richtext/textobject	12057	11753	11257	10729	6505
script/calculator	431935	431935	431935	431919	430191
script/customclass	39761	39489	39041	37697	18513
script/defaultprototypes	14234	14010	14010	13818	8778
script/marshal	10225	10225	10225	9921	7569
script/qscript	45281	45009	44417	42289	18913
sql/cachedtable	15577	15305	14425	14041	9113
sql/drilldown	434069	433637	431381	428549	414789
sql/masterdetail	262420	262116	258740	256900	228180
sql/querymodel	18561	18241	18241	17249	11025
sql/relationaltablemodel	15329	15329	15329	15169	11921
sql/sqlwidgetmapper	17825	17569	15937	15041	10641
sql/tablemodel	9467	9467	9467	9197	6385
statemachine/eventtransitions	3337	3337	3337	3337	2377
statemachine/factorial	8273	8049	7713	6897	4225
statemachine/pingpong	4513	4513	4513	4449	2593
statemachine/rogue	14705	14337	13377	11505	7073

Tabelle A.54.: Programmgröße 7/10

Name	*Codeumfang(bytes)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
statemachine/trafficlight	10329	10137	9673	9225	6009
statemachine/twowaybutton	2041	2041	2041	1905	1131
threads/mandelbrot	24609	24273	22569	16793	9177
threads/semaphores	11561	11561	11337	10841	9961
threads/waitconditions	12329	12281	11897	11257	10153
tools/codecs	31913	31545	29801	28153	15289
tools/completer	41725	41469	39485	38461	26685
tools/contiguouscache	7689	7321	6905	6169	3737
tools/customcompleter	31839	31471	30479	29727	19935
tools/echoplugin	9241	8969	8217	7689	4665
tools/i18n	36781	36477	35133	33437	25021
tools/inputpanel	37689	37257	30777	28857	14553
tools/plugandpaintplugins/basictools	0	0	0	0	0
tools/plugandpaintplugins/extrafilters	0	0	0	0	0
tools/regexp	20593	20337	18113	16881	8801
tools/settingseditor	83841	83377	78337	75025	39121
tools/styleplugin	3817	3609	3609	3609	2553
tools/treemodelcompleter	28836	28532	27188	25972	13364
tools/undoframework	35007	34623	31631	29903	16367
touch/dials	8111	8111	7003	6437	2987
touch/fingerpaint	34521	34265	32761	30985	17113
touch/knobs	4281	4281	4281	4169	2185

Tabelle A.56.: Programmgröße 8/10

Name	$Codeumfang(bytes)$				
	alias	*getset*	*operator*	*trunk*	*gnu*
touch/pinchzoom	26602	26170	24594	21394	14130
tutorials/addressbook/part1	4073	3817	3657	3481	2521
tutorials/addressbook/part2	14025	13897	12665	12425	6825
tutorials/addressbook/part3	19129	18873	17113	16505	8617
tutorials/addressbook/part4	27513	27257	25081	24217	12601
tutorials/addressbook/part5	34817	34449	31777	30737	16465
tutorials/addressbook/part6	43185	42817	39713	38433	20689
tutorials/addressbook/part7	51665	51297	47857	46449	24881
uitools/multipleinheritance	18089	17897	15065	14329	7273
uitools/textfinder	438031	437759	437327	436847	432767
widgets/analogclock	6497	6305	6305	5601	3457
widgets/calculator	31721	31353	29353	26617	13881
widgets/calendarwidget	41281	41089	35297	33793	18177
widgets/charactermap	30961	30625	28129	25633	13313
widgets/codeeditor	13281	13073	12673	12049	6849
widgets/digitalclock	4313	4057	4057	3977	2633
widgets/groupbox	11513	11257	11257	11257	6393
widgets/icons	55649	55121	51393	47761	24593
widgets/imageviewer	21953	21697	19057	17745	9073
widgets/lineedits	18873	18601	17657	17481	10601
widgets/movie	18617	18361	15273	14073	6841
widgets/scribble	33385	33145	30953	29017	14665

Tabelle A.58.: Programmgröße 9/10

Name	*Codeumfang(bytes)*				
	alias	*getset*	*operator*	*trunk*	*gnu*
widgets/shapedclock	9729	9473	9473	8593	4337
widgets/sliders	16409	16073	13321	12249	7513
widgets/spinboxes	22969	22777	21465	21049	11833
widgets/stylesheet	73405	73165	65197	63117	40493
widgets/tablet	36721	36305	31121	27473	14385
widgets/tetrix	36929	36241	33009	24801	11457
widgets/tooltips	25154	24974	24526	22862	11406
widgets/wiggly	10065	9697	9617	8753	5249
widgets/windowflags	24905	24601	21161	19897	11161
xml/dombookmarks	36017	35713	34849	34193	17937
xml/htmlinfo	9529	9529	9529	9119	4545
xml/rsslisting	18609	18337	17553	16961	9201
xml/saxbookmarks	39513	39321	37513	36201	18729
xml/streambookmarks	34969	34777	33593	32745	15849
xml/xmlstreamlint	5001	5001	5001	4769	2224

Tabelle A.60.: Programmgröße 10/10

Literaturverzeichnis

[AÖ07] Alexandersson, Ruben ; Öhman, Peter: Implementing fault tolerance using aspect oriented programming. In: *Dependable Computing*. Springer, 2007, S. 57–74

[Blo70] Bloom, Burton H.: Space/time trade-offs in hash coding with allowable errors. In: *Communications of the ACM* 13 (1970), Nr. 7, S. 422–426

[BSS13] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Generative software-based memory error detection and correction for operating system data structures. In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* IEEE, 2013, S. 1–12

[CC07] Chen, Kung ; Chien, Chin-Hung: Extending the field access pointcuts of AspectJ to arrays. In: *Journal of Software Engineering Studies* 2 (2007), Nr. 2, S. 2–11

[Hin01] Hind, Michael: Pointer analysis: haven't we solved this problem yet? In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* ACM, 2001, S. 54–61

[KHH+01] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An overview of AspectJ. In: *ECOOP 2001—Object-Oriented Programming*. Springer, 2001, S. 327–354

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-oriented programming. In: *ECOOP'97—Object-oriented programming*. Springer, 1997, S. 220–242

[LS06] Lohmann, Daniel ; Spinczyk, Olaf: Developing embedded software product lines with aspectc++. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* ACM, 2006, S. 740–742

[Mag06] Magnusson, Johan: *Set and Get in AspectC++*, MS Thesis. Department of Computer Science and Engineering, Chalmers University of Tecnology, Goteborg, Diplomarbeit, 2006

[MK03] Masuhara, Hidehiko ; Kawauchi, Kazunori: Dataflow pointcut in aspect-oriented programming. In: *Programming Languages and Systems*. Springer, 2003, S. 105–121

[Sch15] Schröder, Simon: *Entwicklung und Bewertung von Advice für vordefinierte Operatoren in AspectC++*, Bachelorarbeit. Arbeitsgruppe Eingebettete Systemsoftware, Lehrstuhl Informatik 12, Technische Universität Dortmund, Bachelorarbeit, 2015

[SL07] Spinczyk, Olaf ; Lohmann, Daniel: The design and implementation of AspectC++. In: *Knowledge-Based Systems* 20 (2007), Nr. 7, S. 636–651

Abbildungsverzeichnis

Tabellenverzeichnis

Programm-Listings