technische universität
dortmund

Bachelorarbeit

Entwicklung und Bewertung von Advice für vordefinierte Operatoren in AspectC++

Simon Schröder
27. Februar 2015

Betreuer:
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 27. Februar 2015

Simon Schröder

Zusammenfassung

AspectC++ erweitert die Programmiersprache C++ um aspektorientierte Programmierung und ermöglicht auf diese Weise eine zentrale Erweiterung sowie Konfigurierbarkeit von in einem Quelltext verstreuten Komponenten. Die zurzeit existierende Implementierung des AspectC++-Compilers bietet allerdings keine Möglichkeit, um in einem Advice zu definieren, dass ein zusätzlicher Quelltext auch an vordefinierten Operatoren eingewebt werden soll. Da diese Advice für vordefinierte Operatoren allerdings für die komfortable Erkennung von transienten Fehlern, wie den sogenannten Soft-Errors, unerlässlich sind, befasst sich diese Arbeit sowohl mit der Implementierung dieser neuen Advice als auch mit den daraus resultierenden Änderungen an der Sprachebene von AspectC++. Mit den Ergebnissen der abschließenden simulationsbasierten Evaluation der Fehlertoleranz kann die Relevanz der Implementierung für die Praxis bewertet werden.

Inhaltsverzeichnis

1 Einleitung

> „As the dimensions and operating voltages of computer electronics are reduced to satisfy the consumer's insatiable demand for higher density, functionality, and lower power, their sensitivity to radiation increases dramatically."
> [1, S. 1]

Computersysteme und insbesondere eingebette Systeme sind in den letzten Jahren immer leistungsfähiger und kleiner geworden. Dies hat den Vorteil, dass sie sich immer unauffälliger in das Leben der Menschen einfügen und zusätzlich immer komplexere und wichtigere Aufgaben übernehmen können. Allerdings entstehen bereits durch die Reduzierung der Größe Nachteile. So nimmt die Energiedichte innerhalb der Systeme zu und die Fläche zur Ableitung von Wärme ab, sodass eine zusätzliche Kühlung, die die Reduzierung der Größe wieder zunichte machen würde, erforderlich wäre. Um dies zu vermeiden, wird zusätzlich die Betriebsspannung und somit die Energiedichte gesenkt. Doch besonders die Kombination aus Reduzierung der Größe und Absenkung der Betriebsspannung begünstigt transiente Fehler und inbesondere sogenannte *Soft-Errors*[1][3, S. vii] dramatisch[1, S. 1].
Dies liegt daran, dass diese Soft-Errors dann entstehen, wenn energiegeladene Neutronen aus dem Weltraum oder Alphateilchen aus umgebenen Gegenständen, wie zum Beispiel Verpackungsmaterialien, die Transistoren eines Speicherbausteines oder einer Logikeinheit treffen und hierdurch Bits flippen oder Störungen in Berechnungen verursachen[1, S. 1]. Durch die angesprochene Verdichtung der Systeme steigt somit die Wahrscheinlichkeit, dass Teilchen empfindliche Bereiche eines Systems treffen und in ihnen Soft-Errors verursachen[3, S. vii][4, S. 1]. Zusätzlich sinkt durch die geringere Betriebsspannung die Ladungsmenge, die ein Teilchen mitbringen muss, um beispielsweise den Zustand einer Speicherzelle zu verändern.
Früher waren Soft-Errors vor allem für Anwendungen in der Raumfahrt ein Problem, doch mittlerweile stellen diese auch auf der Erde ein ernsthaftes Problem dar[4, S. 1]. Außerdem betreffen diese transienten Fehler nicht nur hochspezielle Systeme, sondern können auch in gewöhnlichen Servern auftreten. So wurde in einer im Jahr 2009 veröffentlichten Studie[5] festgestellt, dass in ungefähr einem Drittel der Maschinen in der Serverfarm des Unternehmens „Google" mindestens ein Soft-Error pro Jahr auftritt[5, Kap. 3.1].
Aus den genannten Gründen besteht zunehmend die Notwendigkeit bestehende Anwendungen mit möglichst geringem Aufwand zu erweitern, sodass diese die Soft-Errors erkennen können und damit eine Chance haben, diese Fehler zu beheben oder in anderer Weise zu behandeln.

[1] Neu eingeführte Begriffe und insbesondere Fachbegriffe werden in dieser Arbeit während ihrer ersten Verwendung kursiv hervorgehoben[2, S. 1].

1.1 Zielsetzung

Ziel dieser Arbeit ist es, für in der Programmiersprache C++ geschriebe Anwendungen eine komfortable Lösung bereitzustellen, mit der die soeben beschriebenen Soft-Errors erkannt werden können. Diese Lösung soll mit AspectC++ realisiert werden und einen geringen Overhead zur Ausführungszeit aufweisen. Da die für die komfortable Erkennung der Soft-Errors notwendige Unterstützung von Advice für vordefinierte Operatoren in AspectC++ momentan nicht existiert, ist die Erweiterung von AspectC++ um diese Advice ein weiteres unerlässliches Ziel dieser Arbeit. Inwiefern die Implementierung tatsächlich eine Verbesserung der Fehlertoleranz darstellt, soll außerdem durch eine praxisnahe Evaluation bestätigt werden.

1.2 Struktureller Aufbau dieser Arbeit

Nach der einleitenden Motivation werden in dem zweiten Kapitel die genauen Anforderungen an die Sprachebene von AspectC++ und weitere Rahmenbedingungen dieser Arbeit vorgestellt. Das darauf folgende Kapitel 3 beinhaltet eine Untersuchung des Ablaufs sowie der Komponenten des AspectC++-Compilers bezüglich der für die vordefinierten Operatoren notwendigen Änderungen. Die Änderungen der sich daraus ergebenen Komponenten werden daraufhin konkret geplant und implementiert. Die während der Implementierung auftretenden Probleme werden in Kapitel 4 zusammen mit weiteren Situationen, in denen die Implementierung von dem Entwurf abweicht, beschrieben. Inwiefern das Ergebnis, insbesondere bezüglich der Fehlertoleranz, eine Verbesserung darstellt, wird in dem fünften Kapitel evaluiert. Abschließend folgt ein Fazit dieser Arbeit sowie ein Ausblick über weiterführende Verbesserungen und Erweiterungen, die in dem Rahmen dieser Arbeit nicht möglich waren.

2 Anforderungen

2.1 C++

Die Programmiersprache C++[6] bietet eine Vielzahl unterschiedlicher Arten von Objekten. Eine Art sind die sogenannten Zeiger. Ein Zeiger ist ein ganzzahliger positiver Wert, der als Speicheradresse eines anderen Objektes interpretiert wird und sich ebenfalls im Speicher an einer bestimmten Adresse befindet. Aus diesem Grund können natürlich auch Zeiger durch Soft-Errors verfälscht werden. Der Unterschied zu den meisten anderen Objekten ist allerdings die Tatsache, dass der Wert des Zeigers bzw. die Adresse, auf die der Zeiger zeigt, nicht beliebige Werte seines Wertebereichs annehmen darf, denn ein Zeiger muss in ein Speichersegment der aktuell ausgeführten Anwendung zeigen. Wenn dies nicht der Fall ist, dann kann daran das Auftreten eines Fehlers erkannt werden. Der Versuch, den fehlerhaften Zeiger in einem solchen Fall zu *dereferenzieren*, das heißt, die Daten an der Speicheradresse, auf die der Zeiger zeigt, zu lesen, führt im besten Fall zu einer durch den Betriebssystemkern erkannten Zugriffsverletzung (engl. *segmentation fault*) oder im schlechtesten Fall zu fehlerhaftem Verhalten der Anwendung oder zu einem Absturz des kompletten Systems.
Durch den eingeschränkten gültigen Wertebereich des Zeigers eröffnet sich allerdings auch die Möglichkeit vor der Dereferenzierung zu überprüfen, ob der Zeiger nicht in ein Speichersegment der Anwendung zeigt und somit fehlerhaft ist. Auch eine Unterscheidung zwischen einem sogennanten *Funktions-Zeiger*, der die Adresse einer Funktion speichert, und einem Objekt-Zeiger ist anhand von Templates möglich[7, Kap. 6]. Demzufolge kann je nach Art des Zeigers das erlaubte Speichersegment weiter eingeschränkt werden.
Auch in einer arithmetisch-logischer Einheit(ALU) können Soft-Errors Probleme verursachen, indem zum Beispiel die Operanden, Zwischenergebnisse oder das Resultat einer Berechnung verfälscht wird. Mit Hilfe der sogenannten *Residue-Codes*[8, S. 9] wird eine Validierung von ganzzahligen Berechnungen, wie zum Beispiel einer Addition, Subtraktion, Multiplikation oder Division, möglich.

Weiterhin enthält die Programmiersprache C++ standardmäßig Operatoren, die von dem C++-Compiler in dem globalen Namensraum definiert werden und somit überall in einem C++-Quelltext verwendet werden können. Die Funktionsdefinitionen dieser Operatoren existieren allerdings nur implizit und sind daher für den Benutzer nicht sichtbar. Diese, in der Programmiersprache C++ eingebauten Operatoren, werden in dieser Arbeit als *vordefinierte Operatoren* bezeichnet.
Ein weiteres Merkmal der vordefinierten Operatoren ist, dass die als Argumente und

Rückgabewert verwendeten Objekte nicht aus einer Klasse instanziiert wurden[1][6, Kap. 13.6:1][2] und diese Operatoren daher nicht überladbar sind[3,4].
Um die in der Einleitung beschriebenen Anwendungsfälle zu realisieren, sind Überprüfungen an Aufrufen der folgenden vordefinierten Operatoren notwendig:

- Dereferenzierungs-Operator `*`
- Additions-Operator `+`
- Subtraktions-Operator `-`
- Multiplikations-Operator `*`
- Divisions-Operator `/`

Für die Zeiger-Validierung ist es zusätzlich sinnvoll auch Validierungen an impliziten Dereferenzierungen zu erstellen. Solche impliziten Dereferenzierungen treten beispielsweise in dem Member-Zugriffs-Operator `->` oder in einem Aufruf eines Funktions-Zeigers auf.
Die Implementierung der Advice für die genannten vordefinierten Operator ist die minimale Anforderung. Um auch andere Anwendungsfälle abzudecken, sind optional möglichst viele weitere Advice für vordefinierte Operatoren zu implementieren. Die vordefinierten Operatoren, die im Kapitel 13.6 des C++-Standards[6] aufgeführt werden, sind dabei das grobe Ziel.

2.2 AspectC++

Sowohl die Zeiger-Validierung als auch die Überprüfung der ganzzahligen Berechnungen setzen jedoch voraus, dass in einer Anwendung an den entsprechenden Stellen die notwendigen Überprüfungen durchgeführt werden.
Da die Programmiersprache C++ das Überladen des Dereferenzierungs-Operators und der arithmetischen ganzzahligen Operatoren nicht erlaubt[6, Kap. 13.6:1] müssen alle Vorkommen dieser Operatoren im Quelltext um eine entsprechende Überprüfung erweitert werden. Da solche Operatoren in gewöhnlichem C++-Quelltext häufig vorkommen, sind die Erweiterungen über den gesamten Quelltext verstreut und können daher als ein querschneidender Belang oder engl. *crosscutting concern* bezeichnet werden. Für querschneidende Belange bietet sich die aspektorientierte Programmierung (AOP) an, da diese durch sogenannte *Aspekte* eine Beschreibungsvorschrift bietet, um zentral festzulegen,

[1] Diese Nicht-Klassen-Typen sind Fundamentale Typen[6, Kap. 3.9.1], Array-Typen[6, Kap. 8.3.4], Referenz-Typen[6, Kap. 8.5.3], Zeiger-Typen[6, Kap. 8.3.1] oder Aufzählungs-Typen[6, Kap. 7.2].
[2] Angaben zu dem Absatz des Zitates befinden sich in dieser Arbeit hinter einem Doppelpunkt (:). In diesem Fall ist folglich der Absatz 1 des Kapitels 13.6 zitiert worden.
[3] Vordefinierte Operatoren auf Aufzählungen können überladen werden.
[4] Der Kopier-Zuweisungs-Operator[6, Kap. 12.8:17] stellt diesbezüglich eine Ausnahme dar, da er zum einen nicht global sondern implizit in jeder Klasse von dem Compiler automatisch definiert und dadurch trotzdem vordefiniert ist. Zum anderen ist er überladbar, denn er wird in einer Klasse als gelöscht bezeichnet und nicht mehr verwendet, sobald der Benutzer einen eigenen Kopier-Zuweisungs-Operator in dieser Klasse definiert.

an welchen Positionen des Quelltextes zusätzlicher Quelltext eingefügt werden soll[9, S. 30].
Zur Ergänzung der Programmiersprache C++ um die aspektorientierte Programmierung eignet sich die Erweiterung *AspectC++*[10]. Diese Erweiterung beinhaltet einen Quelltext-Zu-Quelltext-Compiler oder engl. *Source-To-Source-Compiler*, der als Eingabe einen C++-Quelltext und Aspekte erhält und als Ausgabe C++-Quelltext, in den die Aspekte eingewebt wurden, generiert.
Für die Analyse des Eingabe-C++-Quelltextes verwendet der AspectC++-Compiler das Clang-Frontend[11]. Dieses wurde für C-artige Sprachen[5] als Frontend[6] des LLVM-Compiler-Backends[7] entwickelt und bietet eine umfassende Programmierschnittstelle oder engl. *application programming interface*(API), über die der AspectC++-Compiler das Clang-Frontend zur Analyse und Veränderung von Quelltexten nutzen kann.
Das aktuell als Alternative unterstützte *Puma*-Frontend[14] ist durch die fehlende Unterstützung der Version *C++0x* der Programmiersprache C++ nicht mehr aktuell und wird daher in dieser Arbeit nicht weiter betrachtet.

Das Einweben in den durch das Frontend analysierten C++-Quelltext nimmt der AspectC++-Compiler auf Basis der sogenannten *Advice*, die innerhalb eines Aspektes definiert werden, vor. In Abbildung 2.1 ist innerhalb der türkisfarbenen Markierung ein solcher Advice zu sehen.

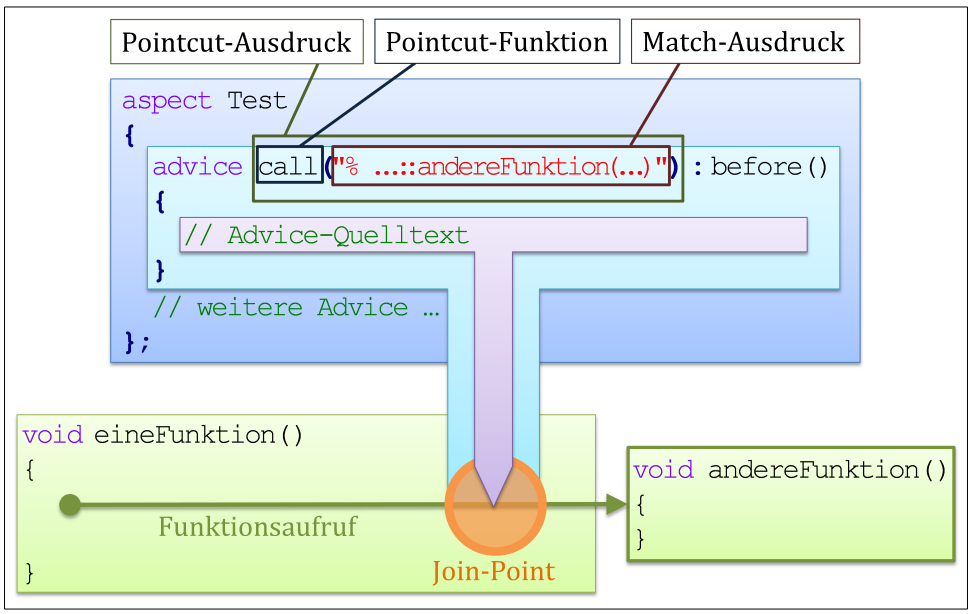

Abbildung 2.1: Schaubild, welches wichtige Begriffe und Zusammenhänge bezüglich AspectC++ aufzeigt

[5] *C*, *C++*, *Objective C* und *Objective C++*[12]
[6] Das Frontend eines Compilers übernimmt die lexikalische, syntaktische und semantische Analyse des in der Quell-Sprache vorliegenden Quelltextes und stellt das Ergebnis dieser Analyse als von der Quell-Sprache unabhängigen Zwischencode dem Compiler-Backend zur Verfügung[13, S. 2].
[7] Das Backend eines Compilers optimiert den Zwischencode und generiert aus diesem anschließend den Quelltext in der Ziel-Sprache (oft Maschinenquelltext)[13, S. 2].

Dieser Advice wird von dem AspectC++-Compiler in dem Quelltext vor dem Aufruf der Funktion `andereFunktion` eingewebt. Solche Positionen im Quelltext, an denen ein Advice eingewebt werden kann, werden als *Join-Points* bezeichnet. Eine Menge, die Join-Points enthält, wird *Pointcut* genannt.
Ein Advice besteht, wie in der Abbildung 2.1 sichtbar ist, aus dem Schlüsselwort `advice`, einem *Pointcut-Ausdruck* und der Advice-Art. Ein Pointcut-Ausdruck setzt sich aus *Pointcut-Funktionen* (siehe Abschnitt 2.4) und *Match-Ausdrücken* zusammen, die jeweils einen Pointcut zurückgeben. Diese Pointcuts können in dem Pointcut-Ausdruck unter Verwendung der Operatoren `&&` (Schnitt), `||` (Vereinigung) und `!` (Inverse) kombiniert werden. Der Rückgabewert eines Pointcut-Ausdruckes ist die durch ihn beschriebene Menge von Join-Points und somit auch ein Pointcut.
Allgemein wird in dem C++-Quelltext an einem bestimmten Join-Point J der Advice-Quelltext des Advice A genau dann eingewebt, wenn sich der Join-Point J in dem Pointcut befindet, der durch den Pointcut-Ausdruck des Advice A zurückgeben wird.
Der Pointcut-Ausdruck in der Abbildung 2.1 gibt die Menge aller Join-Points, die sich an einem Aufruf befinden und die sich in dem durch den Match-Ausdruck beschriebenen Pointcut befinden, zurück. Der durch den Match-Ausdruck zurückgegebene Pointcut enthält jeden Join-Point, der sich an einer (Member-)Funktion[8] befindet, die

- in einem beliebigen Gültigkeitsbereich (`....::`) definiert wurde,
- den Namen `andereFunktion` hat,
- einen beliebigen Typ (`%`) des Rückgabewertes und
- beliebige Typen (`...`) der Argumente besitzt.

Der Pointcut-Ausdruck beschreibt folglich alle **Aufrufe** dieser Funktionen, da die *call*-Pointcut-Funktion alle Aufrufe der durch den Match-Ausdruck beschriebenen Funktionen zurückgibt.
Da der abgebildete Join-Point diese Bedingungen erfüllt und sich somit in der durch den Pointcut-Ausdruck beschriebenen Menge von Join-Points befindet, webt der AspectC++-Compiler den Advice-Quelltext des abgebildeten Advice an dem abgebildeten Join-Point ein.

Join-Points an Aufrufen von vordefinierten Operatoren existieren in der bisherigen Variante des AspectC++-Compilers[9] nicht. Infolgedessen besteht keine Möglichkeit die beschriebenen Überprüfungen an den Aufrufen des Dereferenzierungs-Operators sowie der arithmetischen ganzzahligen Operatoren einzuweben. Dies unter der Voraussetzung eines minimalen Laufzeit-Overheads des verwebten Quelltextes zu realisieren ist die zentrale Anforderung an diese Arbeit.

[8] Um auszudrücken, dass es sich in einem gegebenen Fall sowohl um eine Funktion als auch um eine Member-Funktion handeln kann, wird in dieser Arbeit die abkürzende Schreibweise *(Member-)Funktion* verwendet.

[9] Die bisherige Variante des AspectC++-Compilers, die keine Unterstützung für vordefinierte Operatoren bietet, wird in dieser Arbeit als ***bisherige***, ***nicht erweiterte*** oder ***nicht angepasste*** Variante bezeichnet.

Ein Teil der Unterstützung von Advice für vordefinierte Operatoren besteht darin, dass die Sprachebene von AspectC++ das Beschreiben solcher Advice ermöglicht. In den folgenden Abschnitten werden daher die Anforderungen an die Sprachebene von AspectC++ bezüglich der Unterstützung von Advice an vordefinierten Operatoren beschrieben. Ein Teil dieser Anforderungen sind in Abbildung 2.2 dargestellt.

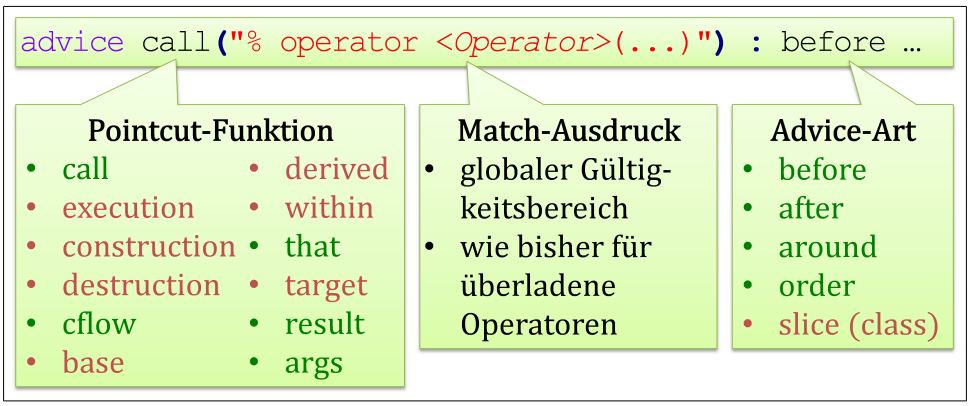

Abbildung 2.2: Übersicht der Advice-Arten und Pointcut-Funktionen sowie des Match-Ausdruckes. Die Farben Grün und Rot zeigen an, ob der jeweilige Begriff für vordefinierte Operatoren verfügbar oder nicht verfügbar sein soll. Die Match-Ausdrücke für vordefinierte Operatoren sollen den bisher für überladene Operatoren verwendeten Match-Audrücken gleichen.

Da die Sprachebene das Bindeglied zwischen dem Benutzer und dem AspectC++-Compiler darstellt, ist die grundsätzliche Anforderung, dass sich die Syntax und Semantik der Sprache AspectC++ für die vordefinierten Operatoren so weit wie möglich an der bestehenden Syntax und Semantik orientiert. Da durch die Match-Ausdrücke für überladenene Operatoren bereits eine Syntax zur Beschreibung von Operatoren vorliegt, betreffen die im Folgenden beschriebenen Anforderungen vor allem Einschränkungen bezüglich der Semantik.

2.3 Advice-Arten

AspectC++ bietet sechs Arten von Advice an, die innerhalb der Abbildung 2.2 rechts zu sehen sind. Die drei Arten *before*, *after* und *around* bewirken das Einweben des Advice Quelltextes vor, nach oder um die durch den Pointcut-Ausdruck beschriebenen Join-Points. Das Einweben von Quelltext mit diesen genannten Arten soll möglich sein.
Mit der Advice-Art *order* kann festgelegt werden in welcher Reihenfolge mehrere Advice von verschiedenen Aspekten an einem Join-Point eingewebt werden sollen. Auch diese Advice-Art soll mit Advice für vordefinierte Operatoren möglich sein.
Die beiden letzten Advice-Arten *slice* und *slice class : public* werden verwendet, um eine Klasse, in der oder für die sich die Join-Points des Advice befinden, um weitere (Member-)Funktionen oder Basisklassen zu erweitern. Da vordefinierte Operatoren globale implizite

Funktionen sind und somit mit keiner Klasse in Verbindung stehen, sind diese beiden Advice-Arten im Fall eines Advice für vordefinierte Operatoren nicht verfügbar.

2.4 Pointcut-Funktionen

Mit Hilfe von Pointcut-Funktionen[15] können Pointcuts manipuliert werden. Diese Manipulationen beinhalten das Verändern des Pointcuts durch das Entfernen oder Hinzufügen von Join-Points, aber auch die Manipulation der einzelnen Join-Points in dem Pointcut an sich. So kann beispielsweise mit der *call*-Pointcut-Funktion aus einem Pointcut mit einem Join-Point an einer Funktionsdefinition (*Namens-Join-Point*) ein Pointcut mit den Join-Points an den Aufrufen dieser Funktion (*Code-Join-Point*) erstellt werden.

Um in einem Pointcut-Ausdruck eines Advice zu definieren, dass der dazugehörige Advice-Quelltext nur an (Member-)Funktionsaufrufen eingewebt werden soll, existieren in AspectC++ zum einen die bereits erwähnte call-Pointcut-Funktion und zum anderen die *execution*-Pointcut-Funktion. Diese beiden Pointcut-Funktionen erwarten jeweils einen Pointcut mit Namens-Join-Points und geben einen Pointcut mit den Code-Join-Points an den entsprechenden Aufrufen zurück. Der Unterschied zwischen diesen beiden Pointcut-Funktionen besteht darin, dass die call-Pointcut-Funktion nur die sogenannten *Call-Code-Join-Points* zurückgibt, die sich an dem Aufruf und noch in dem Gültigkeitsbereich, in dem der Aufruf stattfindet, befinden. Die durch die execution-Pointcut-Funktion zurückgegebenen *Execution-Code-Join-Points* befinden sich im Gegensatz dazu in dem Gültigkeitsbereich der aufgerufenen (Member-)Funktion.
Da für einen vordefinierten Operator keine explizite Funktionsdefinition existiert, sind für diese auch keine Execution-Join-Points vorhanden. Daraus folgt, dass die execution-Pointcut-Funktion für Aufrufe an vordefinierten Operatoren immer einen leeren Pointcut zurückgeben würde und daher nicht verfügbar sein soll.

Der an einem bestimmten Join-Point eingewebte Advice-Quelltext eines Advice mit einer *cflow*-Pointcut-Funktion, wird nur ausgeführt, falls sich dieser Join-Point in dem dynamischen Ausführungsablauf eines durch das Argument der *cflow*-Pointcut-Funktion beschriebenen Join-Points befindet. Als Argument dieser *cflow*-Pointcut-Funktion sollen Join-Points an vordefinierten Operatoren nicht verfügbar sein, da ein vordefinierter Operator keine anderen (Member-)Funktionen aufruft und es somit keinen durch einen Aufruf eines vordefinierten Operators entstandenen Ausführungsablauf gibt. Das Einschränken der Ausführung des Advice-Quelltextes mit dieser *cflow*-Pointcut-Funktion soll allerdings auch im Fall eines Advice für vordefinierte Operatoren möglich sein.
Auch die *within*-Pointcut-Funktion soll mit Join-Points an vordefinierten Operatoren als Argument nicht verfügbar sein, da es keine Join-Points innerhalb oder engl. *within* eines vordefinierten Operators gibt.

Da die Pointcut-Funktionen *construction*, *destruction*, *base* und *derived* jeweils einen Pointcut mit Join-Points an Klassen oder Member-Funktionen als Argument erwarten, sollen diese nicht verfügbar sein, falls dieser Pointcut nur Join-Points an vordefinierten Operatoren enthält. Diese Einschränkung ist nötig, da vordefinierte Operatoren weder

Klassen noch Member-Funktionen sind.

Die *args*-, *result*-, *that*- und *target*-Pointcut-Funktionen erwarten als Argument keinen Pointcut sondern eine Beschreibung eines Typs. Daher sind hier keine Veränderungen nötig.

2.5 Join-Point-API

Mit Hilfe der Join-Point-API[16, S. 30] kann der Benutzer innerhalb des Advice-Quelltextes auf statische und dynamische Informationen des aktuellen Join-Points zugreifen. Solche Informationen sind zum Beispiel

- die Signatur des Join-Points,
- die Typen der Argumente,
- der Typ des Rückgabewertes,
- Zeiger auf die Argumente,
- ein Zeiger auf den Rückgabewert,
- ein Zeiger auf das Ziel-Objekt und
- ein Zeiger auf das umgebene Objekt.

Im Fall eines Join-Points an einem Aufruf eines vordefinierten Operators sollen alle Informationen der Join-Point-API wie bisher zur Verfügung stehen. Allerdings wird der Zeiger auf das Ziel-Objekt des Aufrufs immer `0` sein, da vordefinierte Operatoren keine Member-Funktionen sind und somit nicht auf einem Objekt aufgerufen werden.

3 Entwurf

Nachdem im Kapitel 2 die Anforderungen formuliert worden sind, werden die dafür nötigen Änderungen an dem AspectC++-Compiler aufgezeigt und beschrieben.
Dazu ist in Abbildung 3.1 der schematische und vereinfachte Ablauf des Kompiliervorganges des AspectC++-Compilers abgebildet. Dieser Ablauf wird zunächst kurz beschrieben und nachfolgend auf Änderungen in Bezug auf die vordefinierten Operatoren untersucht.

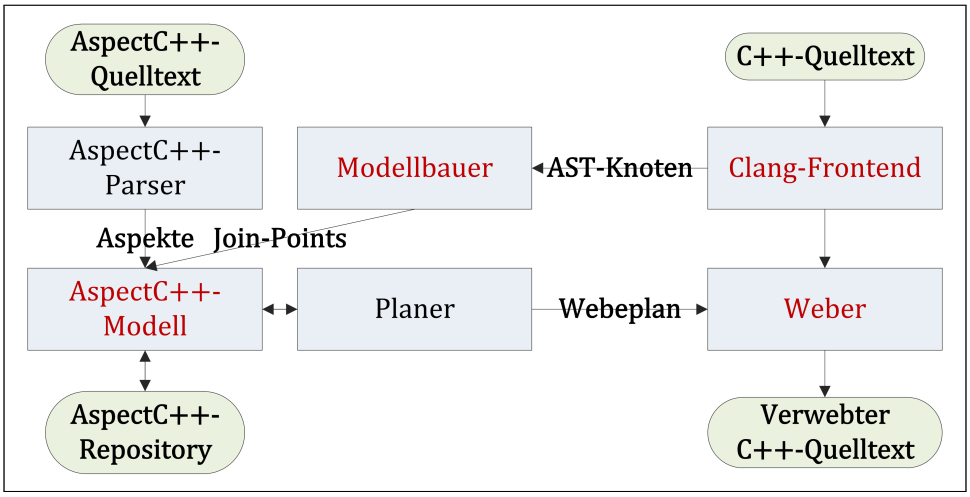

Abbildung 3.1: Schematischer und vereinfachter Ablauf des Kompiliervorganges des AspectC++-Compilers
Elemente, an denen im Zuge dieser Arbeit Änderungen vorgenommen werden müssen, sind rot markiert.

Zu Beginn des Ablaufs wird der Eingabe-C++-Quelltext von dem Clang-Frontend zur Generierung eines abstrakten Syntaxbaumes (AST) verwendet. Während dieser AST anschließend durchlaufen wird, werden aus benötigen AST-Knoten durch den Modellbauer Join-Points erstellt. Diese Join-Points werden danach zusammen mit den geparsten Aspekten des AspectC++-Quelltextes in dem internen *AspectC++-Modell* registriert. Der *Planer* gleicht die Join-Points mit den Pointcut-Ausdrücken der in den Aspekten vorhandenen Advice ab und erstellt hieraus einen Webeplan, der festlegt, an welchen Join-Points welcher Advice eingewebt wird. Dieser Webeplan wird schließlich von dem *Weber* unter Verwendung der durch das Clang-Frontend gewonnenen C++-Quelltext-Informationen umgesetzt. Der Eingabe-C++-Quelltext inklusive der eingewebten Aspekte stellt das Resultat dar.

Dieser Ablauf funktioniert momentan mit Advice für vordefinierte Operatoren nicht bzw. diese Advice werden nicht eingewebt. Dies liegt daran, dass

- die die vordefinierten Operatoren betreffenden AST-Knoten nicht aus dem AST extrahiert werden,
- das AspectC++-Modell Join-Points an vordefinierten Operatoren nicht verwalten kann,
- und dass der Weber die für die vordefinierten Operatoren spezifischen Schreibweisen nicht beherrscht.

Diese Probleme sind durch eine rote Schrift auch in Abbildung 3.1 kenntlich gemacht. Der Modellbauer ist ebenfalls rot markiert, da dieser die Join-Points in dem zu veränderen AspectC++-Modell registriert und daher bezüglich der Änderungen im AspectC++-Modell ebenfalls geändert werden muss.

3.1 Grundlagen

Damit die Änderungen an den den AST betreffenden Teilen des Clang-Frontend sowie dem AspectC++-Modell besser beschrieben werden können, enthält dieser Abschnitt grundlegende Informationen über den AST des Clang-Frontends und des AspectC++-Modells.

3.1.1 Abstrakter Syntaxbaum des Clang-Frontends

Ein abstrakter Syntaxbaum (engl. **A**bstract **S**yntax **T**ree, kurz AST) repräsentiert die syntaktische Struktur von einem Programmquelltext. Ein AST ist eine Baumstruktur, die unter anderem von Compilern während des Parsens eines Quelltextes angelegt und zur Speicherung von den aus dem Quelltext gewonnenen Informationen verwendet wird. Jeder Knoten des AST entspricht dabei einem Programmkonstrukt, wie zum Beispiel einer Variablendeklaration oder einer *if-else*-Verzweigung. Aber auch implizite Programmkonstrukte, wie z.B. implizite Konversionen, werden in den AST aufgenommen.
Die Kind-Knoten eines Knotens entsprechen den Programmkonstrukten, aus denen der Knoten zusammengesetzt ist. So hat beispielsweise ein *if-else*-Verzweigungs-Knoten drei Kind-Knoten: Einen Knoten, der die *if*-Bedingung darstellt, und zwei Knoten für den *if*- und *else*-Bereich.
Außerdem werden in den Knoten Eigenschaften des jeweiligen Programmkonstruktes und auch Metadaten, wie Zeilen- und Spaltennummer, gespeichert.
Im Gegensatz zu einem konkreten Syntaxbaum beinhaltet ein abstrakter Syntaxbaum nicht alle Merkmale des Quelltextes, wie z.B. Leerräume.

Das Clang-Frontend orientiert sich während des Generierens des Clang-AST, der im Folgenden nur *AST* genannt wird, im Vergleich zu anderen Compilern sehr nah am Quelltext [17]. So sind unter anderem Klammer-Ausdrücke, Kompilierzeit-Konstanten

und Kommentare in diesem AST vorhanden[17].
Für die Knoten existieren die drei Basisklassen *clang::Decl*, *clang::Stmt* und *clang::Type*, die eine Deklaration, ein Statement oder einen Typ beschreiben [18]. Jede dieser Klassen ist die Spitze einer Klassen-Hierarchie, in der für jedes Programmkonstrukt und viele Gruppen von Programmkonstrukten jeweils eine Klasse existiert. In Abbildung 3.2 sind Auszüge dieser Klassenhierarchien mit den für diese Arbeit wichtigen Klassen dargestellt.

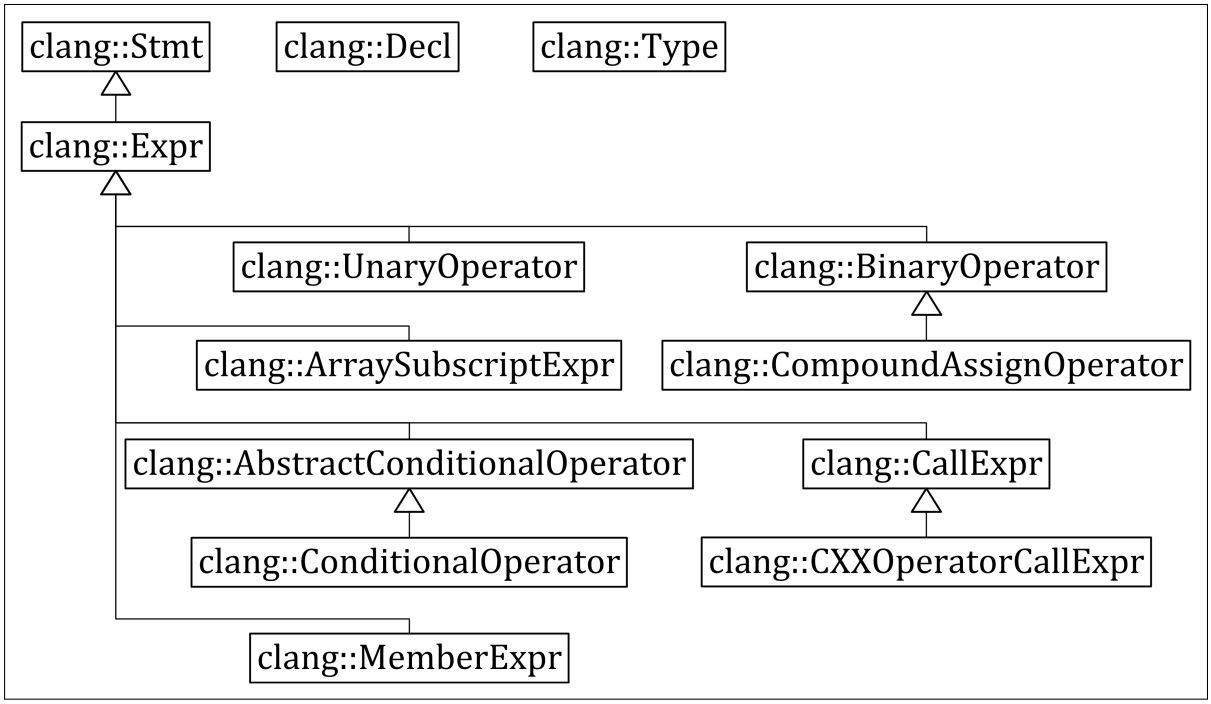

Abbildung 3.2: Die drei Basisklassen der AST-Knoten und die für diese Arbeit relevanten abgeleiteten Klassen der Basisklasse *clang::Stmt*

Ein Knoten des AST, der einem der in dieser Arbeit behandelten vordefinierten Operatoren entspricht, wird durch eine der Klassen *clang::UnaryOperator* (unäre vordefinierte Operatoren), *clang::BinaryOperator* (binäre vordefinierte Operatoren), *clang::ArraySubscriptExpr* (Array-Index-Operator `[]`) oder *clang::ConditionalOperator* (ternärer Operator `?:`) beschrieben. Diesen Klassen ist die gemeinsame Klasse *clang::Expr* übergeordnet, die allgemein einen Ausdruck beschreibt. Eine Basisklasse für vordefinierte Operatoren existiert aktuell nicht.

Mit dem Clang-Befehl *clang -Xclang -ast-dump -fsyntax-only <Name der Quelltext-Datei>* lässt sich der AST textuell ausgeben. In Abbildung 3.3 ist das Ergebnis der Ausführung des genannten Befehls für den C++-Quelltext 3.1 zu sehen.

```
1 int main() {
2   int i = 1;
3   int* p = &i;    // Adress-Operator
```

```
4    *p = i + 2;      // Binärer Plus-, Zuweisungs- und ↩
         Dereferenzierungs-Operator
5    if(*p == 1) {    // Dereferenzierungs- und boolescher Vergleichs-↩
         Operator
6      i = 2;         // Zuweisungs-Operator
7    }
8    else {
9      i = 3;         // Zuweisungs-Operator
10   }
11 }
```

Quelltext 3.1: C++-Quelltext zur Demonstrierung der textuellen Repräsentation des AST des Clang-Frontends. In den Kommentaren sind die in der jeweiligen Zeile vorkommenden vordefinierten Operatoren aufgelistet.

```
TranslationUnitDecl 0x1639560 <invalid sloc>
|-TypedefDecl 0x1639aa0 <invalid sloc> __int128_t '__int128'
|-TypedefDecl 0x1639b00 <invalid sloc> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x1639ec0 <invalid sloc> __builtin_va_list '__va_list_tag [1]'
`-FunctionDecl 0x1639f60 ast_beispiel.cpp:1:1 line:11:1 main 'int (void)'
  `-CompoundStmt 0x1681668 line:1:12 line:11:1
    |-DeclStmt 0x163a0e8 line:2:3 col:12
    | `-VarDecl 0x163a070 col:3 col:11 i 'int'
    |   `-IntegerLiteral 0x163a0c8 col:11 'int' 1
    |-DeclStmt 0x163a1e0 line:3:3 col:14
    | `-VarDecl 0x163a140 col:3 col:13 p 'int *'
    |   `-UnaryOperator 0x163a1c0 col:12 col:13 'int *'
    |     `-DeclRefExpr 0x163a198 col:13 'int' lvalue Var 0x163a070 'i' 'int'
    |-BinaryOperator 0x1681430 line:4:3 col:12 'int' lvalue
    | |-UnaryOperator 0x163a238 col:3 col:4 'int' lvalue
    | | `-ImplicitCastExpr 0x163a220 col:4 'int *' LValueToRValue
    | |   `-DeclRefExpr 0x163a1f8 col:4 'int *' lvalue Var 0x163a140 'p' 'int *'
    | `-BinaryOperator 0x1681408 col:8 col:12 'int'
    |   |-ImplicitCastExpr 0x16813f0 col:8 'int' LValueToRValue
    |   | `-DeclRefExpr 0x163a258 col:8 'int' lvalue Var 0x163a070 'i' 'int'
    |   `-IntegerLiteral 0x16813d0 col:12 'int' 2
    `-IfStmt 0x1681638 line:5:3 line:10:3
      |-<<<NULL>>>
      |-BinaryOperator 0x16814f0 line:5:6 col:12 '_Bool'
      | |-ImplicitCastExpr 0x16814d8 col:6 col:7 'int' LValueToRValue
      | | `-UnaryOperator 0x1681498 col:6 col:7 'int' lvalue
      | |   `-ImplicitCastExpr 0x1681480 col:7 'int *' LValueToRValue
      | |     `-DeclRefExpr 0x1681458 col:7 'int *' lvalue Var 0x163a140 'p' 'int *'
      | `-IntegerLiteral 0x16814b8 col:12 'int' 1
      |-CompoundStmt 0x1681588 col:15 line:7:3
      | `-BinaryOperator 0x1681560 line:6:5 col:9 'int' lvalue
      |   |-DeclRefExpr 0x1681518 col:5 'int' lvalue Var 0x163a070 'i' 'int'
      |   `-IntegerLiteral 0x1681540 col:9 'int' 2
      `-CompoundStmt 0x1681618 line:8:8 line:10:3
        `-BinaryOperator 0x16815f0 line:9:5 col:9 'int' lvalue
          |-DeclRefExpr 0x16815a8 col:5 'int' lvalue Var 0x163a070 'i' 'int'
          `-IntegerLiteral 0x16815d0 col:9 'int' 3
```

Abbildung 3.3: Textuelle Repräsentation des AST des Quelltextes 3.1

Die Baumstruktur des AST ist deutlich zu erkennen und spiegelt die geschachtelten Aufbau des C++-Quelltextes wider. Die Funktion `main` stellt ein Programmkonstrukt dar und wird in dem AST durch den einzigen Knoten des Typs *FunctionDecl* abgebildet. Dessen Kind-Knoten ist ein Knoten des Typs *CompoundStmt*, der den Funktions-Rumpf repräsentiert. Das Programmkonstrukt in der ersten Zeile innerhalb des Rumpfes, das eine Variablendeklaration darstellt, wird in dem AST durch einen Knoten des Typs *DeclStmt* abgebildet. Durch die Kind-Knoten dieses Knotens wird der Typ der deklarierten Variable und ihr Initialisierungswert beschrieben.
Die Zuweisung `=` innerhalb des sich in Zeile 4 befindenen Quelltextes `*p = i + 2;` setzt sich im AST aus einem Knoten des Typs *BinaryOperator*, der die rechte Seite der Zuweisung und somit die Addition repräsentiert, und aus einem Knoten des Typs *UnaryOperator* zusammen, der die Dereferenzierung der Variablen `p` darstellt.
Im weiteren Verlauf können die *if-else*-Verzweigung und die als deren Kind-Knoten dargestellte Bedingung, sowie Zuweisungen erkannt werden.
Auch viele Eigenschaften der AST-Knoten sind in dieser textuellen Darstellung aufgeführt. Die erste dieser Eigenschaften ist die Adresse, an der sich der jeweilige Knoten im Speicher befindet. Anschließend folgt eine Angabe der Position, an der sich das entsprechende Programmkonstrukt in dem C++-Quelltext befindet. Abhängig von dem Typ des Knotens sind außerdem Informationen bezüglich des Variablennamens, der Werte-Kategorie (siehe [6, Kap. 3.10:1]) oder des Typs vorhanden.

3.1.2 Internes Modell des AspectC++-Compilers

Das AspectC++-Modell ist eine interne Datenstruktur des AspectC++-Compilers, in der alle benötigten Informationen sowohl des geparsten C++Quelltextes als auch der geparsten Aspekte gespeichert werden. Das Modell teilt sich in drei verschiedene Gruppen von Klassen auf, die durch unterschiedliche Präfixe kenntlich gemacht werden.

Die Klassen mit dem Präfix *TI_* (von **T**ransform-**I**nformation) bilden eine Klassenhierarchie, in der für alle Programmkonstrukte, die aus dem C++-Quelltext geparst wurden, eine Klasse vorhanden ist. Objekte dieser Klassen speichern Informationen, die beim Einweben der Aspekte in den Quelltext verwendet werden und sind abhängig von dem Clang-Frontend.

In der Klassenhierarchie der Klassen mit dem Präfix *ACM_* (von **A**spect**C**++ **M**odel) ist ebenfalls für jedes Programmkonstrukt eine Klasse vorhanden. Die Objekte dieser Klassen sind allerdings unabhängig von dem Clang-Frontend und werden zur Speicherung von allgemeinen Informationen über die jeweiligen Programmkonstrukte genutzt. Zusätzlich existieren in dieser Klassenhierarchie Klassen, deren Objekte Informationen zu Konstrukten, die aus dem AspectC++-Quelltext geparst worden sind, speichern. Solche Konstrukte sind zum Beispiel die im Abschnitt 2.2 beschriebenen Aspekte, Advice oder Pointcuts. Des Weiteren können die Objekte, die aus einer der Klassen in dieser Klassenhierarchie instanziiert worden sind, als xml-Datei serialisiert werden. Das Ergebnis dieser Serialisierung wird als AspectC++-Repository bezeichnet und verwendet, um Informationen an die nachfolgenden Instanzen weiterzugeben, wenn zusammenhängen-

de Quelltext-Dateien nacheinander von jeweils einer eigenen Instanz des AspectC++-Compilers verarbeitet werden.

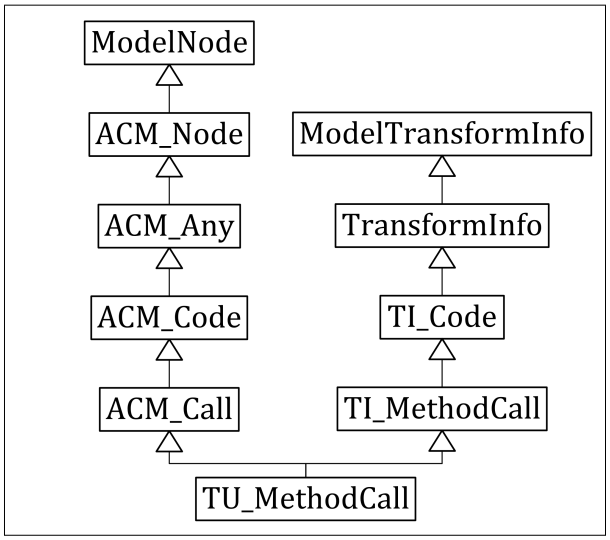

Abbildung 3.4: Basisklassen der Klasse *TU_MethodCall*

Das Präfix *TU_* (von **T**ranslation **U**nit) wird bei Klassen verwendet, die von der entsprechenden *ACM_*- und *TI_*-Klasse erben. Diese Klassen haben folglich sowohl Zugriff auf die allgemeinen als auch die zum Einweben benötigten Informationen und stellen - wie in Abbildung 3.4 erkenntlich - somit das Bindeglied zwischen den *ACM_*- und den *TI_*-Klassen dar.

3.2 Extrahierung der AST-Knoten

Informationen zu den einzelnen AST-Knoten können während des Durchlaufens des AST aus diesem erhalten werden. Dazu müssen die benötigten AST-Knoten in dem AST erkannt und deren Informationen zur weiteren Verarbeitung extrahiert werden.
Um dies zu realisieren, werden in dem Quelltext des AspectC++-Compilers die abstrakten Clang-Klassen *clang::ASTConsumer*[19] und *clang::RecursiveASTVisitor<Derived>*[20] in der Klasse *ClangASTConsumer* implementiert. Diese Klasse wird außerdem als Argument für das Template Argument *Derived* verwendet.
Der *ASTConsumer* stellt durch Überschreiben entsprechender Member-Funktionen Einstiegspunkte in den AST und Informationen über den AST-Kontext bereit[21]. AspectC++ verwendet zum Erhalten eines Einstiegspunktes die Member-Funktion *ASTConsumer::HandleTopLevelDecl*, die von dem Clang-Frontend für jede Übersetzungseinheit[1] aufgerufen wird. Über den Parameter ist ein Zugriff auf die sogenannten *Top-Level-*

[1]Eine Quelltext-Datei zusammen mit allen Header- und Quelltext-Dateien, die mit der Präprozessor-Anweisung `#include`(siehe [6, Kap. 16.2]) eingefügt wurden und abzüglich der durch die bedingte Inklusion des Präprozessors (siehe [6, Kap. 16.1]) übersprungenen Quelltext-Zeilen nennt man Übersetzungseinheit oder engl. *translation unit*[6, Kap. 2.1:1].

Deklarationen der Übersetzungseinheit, die sich auf der obersten Ebene des AST befinden, möglich. Ein Beispiel für eine Top- Level-Deklaration ist der AST-Knoten mit dem Typ *TranslationUnitDecl*, der sich in der ersten Zeile der textuellen Repräsentation des AST in Abbildung 3.3 befindet.
Von diese Top-Level-Deklaration lexikalisch [2] ausgehend werden für jeden nachfolgenden Knoten weitere Member-Funktionen in der Klasse *clang::RecursiveASTVisitor* aufgerufen.
Diese nachfolgenden Knoten werden in einer aus der sogenannten *Clang-Art* gebildeten Hierarchie weiter differenziert. Diese Clang-Art entspricht weitestgehend dem Typ des AST-Knotens, wird allerdings auch zu genaueren Differenzierung von AST-Knoten des gleichen Typs verwendet. So haben beispielsweise die AST-Knoten des Zuweisungs- und des binären Plus-Operators den gleichen Typ *clang::BinaryOperator*, allerdings die unterschiedliche Clang-Arten *BinAssign* bzw. *BinPlus*. Eine vollständige Liste mit allen Typen und Clang-Arten der AST-Knoten der vordefinierten Operatoren stellt die Tabelle A.1 dar, die sich im Angang befindet. Für jede Clang-Art in dieser Hierarchie existiert jeweils genau eine *Traverse-*, *WalkUpFrom-* und *Visit-Member-Funktion*. Die Namen dieser Member-Funktionen ergiben sich jeweils aus dem Präfix „Traverse“, „WalkUpFrom“ oder „Visit“ und der Clang-Art des AST-Knotens. So lautet der Name der Traverse-Member-Funktion für den AST-Knoten eines vordefinierten Dereferenzierungs-Operators „TraverseUnaryDeref“, da dieser durch den Typ *clang::UnaryOperator* und die Clang-Art *UnaryDeref* repräsentiert wird.
In dem Quelltext 3.2 befindet sich das genaue Schema, durch das die Reihenfolge der Aufrufe der verschiedenen Traverse-, WalkUpFrom- und Visit-Member-Funktionen festgelegt wird[20, Abschnitt „Detailed Description“].

```
1  // Einstiegspunkt für einen AST-Knoten "node":
2  RecursiveASTVisitor::Traverse#TOPLEVELKIND#(#TOPLEVELKIND#* node) {
3    Derived::Traverse#NODEKIND#(node);
4  }
5  RecursiveASTVisitor::Traverse#NODEKIND#(#NODETYPE#* node) {
6    Derived::WalkUpFrom#NODEKIND#(node);
7    for(int i = 0; i < node_child_count; ++i) {
8      Derived::traverse#NODECHILDRENKINDS(i)#(node_children(i))
9    }
10 }
11 RecursiveASTVisitor::WalkUpFrom#NODEKIND#(#NODETYPE#* node) {
12   if(#NODEKIND# != #TOPLEVELKIND#) {
13     Derived::WalkUpFrom#NODEBASEKIND#(node);
14   }
15   Derived::Visit#NODEKIND#(node);
16 }
17 Derived::WalkUpFrom#NODEKIND#(#NODETYPE#* node) {
```

[2] „Lexikalisch“ bedeutet, dass der Ort im Quelltext und nicht die semantische Einordnung gemeint ist[22, Beschreibung zu "clang_getCursorLexicalParent"].

```
18    // Entsprechende Member-Funktion der Basisklasse aufrufen:
19    RecursiveASTVisitor<Derived>::WalkUpFrom#NODEKIND#(#NODETYPE#* ↩
          node);
20  }
21  Derived::Visit#NODEKIND#(#NODETYPE#* node) {
22    // In Reihenfolge der PRE-Order-Tiefensuche: Visit-Quelltext
23  }
24  Derived::Traverse#NODEKIND#(#NODETYPE#* node) {
25    // Entsprechende Member-Funktion der Basisklasse aufrufen:
26    RecursiveASTVisitor<Derived>::Traverse#NODEKIND#(#NODETYPE#* node↩
          );
27    // In Reihenfolge der POST-Order-Tiefensuche: Traverse-Quelltext
28  }
```

Quelltext 3.2: Schematischer Aufbau und Zusammenhang der Member-Funktionen der Klasse *clang::RecursiveASTVisitor<Derived>*, durch den die Reihenfolge der Aufrufe dieser Member-Funktionen entsteht

Variablen:

`node` = AST-Knoten
`node_child_count` = Anzahl Kind-Knoten von *node*
`node_children(i)` = i-ter Kind-Knoten von *node*
`Derived` ≙ Klasse, die von *RecursiveASTVisitor<Derived>* abgeleitet ist (hier: `Derived` = *ClangASTConsumer*).

Platzhalter:

`#TOPLEVELKIND#` ≙ *Decl*, *Stmt* oder *Type* (je nach Typ von `node`)
`#NODEKIND#` ≙ jeweilige tatsächliche Art des Knotens `node` (z.B. *UnaryOperator*)
`#NODETYPE#` ≙ jeweiliger Typ des Knotens `node` (z.B. *clang::UnaryOperator*)
`#NODEBASEKIND#` ≙ Art des Eltern-Knotens von `node` (z.B. *Expr*)
`#NODECHILDRENKINDS(i)#` ≙ Art des i-ten Kind-Knotens von `node` (z.B. *UnaryDeref*)

In den Fall das `node` dem Zuweisungs-Operator aus dem Quelltext `*p = i + 2;` der Zeile 4 des oben bereits genauer beschrieben C++-Quelltextes 3.1 entsprechen würde, wäre `node` ein Objekt vom Typ *clang::BinaryOperator* und hätte die Clang-Art *BinAssign*. Für diesen Fall würden die Member-Funktionen aus Quelltext 3.2 in der in folgender Aufzählung vorhandenen Reihenfolge aufgerufen werden. Da die WalkUpFrom-Member-Funktionen nur die Klassenhierarchie eines einzelnen AST-Knotens durchlaufen, wurden die Aufrufe dieser aus Gründen der Übersichtlichkeit weggelassen. Außerdem sind zusätzlich zu den Signaturen die Zeilennummern der Member-Funktionen in dem Quelltext 3.2 sowie die Ausführungen der in Zeile 22 und 27 angegebenen *Visit-* und *Traverse-Quelltexte* inklusive der jeweiligen Clang-Art angegeben.

1. `RecursiveASTVisitor::TraverseStmt(clang::Stmt* node)` (Zeile 2)
2. `Derived::TraverseBinAssign(clang::BinaryOperator* node)` (Zeile 24)
3. `RecursiveASTVisitor::TraverseBinAssign(clang::BinaryOperator* node)` (Zeile 5)

4. `Derived::VisitExpr(clang::Expr* node)` (Zeile 21)
5. ***Visit**-Quelltext für **Zuweisung**s-Operator als **Expr*** (Zeile 22)
6. `Derived::VisitBinaryOperator(clang::BinaryOperator* node)` (Zeile 21)
7. ***Visit**-Quelltext für **Zuweisung**s-Operator als **BinaryOperator*** (Zeile 22)
8. `Derived::VisitBinAssign(clang::BinaryOperator* node)` (Zeile 21)
9. ***Visit**-Quelltext für **Zuweisung**s-Operator als **BinAssign*** (Zeile 22)
10. `Derived::TraverseUnaryDeref(clang::UnaryOperator* node)`
11. `RecursiveASTVisitor::TraverseUnaryDeref(clang::UnaryOperator* node)` (Zeile 5)
12. `Derived::VisitExpr(clang::Expr* node)` (Zeile 21)
13. ***Visit**-Quelltext für **Dereferenzierung**s-Operator als **Expr*** (Zeile 22)
14. `Derived::VisitUnaryOperator(clang::UnaryOperator* node)` (Zeile 21)
15. ***Visit**-Quelltext für **Dereferenzierung**s-Operator als **UnaryOperator*** (Zeile 22)
16. `Derived::VisitUnaryDeref(clang::UnaryOperator* node)` (Zeile 21)
17. ***Visit**-Quelltext für **Dereferenzierung**s-Operator als **UnaryDeref*** (Zeile 22)
18. *<ImplicitCastExpr* und *DeclRefExpr>*
19. ***Traverse**-Quelltext für den **Dereferenzierung**s-Operator als **UnaryDeref*** (Zeile 27)
20. `Derived::TraverseBinPlus(clang::BinaryOperator* node)` (Zeile 24)
21. `RecursiveASTVisitor::TraverseBinPlus(clang::BinaryOperator* node)` (Zeile 5)
22. `Derived::VisitExpr(clang::Expr* node)` (Zeile 21)
23. ***Visit**-Quelltext für **Plus**-Operator als **Expr*** (Zeile 22)
24. `Derived::VisitBinaryOperator(clang::BinaryOperator* node)` (Zeile 21)
25. ***Visit**-Quelltext für **Plus**-Operator als **BinaryOperator*** (Zeile 22)
26. `Derived::VisitBinPlus(clang::BinaryOperator* node)` (Zeile 21)
27. ***Visit**-Quelltext für **Plus**-Operator als **BinPlus*** (Zeile 22)
28. *<ImplicitCastExpr*, *IntegerIntegral* und *DeclRefExpr>*
29. ***Traverse**-Quelltext für den **Plus**-Operator als **BinaryPlus*** (Zeile 27)
30. ***Traverse**-Quelltext für den **Zuweisung**s-Operator als **BinAssign*** (Zeile 27)

Die **fett** markierten Wörter geben einen schnellen Überblick über die Ausführungsreihenfolge der Visit- und Traverse-Quelltexte und die entsprechende Clang-Art. So werden daran die unterschiedlichen Reihenfolgen der Aufrufe des Visit-Quelltextes und des Traverse-Quelltextes deutlich. Die Reihenfolge, in der der Quelltext in den Visit-Member-Funktionen aufgerufen wird, entspricht der Reihenfolge einer Pre-Order-Tiefensuche

durch den Teil-AST des AST-Knotens. Der Traverse-Quelltext innerhalb der Traverse-Member-Funktion wird hingegen in der Reihenfolge der Post-Order-Tiefensuche aufgerufen.
Zusätzlich ist zu erkennen, dass die Visit-Member-Funktionen für jede Clang-Art in der Hierarchie aufgerufen wird. Dadurch entsteht die Möglich mittels einer einzigen Visit-Member-Funktion beispielsweise für jeden BinaryOperator-AST-Knoten benachrichtigt zu werden. Im Fall der Traverse-Member-Funktionen wird diese immer nur für die in der Clang-Art-Hierarchie am weitesten unten liegenden Clang-Art aufgerufen. Die Member-Funktion *TraverseBinaryOperator* wird also nie aufgerufen.

Wie in Abschnitt 3.1.1 erläutert, bietet die Klasse *clang::RecursiveASTVisitor* sowohl die Visit-Member-Funktionen als auch die Traverse-Member-Funktionen, um während des Durchlaufens des AST für die gewünschten AST-Knoten durch einen Aufruf benachrichtigt zu werden. Da an den Join-Points in der Reihenfolge gewebt wird, in der sie in dem im Abschnitt 3.1.2 beschriebenen AspectC++-Modell registriert wurden, würde bei einer Registrierung in der Visit-Member-Funktionen und somit in der Reihenfolge der Pre-Order-Tiefensuche zuerst an den zuletzt aufgerufenen Join-Points gewebt und anschließend innerhalb dieses gewebten Quelltextes erneut gewebt werden. Um dieses Einweben in eingewebten Quelltext und den damit verbundenen Aufwand zu verhindern, wird die Registrierung der AST-Knoten innerhalb der in der Reihenfolge der Post-Order-Tiefensuche aufgerufenen Traverse-Member-Funktionen durchgeführt.

In dem Quelltext der bisherigen Variante des AspectC++-Compilers werden die die vordefinierten Operatoren betreffenden Traverse-Member-Funktionen der Klasse *clang::RecursiveASTVisitor* (siehe Abschnitt 3.1.1) nicht überschrieben. Als Herangehensweise für das Überschreiben dieser Member-Funktionen bietet es sich an, die *TraverseStmt*-Member-Funktion des *RecursiveASTVisitor*s zu überschreiben und innerhalb dieser anhand der *StmtClass*-Aufzählung [23, „clang::Stmt::StmtClass“] festzustellen, ob es sich um einen vordefinierten Operator handelt. Handelt es sich bei einem AST-Knoten um einen vordefinierten Operator, so wird dieser weiter verarbeitet.
Dazu wird in allen Traverse-Member-Funktionen für Aufrufe von vordefinierten Operatoren der jeweilige AST-Knoten an eine *Handler-Member-Funktion* übergeben, die zentral die weitere Verarbeitung übernimmt. Zuerst wird in dieser Member-Funktion geprüft, ob der semantische Gültigkeitsbereich, in dem der Aufruf des vordefinierten Operators stattfindet, sich innerhalb einer Übersetzungseinheit, in der gewebt werden soll, befindet. Dazu verwaltet der *ClangASTConsumer* als Member einen Stack, auf dem die im AST übergeordneten Gültigkeitsbereiche als Objekte des Typs *clang::Decl** verwaltet werden. Dieser Stack wird in den Traverse-Member-Funktionen für Deklarationen aktuell gehalten. Falls der Gültigkeitsbereich des Aufrufs nicht der *AC*-Namensraum oder ein anderer von dem AspectC++-Compiler generierter Bereich ist, wird der AST-Knoten zur Registrierung im AspectC++-Modell an den Modellbauer übergeben.

3.3 Änderungen des AspectC++-Modells

Bevor der Modellbauer allerdings die extrahierten AST-Knoten als Join-Points in dem AspectC++-Modell speichern kann, muss das Modell solche Join-Points aufnehmen können.

Aufrufe von (Member-)Funktionen und insbesondere Aufrufe von überladenen Operatoren werden in AspectC++-Modell durch Objekte der Klasse *TU_MethodCall* repräsentiert. Da überladene Operatoren (Member-)Funktionen sind, werden auch diese durch Objekte dieser Klasse repräsentiert. Daher fiel die Entscheidung, diese Klasse auch für Aufrufe von vordefinierten Operatoren zu verwenden. Tabelle 3.1 demonstriert die Unterschiede zwischen Aufrufen von (Member-)Funktionen und vordefinierten Operatoren.

	Aufrufe von (Member-)Funktionen (inkl. überladene Operatoren)	**Aufrufe von vordefinierten Operatoren**
Basisklasse	*clang::CallExpr*	*clang::Expr*
Funktionsdefinition	explizit	implizit
Funktionsartiger Aufruf von Operatoren	möglich	nein

Tabelle 3.1: Unterschiede zwischen Aufrufen von (Member-)Funktionen (inkl. überladene Operatoren) und Aufrufen von vordefinierten Operatoren

Um auch vordefinierte Operatoren als Objekte der Klasse *TU_MethodCall* darzustellen, müssen die Unterschiede bezüglich der Basisklasse und der Funktionsdefinition beachtet werden. Die Möglichkeit des funktionsartigen Aufrufs hat nur Einfluss auf das Einweben der Advice, welches in dem Abschnitt 3.5 beschrieben wird.
Wie in Abbildung 3.4 zu sehen ist, erbt die Klasse *TU_MethodCall* von den Klassen *ACM_Call* und *TI_MethodCall*. Die Member der Klasse *ACM_Call*, die in Abbildung 3.5 aufgelistet sind, sind unabhängig von der Art des Aufrufs. Auch an den Member-Funktionen müssen keine Änderungen vorgenommen werden.

Abbildung 3.5: Member der Klasse *ACM_Call*

Anders sieht dies bei der Klasse *TI_MethodCall*, deren Member in Abbildung 3.6 aufgelistet sind, aus.

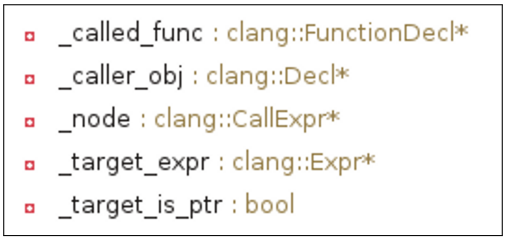

Abbildung 3.6: Member der Klasse *TI_MethodCall*

Der Member *_node* speichert einen *clang::CallExpr*-Zeiger, der auf den dem Aufruf entsprechenden AST-Knoten zeigt. Da für die vordefinierten Operatoren die Basisklasse, die in der Clang-Typ-Hierarchie am weitesten unten steht, *clang::Expr* lautet und diese wiederum die Basisklasse der Klasse *clang::CallExpr* ist, kann *_node* nicht auf einen AST-Knoten eines vordefinierten Operators zeigen. Um dies zu ermöglichen muss der Typ von *_node* auf *clang::Expr** geändert werden.
Der Member *_called_func* ist ebenfalls problematisch, da er auf die (Member-)Funktions-Definition des Aufrufs zeigt. Weil für einen vordefinierten Operator keine explizite Funktionsdefinition vorhanden ist, auf die der Member zeigen könnte, wird dieser im Fall eines vordefinierten Operators auf den Wert *0* gesetzt.

Die Member-Funktionen dieser Klasse müssen so angepasst werden, dass sie je nach tatsächlichen Typs des Members *_node* die entsprechenden Member-Funktionen auf dem Objekt des AST-Knoten aufrufen. Bis auf diese Unterscheidungen bleiben die Funktionsweisen der Member-Funktionen allerdings identisch.

3.4 Registrierung der Join-Points im AspectC++-Modell

Nachdem das AspectC++-Modell Join-Points an Aufrufen von vordefinierten Operatoren aufnehmen kann, müssen diese Aufrufe korrekt im Modell registriert werden.

Im bisherigen AspectC++ existieren dazu Member-Funktionen, die Join-Points an Aufrufen von (Member-)Funktionen und auch die (Member-)Funktionen an sich im AspectC++-Modell registrieren. Jedoch erwarten diese Registrier-Member-Funktionen unter anderem eine (Member-)Funktions-Deklaration (als *clang::FunctionDecl*-Zeiger) und einen Aufruf-Ausdruck (als *clang::CallExpr*-Zeiger). Da die AST-Knoten der vordefinierten Operatoren weder durch eine *clang::CallExpr* dargestellt werden noch einen Verweis auf eine explizite Funktionsdefinition besitzen, müssten sowohl die Signatur als auch die Funktionalität der bestehenden Registrier-Member-Funktionen angepasst werden. Diese Anpassungen würden jedoch den Aufwand zur Registrierung bisheriger Join-Points aufgrund der Unterscheidungen zwischen vordefinierten Operatoren und (Member-)Funktionen vergrößern.
Um diesen Overhead zu vermeiden, wird eine eigene Registrier-Member-Funktion für vordefinierte Operatoren implementiert, die sich an den bestehenden Registrier-Member-Funktionen *ClangModelBuilder::register_function* und *ClangModelBuilder::register_call* orientiert und die Besonderheiten und Spezialfälle der vordefinierten Operatoren berück-

sichtigt. Diese Registrier-Member-Funktion wird in die Klasse *ClangModelBuilder* eingefügt und im Folgenden bezüglich des Aufbaus und der Funktionsweise beschrieben.

Der erste Parameter der Registrier-Member-Funktion ist ein Zeiger auf den AST-Knoten, der im AspectC++-Modell registriert werden soll. Da der Zeiger-Typ dieses Parameters *clang::Expr* ist und nicht alle AST-Knoten, die durch ein Objekt dieser Klasse dargestellt werden, Aufrufe von vordefinierten Operatoren sind, wird zunächst überprüft, ob es sich um einen vordefinierten Operator handelt.
Anschließend wird der zweite Parameter validiert, der auf ein *clang::DeclaratorDecl*-Objekt zeigt. Dieses Objekt stellt die lexikalische Definition des Eltern-AST-Knotens, also die Definition, in der der Aufruf des vordefinierten Operators stattfindet, dar. Falls es sich bei dieser Definition um eine (Member-)Funktionsdefinition handelt, darf diese zum einen keine Template-Argumente besitzen und sich zum anderen nicht innerhalb einer Definition[3], die Template-Argumente besitzt, befinden. Diese Einschränkung kommt zustande, da AspectC++ das Weben in diese Situationen aktuell nicht unterstützt. Des Weiteren darf diese Definition selbst nicht von dem AspectC++-Compiler generiert worden sein und sich auch nicht innerhalb einer Definition[4] befinden, die von dem AspectC++-Compiler generiert worden ist. Ist mindestens eine dieser beiden Bedingungen nicht erfüllt, so wird der AST-Knoten und somit der Join-Point an dem Aufruf des vordefinierten Operators nicht weiter betrachtet. Ansonsten wird die Definition des Eltern-AST-Knotens im AspectC++-Modell registriert.
Um die Operator-Signatur zu generieren, werden neben dem Operatornamen, der eine Eigenschaft des AST-Knotens ist, zusätzlich die Typen der Argumente benötigt. Die Signatur wird dann durch die Konkatenation des Operatornamens, der öffnenden runden Klammer, einer durch Kommata getrennte Liste der Argumenttypen als Zeichenkette und der schließenden runden Klammer gebildet.
So ergibt sich für den vordefinierten binären Plus-Operator, der für zwei Werte des Typs *int* aufgerufen wird, die Signatur „operator +(int,int)“. Falls es sich bei einem Argumenttyp um einen der Typen *long long*, *long* oder *short* handelt, wird an dessen textuelle Repräsentation das Postfix „ int“ angehängt. Dies sorgt dafür, dass unterschiedliche Schreibweisen des gleichen Typs (siehe [6, S. 149]) zur gleichen Signatur führen.
Anhand der generierten Signatur kann nun überprüft werden, ob sich die implizite Operator-Funktions-Definition bereits im AspectC++-Modell befindet. Ist dies nicht der Fall, so wird die implizite Funktions-Definition des vordefinierten Operators als Element des AspectC++-Modells erstellt und in dieses eingefügt. Dazu wird ein neues Objekt der Klasse *TU_Function*, die ein Funktions-Definitions-Element im AspectC++-Model darstellt, instanziiert und dessen Eigenschaften wie folgt gesetzt:

- Name des Operators entsprechend Kapitel 13.6 des C++-Standards[6]
- Typen der Argumente und Rückgabetyp
- Kein Verweis auf den AST-Knoten der impliziten Funktions-Definition des vordefinierten Operators, da dieser nicht vorhanden ist.

[3](Member-)Funktionsdefinition oder Klassendefinition
[4](Member-)Funktionsdefinition, Klassendefinition oder Namensraum-Definition

- Keine variable Argumentanzahl, da alle vordefinierten Operatoren, wie in dem Kapitel 13.6 des C++-Standards[6] zu sehen ist, eine feste Argumentanzahl haben.
- Keine *const*- bzw. *volatile*-Qualifizierer, da ein vordefinierter Operator keine Memberfunktion ist.

Da ein vordefinierter Operator einer globalen Funktion entspricht, wird das erstellte Funktions-Definitions-Element im AspectC++-Modell als Kind des Elements, das den globalen Namensraum darstellt, eingefügt.
Nachdem die Funktions-Definition des vordefinierten Operators im AspectC++-Modell vorhanden ist, muss der Join-Point an dem Aufruf des vordefinierten Operators im Modell registriert werden. Dies geschieht, indem eine Repräsentation des Aufrufs im AspectC++-Modell als ein Objekt der Klasse *TU_MethodCall* erstellt wird und anschließend folgende Eigenschaften erhält:

- Verweis auf das bereits im AspectC++-Modell registrierte Funktions-Definitions-Element als Ziel des Aufrufs
- ID, um mehrere Aufrufe des selben vordefinierten Operators zu unterscheiden
- Kein Verweis auf den AST-Knoten der Definition des aufgerufenen Operators, da dieser nicht vorhanden ist.
- Verweis auf den AST-Knoten des Aufrufs des vordefinierten Operators (als *clang::Expr**)
- Verweis auf den AST-Knoten der Definition der (Member-)Funktion oder Variable, in deren Bereich der Aufruf des vordefinierten Operators stattfindet
- Quelltext-Position des Aufrufs des vordefinierten Operators als *clang::SourceLocation*

Nach dem Setzen der Eigenschaften wird das neu erstellte Objekt als Kind des AspectC++-Modell-Elements, das die Funktions-Definition des vordefinierten Operators darstellt, eingefügt.
Damit ist die Registrierung des Join-Points an dem Aufruf des vordefinierten Operators im AspectC++-Modell abgeschlossen.

3.5 Änderungen am Quelltext-Weber

Der Quelltext-Weber des AspectC++-Compiler webt Advice-Quelltext an den gewünschten Join-Points in den C++-Quelltext ein. Diese Paare aus Join-Point und Advice wurden durch den hier nicht näher beschriebenen Planer durch den Abgleich des Pointcut-Ausdruckes des Advice mit dem Join-Point bestimmt. Um nun den Advice-Quelltext an einer bestimmten Stelle im Quelltext bzw. an einem Join-Point einzuweben, müssen im Grundlegendem

1. die Struktur oder engl. *struct*, aus dem das Join-Point-Objekt des Join-Point-Advice-Paares instanziiert wird, in den Quelltext eingefügt werden,

2. der *Aufruf-Wrapper*, der den Aufruf des vordefinierten Operators kapselt, in Abhängigkeit des Advice-Quelltextes und der Art des vordefinierten Operators generiert und oberhalb des Aufrufs des vordefinierten Operators im Quelltext eingefügt werden und

3. der Aufruf des vordefinierten Operators im Quelltext durch den Aufruf des Aufruf-Wrappers ersetzt werden. Die Argumente des vordefinierten Operators und eventuell ein Zeiger auf das Objekt, in dem der Aufruf stattfindet, werden durch den Aufruf des Aufruf-Wrappers an diesen übergeben.

Diese grundlegenden Verarbeitungsschritte spiegeln sich im Quelltext 3.3 wieder, in dem das Ergebnis des Webens von Advice-Quelltext an einem Aufruf des globalen überladenen Plus-Operators für den Typ *ExampleClass* zu sehen ist.

```
1  // Struktur, aus dem das Join-Point-Objekt instanziiert wird:
2  template <typename TResult, typename TThat, typename TTarget, ↩
       typename TArgs>
3  struct TJP_0 {
4    // Member-Funktion, die die Signatur des Join-Points zurückgibt:
5    inline static const char *signature () {
6      return "ExampleClass operator +(ExampleClass,ExampleClass)";
7    }
8  };
9
10 // Aufruf-Wrapper:
11 template <typename TArg0, typename TArg1, typename TResult>
12 __attribute__((always_inline)) inline
13 TResult __call_0 (TArg0 arg0, TArg1 arg1, AC::RT<TResult>) {
14   // Join-Point-Typ mit den konkreten Typen, die an diesem
15   // Join-Point vorliegen, als neuen Typ definieren:
16   typedef TJP_0<TResult, ::ExampleClass, void, AC::TL<::↩
         ExampleClass, AC::TL<::ExampleClass, AC::TLE>>> __TJP;
17   // Puffer für den Rückgabewert erstellen:
18   AC::ResultBuffer< TResult  > __result_buffer;
19   // Join-Point-Objekt erstellen:
20   __TJP tjp;
21   // Advice-Quelltext ausführen:
22   AC::invoke_ExampleAspect_ExampleAspect__a0_before<__TJP> (&tjp);
23   // Operator aufrufen:
24   ::new (&__result_buffer) TResult  (::operator +(arg0, arg1));
25   // Rückgabewert zurückgegeben:
26   return (TResult &)__result_buffer;
27 }
28
```

```
29 // Beispiel-Quelltext mit eingewebtem Aufruf des Aufruf-Wrappers:
30 int main() {
31   // Lokale Variablen definieren und initialisieren:
32   ExampleClass a = ExampleClass(1), b = ExampleClass(2);
33   int d = 1, e = 2;
34   // Aufruf des überladenen binären Plus-Operators:
35   ExampleClass c = __call_0< ExampleClass &, ExampleClass & >(a, b,↩
         __AC_TYPEOF((a + b)));
36   // Aufruf des vordefinierten binären Plus-Operators:
37   int f = d + e;
38 }
```

Quelltext 3.3: Der von dem AspectC++-Compilter generierte Quelltext für den Beispiel-Quelltext und -Aspekt in A.1. (Zur besseren Lesbarkeit wurde der Name des Aufruf-Wrappers gekürzt und in der Join-Point-Struktur wurden *typedef*s, *enum*s und *struct*s für Argument-, Rückgabe- und weitere Typen entfernt, da diese in diesem Beispiel nicht verwendet werden.)

In den Zeilen 1 bis 8 ist das Ergebnis des ersten Verarbeitungsschrittes zu sehen. Da die Signatur bereits bei der Registrierung des Join-Points im AspectC++-Modell gebildet wurde und alle anderen Elemente der Struktur unabhängig davon sind, ob es sich um einen vordefinierten Operator handelt, sind für diesen Verarbeitungsschritt keine Änderungen an dem AspectC++-Compiler nötig.

Der Aufruf-Wrapper, der sich über die Zeilen 10 bis 27 erstreckt, wird in dem zweiten Verarbeitungsschritt generiert. Hier ist die Zeile 24 interessant, denn hier wird der Aufruf des überladenen Operators, der im Quelltext durch den Aufruf-Wrapper ersetzt worden ist, durchgeführt.

Wie zu sehen ist, werden im bisherigen AspectC++-Compiler auch überladenene Operatoren, die mit der Operatorschreibweise (zum Beispiel „a + b“) aufgerufen worden sind, innerhalb des Aufruf-Wrappers mit der Funktionsschreibweise[6, Kap. 13.5:4] (zum Beispiel „operator +(a, b)“) aufgerufen. Da die Funktionsschreibweise bei vordefinierten Operatoren durch den C++-Standard im Kapitel 13.6:1 nur für die Überladungs-Auflösung erlaubt und für andere Zwecke verboten ist, wäre der bisherige Aufruf der überladenenen Operatoren für die vordefinierten Operatoren nicht gültig. Um in dieser Zeile also gültigen C++-Quelltext zu generieren, muss die dafür zuständige Member-Funktion *CodeWeaver::make_proceed_code* des AspectC++-Compilers angepasst werden.

Zur Bildung der Aufruf-Zeichenkette, die den Aufruf des vordefiniertenals C++-Quelltext darstellt, wurde im bisherigen AspectC++-Compiler vereinfacht[5] zuerst der (Member-)Funktionsname bzw. Operatorname des Aufrufs inklusive öffnender runder Klammer generiert, anschließend wurden mit einer *for*-Schleife die Argumente sowie trennende Kommata angehängt und zum Schluss wurde die schließende runde Klammer ergänzt. Die Aufruf-Zeichenkette „beispielfunktion(arg0, arg1, arg2)“ stellt eine beispielhaftes

[5]Template-Argumente und Gültigkeitsbereichs-Qualifizierer sind für den Vergleich mit vordefinierten Operatoren nicht relevant (da vordefinierte Operatoren keine Template-Argumente besitzen und ihre Gültigkeit global ist) und werden daher hier nicht berücksichtigt.

Ergebnis des beschriebenen Vorgangs dar.
Für einen Aufruf eines vordefinierten Operators sieht die Bildung der Aufruf-Zeichenkette allerdings anders aus. So muss beispielsweise im Fall des binären vordefinierten Plus-Operators die Zeichenkette aus dem ersten Argument, gefolgt von dem Plus und dem zweiten Argument generiert werden, sodass sich „arg0 + arg1“ ergibt. Allgemein ergibt sich für binäre vordefinierte Operatoren das Schema `((arg0) Operator (arg1))`. Ein Ausnahme bildet der Array-Index-Operator `[]`, dessen Aufruf dem Schema `((arg0)[arg1])` folgt.
Das Schema für die Aufrufe der unären Operatoren ergibt sich als `(Operator (arg0))`. Der Aufruf des einzigen ternären *Conditional*-Operators folgt dem Schema `(arg0 ? arg1 : arg2)`.
Die runden Klammern um den Gesamtausdruck stellen sicher, dass zuerst der vordefinierte Operator innerhalb der Zeichenkette und erst danach umgebene Operatoren aufgerufen werden. Ohne diese Klammern ergeben sich in einigen Fällen durch die Auswertungs-Rangfolge der vordefinierten Operatoren falsche Auswertungsreihenfolge. So würde beispielsweise ohne die Klammern die Anwendung des Adress-Operators auf die Aufruf-Zeichenkette, die einen Aufruf des binären Plus-Operators enthält, dazu führen, dass der Adress-Operator nur auf das erste Argument des Plus-Operators und nicht auf den gesamten Aufruf angewendet wird.
Die runden Klammern um die einzelnen Argumente sind in Fällen nötig, in denen die Argumente keine Variablen sondern Ausdrücke mit vordefinierten Operatoren sind. So würde beispielsweise die Anwendung des Multiplikations-Operators auf die Argumente `a + b` und `c + d` ohne diese Klammern zu dem Gesamtausdruck `(a + b * c + d)` führen. Durch die Auswertungsrangfolge der vordefinierten Operatoren würde der Ausdruck fälschlicherweise als `(a + (b * c) + d)` und nicht als `((a + b) * (c + d))` interpretiert.
Wie zu erkennen ist, unterscheidet sich die Generierung des Aufrufs je nach Art des Aufrufs. Die Gemeinsamkeit ist allerdings, dass sich der Aufruf durch die Argumente zusammensetzt, die voneinander durch Trennzeichen, wie zum Beispiel dem Komma, getrennt werden. Daher wird ein Array (als *std::vector<std::string>*) verwendet, das mit den je nach Art des Aufrufs unterschiedlichen Trennzeichen gefüllt wird. Anschließend wird dieses Array zentral durchlaufen und abwechselnd ein Trennzeichen aus dem Array und ein Argument an die Aufruf-Zeichenkette angehängt. Dieses Vorgehen sorgt dafür, dass die Aufruf-Zeichenkette unabhängig von der Art des Aufrufs an einer zentralen Stelle zusammengesetzt wird. Somit wird die Redundanz im Quelltext reduziert und die Lesbar- und Wartbarkeit erhöht.

Im dritten Verarbeitungsschritt wird der Aufruf des überladenen Operators durch den Aufruf des Aufruf-Wrappers ersetzt. Da ein Aufruf eines überladenenen Operators in Operatorschreibweise sich syntaktisch nicht von einem Aufruf eines vordefinierten Operators unterscheidet und somit die Funktionalität bereits vorhanden ist, sind in diesem Verarbeitungsschritt keine Änderungen notwendig.

4 Implementierung

In diesem Kapitel werden Situationen und Konstellationen genannt, an denen die Implementierung nicht entsprechend dem in Kapitel 3 erläuterten Entwurf durchgeführt werden kann. Im Gegensatz dazu werden die Implementierungsschritte, die entsprechend dem Entwurf umgesetzt werden, hier nicht erwähnt.

Konstellationen, in denen von dem Entwurf abgewichen werden muss, sind zuerst durch die Probleme während der Extrahierung der AST-Knoten entstanden. Die hierdurch nötige abweichende Generierung der Traverse-Member-Funktionen wird in dem ersten Abschnitt dieses Kapitels erläutert.
Außerdem werden durch die fehlende Basisklasse der vordefinierten Operatoren ein Helfer-Typ und Helfer-Funktionen erstellt, um ein einheitliches Interface für den Zugriff auf die Eigenschaften eines vordefinierten Operators zu generieren.
Weiterhin entstehen Situationen, in denen für Sonderfälle einiger vordefinierter Operatoren die Implementierung erweitert werden muss oder in denen eine Implementierung nur mit Einschränkungen möglich ist. Die Implementierungen solcher Sonderfälle werden in dem Abschnitt 4.3 genauer untersucht.

4.1 Generierung der Traverse-Member-Funktionen

Im Entwurf war geplant, die TraverseStmt-Member-Funktion in der Klasse zu überschreiben und anschließend anhand der StmtClass-Aufzählung herauszufinden, ob es sich um einen vordefinierten Operator handelt. Das Problem dabei besteht darin, dass der Parameter der TraverseStmt-Member-Funktion in einigen Fällen nicht auf den entsprechenden AST-Knoten zeigt, sondern ein Null-Zeiger ist. Dies tritt beispielsweise im Fall des binären Plus-Operators auf. Da die Clang-Klasse *RecursiveASTVisitor* größtenteils durch Makro-Ersetzung [6, Kap. 16.3] generiert wird, ist zu erwarten, dass das Debuggen des soeben genannten Problems sehr zeitaufwendig ist. Außerdem ist es fraglich, ob die eventuell gefundene Ursache des Problems in annehmbarer Zeit beseitigt werden kann.
Aus diesen Gründen wird die Möglichkeit, in der Clang-Art-Hierarchie tiefer zu gehen und die Member-Funktionen *TraverseUnaryOperator* und *TraverseBinaryOperator* zu überschreiben, betrachtet. Doch diese Traverse-Member-Funktionen werden, wie im Kapitel 3.2 beschrieben, nicht aufgerufen.
Daher bleibt nur die Möglichkeit, in der Clang-Art-Hierarchie auf die unterste Ebene zu gehen und für jede Clang-Art der vordefinierten Operatoren die individuelle Traverse-Member-Funktion zu überschreiben.
Während der Analyse dieser individuellen Traverse-Member-Funktionen fällt auf, dass

sich deren Signaturen aus dem Schema
`bool Traverse<Clang-Operator-Art>(clang::<Typ des AST-Knotens>*)`
zusammensetzen lässt. Die für die Platzhalter *<Clang-Operator-Art* und *<Typ des AST-Knotens* benötigten Werte sind in der Tabelle A.1 aufgelistet.
Das funktionsartiges Makro [6, Kap. 16.3:10], das aus den soeben genannten Angaben der Clang-Operator-Art und dem Typ des AST-Knotens eine Traverse-Member-Funktion für den entsprechenden vordefinierten Operator erzeugt, ist in dem Quelltext 4.1 zu sehen.

```
1 #define CREATE_BUILT_IN_OPER_TRAVERSE(OPER_KIND, NODE_TYPE)          \
2 bool Traverse##OPER_KIND(clang::NODE_TYPE* built_in_oper) {         \
3   bool base_class_result = RecursiveASTVisitor<ClangASTConsumer>::↩
      Traverse##OPER_KIND(built_in_oper);                             \
4   handle_built_in_operator_traverse(built_in_oper);                 \
5   return base_class_result;                                         \
6 }
```

Quelltext 4.1: Funktionsartiges Makro zur Generierung von Traverse-Member-Funktionen. Der erste Parameter stellt die Clang-Operator-Art und der Zweite den Clang-Typ des entsprechenden AST-Knotens dar. In Tabelle A.1 sind diese Parameterwerte in Abhängigkeit des Operators zu finden.

In Zeile 2 ist das oben beschriebene Schema der Signatur der Traverse-Member-Funktion ersichtlich. Zur Generierung des Rumpfes wird analog zur Signatur nur die Clang-Operator-Art (`OPER_KIND`) und der Typ des AST-Knotens (`NODE_TYPE`) benötigt. In diesem aus den Zeilen 3 bis 5 entstehenden Rumpf der Traverse-Member-Funktion wird die entsprechende Traverse-Member-Funktion der Basisklasse *RecursiveASTVisitor* aufgerufen und das Ergebnis dieses Aufrufs gespeichert. Dies ist notwendig, da ohne den Aufruf der Member-Funktion in der Basisklasse der AST, wie in dem Kapitel 3.1.1 erklärt, nicht weiter durchlaufen würde. Danach wird die Member-Funktion *handle_built_in_operator_traverse* aufgerufen, die die weitere Verarbeitung der AST-Knoten übernimmt. Durch die Rückgabe des zwischengespeicherten Ergebnisses aus Zeile 3 wird sichergestellt, dass der Durchlauf durch den Clang-AST nicht verändert wird.

Um die Redundanz weiter zu reduzieren, werden für die Gruppen von vordefinierten Operatoren, deren AST-Knoten vom gleichen Typ sind, weitere spezielle funktionsartige Makros definiert. Diese benötigen zur Generierung der Traverse-Member-Funktion eines vordefinierten Operators aus der jeweiligen Gruppe nur noch die Angabe des Postfix' der Clang-Operator-Art, da alle anderen Angaben für alle vordefinierten Operatoren in einer solchen Gruppe identisch und somit bekannt sind. Intern rufen diese funktionsartigen Makros dann, wie in dem Quelltext 4.2 für unäre vordefinierte Operatoren zu sehen, das soeben beschriebene funktionsartige Makro im Quelltext 4.1 auf.

```
1 #define CREATE_UNARY_BUILT_IN_OPER_TRAVERSE(OPER_POSTFIX)   \
2 CREATE_BUILT_IN_OPER_TRAVERSE(Unary##OPER_POSTFIX, UnaryOperator)
```

Quelltext 4.2: Funktionsartiges Makro, das aus der Angabe des Postfix' der Clang-Operator-Art die entsprechende Traverse-Member-Funktion des unären vordefinierten Operators generiert. Intern wird dazu das in Quelltext 4.1 definierte funktionsartige Makro aufgerufen.

Anschließend werden die Aufrufe dieser funktionsartigen Makros für alle vordefinierten Operatoren mit den entsprechenden in der Tabelle A.1 aufgelisteten Werten in den Quelltext des AspectC++-Compilers eingefügt. Während des Kompiliervorganges generiert der Präprozessor aus diesen dann die benötigten Traverse-Member-Funktionen.

4.2 Helfer-Funktionen und -Typen

In dem Quelltext des AspectC++-Compilers besteht bisher kein zentraler Ort, an dem Frontend-spezifische Helfer-Funktionen und -Typen platziert werden können[1]. Da sich die Verarbeitung von vordefinierten Operatoren über mehrere Klassen des AspectC++-Compilers erstreckt und die benötigten Helfer-Funktionen und -Typen sich nicht einer Klasse zuordnen lassen, wird ein neuer Namensraum mit dem Namen *Helpers* in den Dateien *Helpers.h* und *Helpers.cc* definiert. Dieser neue Namensraum beinhaltet die in diesem Abschnitt beschriebenen Typen und Funktionen und ist auch für zukünftige Frontend-spezifsche Helfer-Funktionen und -Typen gedacht.

Da in Clang für die AST-Knoten der Aufrufe von vordefinierten Operatoren nur die sehr allgemeine Basisklasse *clang::Expr* existiert, wird eine Möglichkeit benötigt, um beispielsweise zu ermitteln, ob es sich bei dem tatsächlichen Typ[2] des AST-Knotens um einen Typ eines vordefinierten Operators handelt. Im Worstcase muss dazu bisher der AST-Knoten mittels der RTTI (**R**un **T**ime **T**ype **I**nformation) des Clang-Frontends[24] mit jedem möglichen Typ eines vordefinierten Operators verglichen werden.
Um diese Vergleiche nicht mehrmals an verschiedenen Stellen im Quelltext des AspectC++-Compilers durchzuführen, wird der Aufzählungs-Typ oder engl. *enumeration type Helpers::enum_expression_type* erstellt, der mit Hilfe von sechs Bits den tatsächlichen Typ des AST-Knoten speichert und in dem Quelltext 4.3 zu sehen ist.

```
1 enum enum_expression_type {
2   ET_BUILTIN_OPERATOR = 16,                    // ^= 0b...0010000
3   // ^= Bitmaske für alle vordefinierten Operatoren
4   ET_BUILTIN_UNARY_OPERATOR = 17,              // ^= 0b...0010001
5   ET_BUILTIN_BINARY_OPERATOR = 18,             // ^= 0b...0010010
6   ET_BUILTIN_ARRAYSUBSCRIPT_OPERATOR = 19,     // ^= 0b...0010011
7   // ^= clang::ArraySubscriptExpr
8   ET_BUILTIN_CONDITIONAL_OPERATOR = 20,        // ^= 0b...0010100
9   // ^= clang::ConditionalOperator
```

[1] Für Helfer-Funktionen, die den Frontend-unabhängigen Teil des AspectC++-Modells betreffen, existieren die Dateien *Utils.h* und *Utils.cc*.
[2] Der tatsächliche Typ ist die Klasse aus der das AST-Knoten-Objekt instanziiert wurde.

```
10
11    ET_BUILTIN_IMPLICIT_OPERATOR = 24,              // ^= 0b...0011000
12    // ^= Bitmaske für alle impliziten vordefinierten Operatoren
13    ET_BUILTIN_IMPLICIT_DEREF_OPERATOR_INSIDE_ARROW = 25,
14                                                    // ^= 0b...0011001
15    ET_BUILTIN_IMPLICIT_FUNCTION_POINTER_DEREF_WHILE_CALL = 26,
16                                                    // ^= 0b...0011010
17
18    ET_NON_BUILTIN_CALL = 32,                       // ^= 0b...0100000
19
20    ET_UNKNOWN = 1                                  // ^= 0b...0000001
21  };
```

Quelltext 4.3: Definition des Aufzählungs-Typs *Helpers::enum_expression_type*, der zur Speicherung des tatsächlichen Typs eines AST-Knotens mit der Basisklasse *clang::Expr* verwendet wird. Die zwei Bitmasken bieten die Möglichkeit einer groben Unterscheidung des Typs.

Für einen AST-Knoten wird einmalig mittels RTTI der tatsächliche Typ des AST-Knotens ermittelt und als Objekt der soeben beschriebenen Aufzählung gespeichert. In darauffolgenden Verarbeitungsschritten wird anschließend zur Unterscheidung des Typs des AST-Knoten der dieses Aufzählungs-Objekt verwendet. Außerdem wird dieses Aufzählungs-Objekt in einem neuen Member innerhalb Klasse *TI_MethodCall* zusammen mit dem AST-Knoten im AspectC++-Modell gespeichert, sodass die Information über den tatsächlichen Typ immer zusammen mit dem AST-Knoten verfügbar ist. Dieser Member ist zusammen mit den anderen Membern der Klasse *TI_MethodCall* in Abbildung 4.1 dargestellt.

Abbildung 4.1: Member der Klasse *TI_MethodCall* nach Anpassung für vordefinierte Operatoren

Aufgrund der gewählten Definition der Aufzählungswerte kann durch eine Konjunktion und einen Vergleich mit einer der in den Zeilen 2 und 11 zusehenden Bitmasken sehr effizient festgestellt werden, ob es sich bei dem zugehörigen AST-Knoten um einen vordefinierten Operator oder um eine implizite Dereferenzierung handelt. Zusätzlich kann dieser Aufzählungs-Typ leicht um weitere Typen, wie zum Beispiel die Typen der überladenen Operatoren und (Member-)Funktionen, erweitert werden.
Auch ob ein AST-Knoten ein bestimmter vordefinierter Operator ist, kann durch einen Vergleich des gewünschten Aufzählungswert mit der zu dem AST-Knoten gehörenden

Aufzählung bestimmt werden.
Sobald der tatsächliche Typ des AST-Knotens bekannt ist, kann der AST-Knoten mittels einem statischen Cast ohne Verwendung der RTTI in ein Objekt seines tatsächlichen Typs überführt werden.

Eine weitere Problematik entsteht durch die vielen Unterscheidungen anhand des Typs des AST-Knotens. Diese sind notwendig, da durch die fehlende Basisklasse keine einheitliche Syntax für den Zugriff auf die Eigenschaften eines vordefinierten Operators existiert. Um beispielsweise die den Argumenten des vordefinierten Operators entsprechenden AST-Knoten zu erhalten muss im Fall des Typs *clang::UnaryOperator* die Member-Funktion `getSubExpr()` aufgerufen werden. In dem Fall des Typs *clang::BinaryOperator* müssen hingegen die Member-Funktionen `getLHS()` und `getRHS()` und im Fall des Typs *ConditionalOperator* die Member-Funktionen `getCond()`, `getTrueExpr()` und `getFalseExpr()` aufgerufen werden.
Unterscheidungen wie diese müssen an vielen Stellen im Quelltext des AspectC++-Compilers durchgeführt werden und schränken dadurch die Les- und Wartbarkeit ein. Aus diesem Grund und um Redundanz zu vermeiden, werden Helfer-Funktionen implementiert, die die genannten Unterscheidungen und die Behandlung von Sonderfällen kapseln und somit ein einheitliches Interface bieten, das unabhängig von dem tatsächlichen Typ des vordefinierten Operators angesprochen werden kann.

4.3 Sonderfälle und Einschränkungen

Vordefinierte Operatoren unterscheiden sich an vielen Stellen von (Member-)Funktionen. So sind diese Operatoren oftmals, im Gegesatz zu anderen Aufrufen, in konstanten Programmkonstrukten erlaubt oder arbeiten mit Typen, die nur implizit existieren. Außerdem ist die Auswertungsreihenfolge der Operator-Argumente in einigen Fällen im Vergleich zu Argumenten eines (Member-)Funktions-Aufrufs durch den C++-Standard fest vorgegeben.
Alle diese Sonderfälle und die eventuell daraus resultierenden Einschränkungen werden in den folgenden Abschnitten analysiert und so weit wie möglich implementiert.

4.3.1 Konstante Ausdrücke

Ein konstanter Ausdruck[6, Kap. 5.19] kann zur Kompilierzeit ausgewertet werden und ist somit zur Laufzeit konstant. Im Gegensatz zu Aufrufen von (Member-)Funktionen sind Aufrufe von vordefinierten Operatoren in konstanten Ausdrücken erlaubt. Zur Demonstrierung solcher Aufrufe dient der Quelltext 4.4.

```
1 class ExampleClass {
2   static const int const_member = 5 * 2; // Wert des statischen
3                                          // und konstanten Members
4   unsigned int bitfield : 4 / 2;         // Größe des Bitfelds
5 };
```

```
6  const int const_two = 3 - 1;                  // Wert der konstanten
7                                                // und globalen Variable
8  static char char_array[const_two + 5];        // Größe des statischen
9                                                // Arrays
10 enum ExampleEnum {
11   ENUM_VALUE = const_two + 1                  // Wert des Aufzählungs-
12 };                                            // elements
13 switch(const int const_temp = 1) {
14   case const_temp + 1: {                      // Vergleichswert für die
15     // ....                                    // case-Bedingung
16     break;
17   }
18 }
```

Quelltext 4.4: Beispiele für vordefinierte Operatoren in zur Kompilierzeit konstanten Ausdrücken
Im Kommentar befindet sich der jeweilige Verwendungszweck des Rückgabewertes des vordefinierten Operators.

Falls an einem sich an einem solchen Aufruf befindenden Join-Point durch den AspectC++-Compiler Advice-Quelltext eingewebt werden würde, würde im Zuge dessen in dem C++-Quelltext der Aufruf des vordefinierten Operators durch den Aufruf des Aufruf-Wrappers ersetzt. Dies würde wiederum dazu führen, dass innerhalb des konstanten Ausdruckes eine Funktion aufgerufen wird. Dies würde die Konstanz des konstanten Ausdruckes zerstören und ist in der Programmiersprache C++ nicht erlaubt[6, Kap. 5.19:2-4]. Weil an solchen Join-Points somit nicht gewebt werden kann, werden diese nicht in das AspectC++-Modell aufgenommen.
Je nach Optimierungsstufe des C++-Compilers ist diese Einschränkung nicht problematisch, da solche konstanten Ausdrücke während des Kompiliervorganges auf einen konstanten Wert zusammengefaltet[3] werden und diese Join-Points somit in diesem Fall zur Laufzeit nicht vorhanden sind.
Außerdem kann - sobald in zukünftigen Versionen des AspectC++-Compilers die Version *C++1y* der Programmiersprache C++ unterstützt wird - durch die Verwendung des *constexpr*-Spezifizierers[6, Kap. 7.1.5] ein Weben an Join-Points in konstanten Ausdrücken ermöglicht werden. Dazu muss der Advice-Quelltext allerdings als *constexpr* dargestellt werden können. Ob dies möglich ist, muss analysiert werden und hängt sowohl vom Advice-Quelltext als auch von den verwendeten Pointcut-Funktionen ab. Genauere Untersuchungen diesbezüglich werden in dieser Arbeit auf Grund der oben genannten geringen Relevanz und des beschränkten Zeitrahmens unterlassen.

[3] Die konstanten Ausdrücke werden von dem Compiler ausgewertet und durch das Ergebnis der Auswertung ersetzt.

4.3.2 Bit-Felder

Ein Bit-Feld[6, Kap. 9.6] ist ein nicht-statischer Member, der einen integralen[4], Aufzählungs- oder booleschen Typ hat und dessen Größe sich „bit-genau“ festlegen lässt. Ein Bit-Feld der Länge zwei wird in Zeile 4 des Beispiel-Quelltexts 4.4 deklariert.
Die genaue Speicher-Allokation und die Byte-Reihenfolge innerhalb eines Bit-Felds sind laut C++-Standard Kapitel 9.6 Absatz 1 abhängig von der Implementierung des Compilers und von dem System, auf dem der Quelltext kompiliert wird. Infolgedessen verbietet der C++-Standard im Absatz 3 des soeben genannten Kapitels das Anwenden des Adress-Operators `&` auf ein Bit-Feld, sodass es kein Zeiger auf ein Bit-Feld gibt. Auch dürfen keine nicht-konstanten Referenzen an ein Bit-Feld gebunden werden[5].
Dies ist problematisch, falls ein Bit-Field als Argument per konstanter Referenz an einen vordefinierten Operator[6] übergeben **und** für diesen Operator ein Advice eingewebt wird. In diesem Fall würde der bisherige AspectC++-Compiler Quelltext generieren, der die Adresse eines per Referenz übergebenen Parameters dazu verwendet, um dessen Wert über das Join-Point-Objekt im Advice-Quelltext zugänglich und änderbar zu machen. Verbotenerweise würde demzufolge zum einen eine nicht-konstante Referenz an ein Bit-Feld gebunden und zum anderen der Adress-Operator für ein Bit-Feld angewendet werden. Analog besteht das Problem bei vordefinierten Operatoren, die ein Bit-Feld per konstanter Referenz zurückgeben.
Um dieses Einweben von ungültigem C++-Quelltext zu verhindern, werden Join-Points, an denen die Referenz eines Bit-Felds für einen vordefinierten Operator verwendet wird[6], nicht in das AspectC++-Modell aufgenommen und somit nicht für das Einweben verwendet.
Theoretisch könnten die betroffenen Argumente und/oder Rückgabewerte im Advice-Quelltext nur lesend verfügbar gemacht werden, indem die Werte der Bit-Felder kopiert würden. Da Bit-Felder immer einen skalaren Typ haben[6, Kap. 9.6 u. 3.9:9] und somit einfach zu kopieren sind[6, Kap. 3.9:9], wäre dies ohne Seiteneffekte möglich. Allerdings müsste dafür zusätzlich zur Registrierungs-Member-Funktion sowohl der Aufruf-Wrapper als auch das Join-Point-Objekt verändert werden, sodass diese nicht mehr mit Zeigern, sondern mit Kopien arbeiten.
Weil kein sinnvoller Anwendungsfall für eine solche Funktionalität erkenntlich ist und die Implementierung den begrenzten Zeitrahmen dieser Arbeit sprengen würde, wird von einer Implementierung abgesehen.

[4] Integrale oder auch *integer* Typen sind *bool*, *char*, *char16_t*, *char32_t*, *wchar_t*, und die „vorzeichenbehafteten“(*signed*) und „vorzeichenlosen“(*unsigned*) Varianten von *char*, *short int*, *int*, *long int* und *long long int*. Siehe C++-Standard[6] Kapitel 3.9.1 Absatz 2, 3 und 7.

[5] Die vordefinierten Operatoren, die eine nicht-konstante Referenz als Parameter erwarten[6] und im Kapitel 13.6 Absatz 18 bis einschließlich 22 des C++-Standards[6] definiert sind, gelten nicht für Bitfelder. Stattdessen sind im Kapitel 5.17 des C++-Standards[6] für Bit-Felder explizite Definitionen ohne Referenzen vorhanden.

[6] Folgende vordefinierte Operatoren sind betroffen:
- (Zusammengesetzte) Zuweisungs-Operatoren (siehe Tabelle A.1)
- Komma-Operator
- Präfix-Inkrement- und Präfix-Dekrement-Operator

4.3.3 Postfix Inkrement- und Dekrement-Operatoren

Wie im C++-Standard[6] Kapitel 13.5.7 beschrieben, besitzen die (impliziten) Funktionsdefinitionen der Postfix Inkrement- und Dekrement-Operatoren ein zweites Argument vom Typ *int*. Dieses Argument dient nur zur Unterscheidung, ob der Pre- und Postfix-Operator aufgerufen werden soll und wird sonst nicht weiter verwendet. Da innerhalb eines Match-Ausdruckes eines Pointcut-Ausdruckes nicht nur die überladenen Operatoren sondern auch die vordefinierten Operatoren durch die Funktionsschreibweise gefunden werden, muss bei den vordefinierten Postfix Inkrement- und Dekrement-Operatoren bei der Registrierung im AspectC++-Modell das *int*-Argument beachtet werden. Das heißt konkret, dass dieses in der Signatur und zu der Liste der Argumente des AspectC++-Modell-Elements, das der impliziten Operator-Funktionsdefinition entspricht, hinzugefügt werden muss. In der Liste der Argumente wird dieses Argument als Default-Argument definiert, da dieses bei dem Aufruf des vordefinierten Operators nicht vorhanden ist.
Durch diese Maßnahmen wird gewährleistet, dass die Postfix Inkrement- und Dekrement-Operatoren in dem Match-Ausdruck über das *int*-Argument gefunden werden können, aber das fehlende Argument im Aufruf während des Einwebens des Advice-Quelltext keine Probleme bereitet.

4.3.4 Adress-Operator für Member(-Funktions)-Zeiger

Falls der unäre Adress-Operator `&` für einen Member oder eine Member-Funktion aufgerufen wird, also die Speicheradresse des Members bzw. der Member-Funktion zurückgegeben werden soll, entsteht im Folgenden beschrieben Problematik.
Da der Typ des Rückgabewert ein Zeiger auf einen Member bzw. eine Member-Funktion ist, folgt daraus, dass der Typ des Arguments des Adress-Operators ein Member bzw. eine Member-Funktion ist. Diese beiden Typen existiert allerdings nur implizit und können nicht als Typ für eine Variable verwendet werden. Infolgedessen lässt sich das Argument nicht in einer Variable zwischenspeichern, was aber für ein vollständiges Einweben des Advice-Quelltextes notwendig wäre. Daraus folgt, dass ein Einweben von Advice-Quelltext an solchen Join-Points nur möglich wäre, wenn innerhalb des Advice-Quelltext nicht auf das Argument zugegriffen würde. Dieses teilweise Einweben wäre durchaus aufwendig und da sich außerdem die Frage bezüglich des Verhältnisses von Zeitaufwand und Relevanz stellt, wurde in diese Arbeit auf eine Implementierung dieses Operators für Member oder Member-Funktionen verzichtet. Ein AST-Knoten eines solchen Operators wird somit für die weitere Verarbeitung ignoriert.

4.3.5 Zeiger-zu-Member-Operatoren für Member-Funktions-Zeiger

Die Zeiger-zu-Member Operatoren `.*` und `->*`[6, Kap. 5.5] erwarten auf der linken Seite ein Objekt bzw. einen Zeiger auf ein Objekt und auf der rechten Seite einen Zeiger auf einen Member oder eine Member-Funktion. Das Ergebnis dieser Operatoren ist der Wert des dem Zeiger entsprechenden Members des angegebenen Objekts bzw. die Funktion der dem Zeiger entsprechenden Member-Funktion des angegebenen Objekts.
Im Fall, dass ein solcher Operator mit einem Member-Funktion-Zeiger aufgerufen wird,

darf die Funktion, die sich als Ergebnis des Operators ergibt, laut dem C++-Standard[6] Kapitel 5.5 Absatz 6 nur für den Funktionsaufruf verwendet und somit nicht zwischengespeichert werden. Aus diesem Grund ist das Einweben an solchen Join-Points nicht möglich.

4.3.6 Links-nach-rechts-Auswertung

Die links-nach-rechts-Auswertung oder engl. *left-to-right-evaluation* bezeichnet ein Vorgehen, bei dem Ausdrücke immer in der Reihenfolge ausgewertet werden, wie sie im Quelltext, von links nach rechts gelesen, vorkommen. Dieses Vorgehen wird in der Programmiersprache C++ für die folgenden vordefinierten Operatoren verwendet:

- Konjunktions-Operator[7] `&&`
- Disjunktions-Operator[7] `||`
- Konditional-Operator[7] `?:`
- Komma-Operator `,`

Da in dem vom AspectC++-Compiler generierten Quelltext die Ausdrücke, die für die Argumente des Operators ausgewertet werden, als Argumente für den Aufruf des Aufruf-Wrappers verwendet werden, ist die Auswertungsreihenfolge dieser Ausdrücke nicht mehr definiert. Dies liegt daran, dass die Auswertungsreihenfolge der Argumente eines Funktionsaufrufs, wie im Kapitel 8.3.6 Absatz 9 des C++-Standards[6] beschrieben, nicht spezifiziert ist. Infolgedessen ist ein Einweben des Advice nicht wie bisher möglich, da dadurch eine undefinierte Reihenfolge, wie in dem Beispiel im Quelltext 4.5 verdeutlicht, entstehen kann.

```
1  // Beispiel-Quelltext:
2  int i = 1;
3  int* ip = 0;
4  int i2 = (ip = &i, *ip); // Definierte Auswertungsreihenfolge, da
5                           // Argument-Ausdrücke des Komma-
6                           // Operators von links nach rechts aus-
7                           // gewertet werden.
8
9  // Beispiel-Quelltext mit eingewebtem Aufruf des
10 // Aufruf-Wrappers für den Komma-Operator:
11 int i = 1;
12 int* ip = 0;
13 int k = __call_0(ip = &i, *ip); // Undefinierte Auswertungsreihen-
14                                 // folge, da die Ausdrücke der
15                                 // Funktionsargumente in
```

[7] Für diesen vordefinierten Operatoren wird zusätzlich zur links-nach-rechts-Auswertung die Kurzschlussauswertung, die in Abschnitt 4.3.7 beschrieben wird, durchgeführt.

```
16                                  // beliebiger Reihenfolge aus-
17                                  // gewertet werden dürfen.
```

Quelltext 4.5: Beispiel-Quelltext und das vereinfachte Ergebnis des Einwebens ohne Beachtung der Auswertungsreihenfolge.
Dem *int*-Zeiger `ip` muss zuerst eine Adresse zugewiesen werden, bevor er dereferenziert werden darf. Im Ergebnis des Einwebens ist dies nicht mehr garantiert und es kann ein Speicherzugriffs-Fehler oder undefiniertes Verhalten entstehen.

Die in dieser Arbeit gewählte Methode, um dieses Problem zu umgehen, ist das Einweben eines Lambda-Ausdruckes[8], in dem die Ausdrücke der Argumente in der richtigen Reihenfolge ausgewertet werden. Durch die Verwendung des „&“ in der *Capture*-Liste des Lambda-Ausdruckes werden zuätzlich zu allen außerhalb des umgebenden Gültigkeitsbereichs verfügbaren Variablen und Funktionen auch alle nicht-statischen lokalen Variablen per Referenz in dem Rumpf des Lambda-Ausdruckes verfügbar gemacht[6, Kap. 5.1.2].
Daher sind in dem Lambda-Ausdruck die gleichen Variablen und Funktionen wie außerhalb vorhanden, wodurch es bei der Auswertung der Argument-Ausdrücke innerhalb des Lambda-Ausdruckes keine Einschränkungen gibt.
Ein Array (als *std::vector<void*>*) speichert die Adressen[9] der aus den Argument-Ausdrücken resultierenden Objekte und stellt den Rückgabewert des Lambda-Ausdruckes dar.
Der soeben beschriebene Lambda-Ausdruck wird in einem Aufruf gekapselt und als Argument an den Aufruf-Wrapper übergeben. Dies führt dazu, dass der Lambda-Ausdruck zur gleichen Zeit wie vorher die Argumente des vordefinierten Operators ausgeführt wird. Zusätzlich wird der Aufruf-Wrapper in diesen Fällen so geändert, dass er ein Array mit Zeigern auf die Ergebnisse der Operator-Argumente als Parameter verarbeiten kann. Momentan wird dies durch eine Makro-Ersetzung der bisherigen Argumente durch einen Ausdruck, der den Wert des entsprechenden Argumentes aus dem durch den Lambda-Ausdruck generierten Array zurückgibt, erreicht. Diese Ersetzung wird nur innerhalb des Rumpfes des Aufruf-Wrappers durchgeführt. So wird erreicht, dass die Generierung des Quelltextes innerhalb des Rumpfes des Aufruf-Wrappers im Fall der links-nach-rechts-Auswertung nicht geändert werden muss. Der Nachteil ist, dass die Makro-Ersetzung alle Bezeichner unabhängig von ihrem Typ ersetzt. Infolgedessen könnten auch Aufrufe von (Member-)Funktion, die den gleichen Namen wie eines der Argumente haben, durch den Array-Zugriffs-Ausdruck ersetzt werden. Dies würde dann zu falschen C++-Quelltext führen.
Allerdings ist kein Fall bekannt, in dem innerhalb des Aufruf-Wrappers eines vordefinierten Operators, der die links-nach-rechts-Auswertung verwendet, eine (Member-)Funktion

[8] Lambda-Ausdrücke bieten ein Möglichkeit, um auf kurze und prägnante Weise ein einfaches Funktions-Objekt zu erstellen[6, Kap. 5.1.2:1]

[9] Falls das aus einem Argument-Audruck resultierende Objekt keine Adresse hat und somit ein *(p)rvalue*[6, Kap. 3.10] ist, wird dieses Objekt auf den Heap kopiert und die Adresse des sich auf dem Heap befindenden Objektes verwendet. (Dieser Kopiervorgang würde auch ohne das Einweben während der Übergabe als Argument an den vordefinierten Operator stattfinden.)
Vor dem Ende des Gültigkeitsbereichs des Aufruf-Wrappers wird das sich auf dem Heap befindende Objekt von dem Heap gelöscht und der allokierte Speicher wieder freigegeben.

aufgerufen wird, die den gleichen Namen wie eines der Argumente hat.
Ein weiterer Nachteil entsteht durch die Verwendung der Lambda-Ausdrücke an sich, denn diese sind erst mit der Version *C++0x* der Programmiersprache C++ eingeführt worden. Somit muss der Ausgabe-Quelltext des AspectC++-Compilers unter Umständen mit einer neueren Version des verwendeten C++-Compilers aufgerufen werden[10]. Aus diesem Grund ist das Einweben an solchen Join-Points optional und kann durch ein gesetztes Makro während des Kompiliervorganges des AspectC++-Compilers aktiviert werden. Im Fall, dass diese Funktionalität deaktiviert ist, werden Join-Points an Aufrufen der oben genannten vordefinierten Operatoren ignoriert.

4.3.7 Kurzschlussauswertung

Die Kurzschlussauswertung oder engl. *short-circuit-evaluation* bezeichnet eine Strategie, bei der ein oder mehrere Teilausdrücke eines Gesamtausdruckes nicht ausgewertet werden, falls die Ergebnisse der Teilausdrücke keinen Einfluss auf das Ergebnis das Gesamtausdruckes haben. Die Kurzschlussauswertung verwendet die links-nach-rechts-Auswertung, die im vorherigem Abschnitt beschrieben wurde, wodurch die Menge der vordefinierten Operatoren, die die Kurzschlussauswertung verwenden, eine Teilmenge der Menge aller vordefinierten Operatoren, die die links-nach-rechts-Auswertung verwenden, ist. In der Programmiersprache C++ existieren zum einen die beiden binären, booleschen, vordefinierten Operatoren Konjunktion(`&&`) und Disjunktion(`||`), die das zweite Argument nicht auswerten, falls durch das Ergebnis des ersten Arguments bereits das Gesamtergebnis feststeht. So ist das Ergebnis einer Konjunktion von zwei booleschen Werten immer *false*, falls das erste Argument den Wert *false* hat. Analog gilt für die Disjunktion zweier boolescher Werte, dass das Gesamtergebnis immer *true* ist, falls das erste Argument den Wert *true* hat. In diesen Fällen wird durch die Kurzschlussauswertung verhindert, dass die Ausdrücke, die den Wert des zweiten Arguments bestimmen, ausgewertet werden.
Zum anderen existiert in der Programmiersprache C++ der ternäre Konditional-Operator `?:`, der je nach Wert des booleschen ersten Arguments, den Ausdruck, der den Wert des zweiten Arguments bestimmt, oder den Ausdruck, der den Wert des dritten Arguments bestimmt, auswertet. Falls der Wert des ersten Arguments *true* ist, wird der Ausdruck des zweiten Arguments und im anderen Fall der Ausdruck des dritten Arguments ausgewertet.
Die Kurzschlussauswertung ist sehr nützlich, da durch diese das überflüssige Auswerten von Ausdrücken vermieden wird und somit die Leistung und Effizienz des vom Compiler generierten Programms zunimmt. Während des Einwebens von Advice an Aufrufen solcher Operatoren ist sie allerdings problematisch, da die kurzgeschlossenen Argumente nicht zur Verfügung stehen. Um solche Fälle zu behandeln, wird die im vorherigem Abschnitt 4.3.6 beschriebene Methode erweitert, sodass für kurzgeschlossene Argumente

[10] Theoretisch könnten die Lambda-Ausdrücke mit lokalen Klassen „nachgebaut“ werden. Dazu müssten alle in den Argument-Ausdrücken verwendeten lokalen Variablen einzeln übergeben werden. Auch das Einweben von Advice an Join-Points, die sich innerhalb der Operatoren mit links-nach-rechts-Auswertung befindet, ist problematisch. Aus diesen und weiteren hier nicht weiter beschriebenen Problemen wurde in dieser Arbeit von der Implementierung abgesehen.

ein Null-Zeiger in das Array, das die Zeiger auf die Ergebnisse vorhält, eingefügt wird. Dadurch entsteht über die Null-Zeiger eine Möglichkeit zu beschreiben, dass der entsprechende Argument-Ausdruck nicht ausgewertet wurde und somit kein Wert für das Argument verfügbar ist.
Im Advice-Quelltext kann der Benutzer den über das Join-Point-Objekt erhaltenen Argument-Zeiger mit dem Null-Zeiger vergleichen und dadurch erkennen, dass das entsprechende Argument nicht verfügbar ist, und dass daher der Zeiger nicht dereferenziert werden darf.
Mit der *args*-Pointcut-Funktion kann definiert werden, welche Typen die Argumente eines Join-Points besitzen dürfen, damit an diesem Join-Point der Advice eingewebt wird. Alternativ kann in der `args`-Pointcut-Funktion eine Liste von sogenannten *Kontext-Variablen*[16, S. 27] angegeben werden.

```
1 advice call("% operator %(...)") && args(a1, a2) : after(int a1, ↩
     int a2) {
2   cout << "Ein Operator mit zwei Argumenten des Typs \"int\" wurde"
3        << "mit den Werten " << a1 << " und " << a2 << " aufgerufen"↩
             << endl;
4 }
```

Quelltext 4.6: Quelltext eines `after`-Advice für alle Operatoren, die zwei Argumente des Typs *int* besitzen. Über die Kontext-Variablen `a1` und `a2` werden die Werte der Argumente als Parameter des Advice verfügbar gemacht.
An einem Aufruf eines vordefinierten Operators, der die Kurzschlussauswertung verwendet, darf ein solcher Advice nicht eingewebt werden, da über die *args*-Pointcut-Funktion auf das zweite, im Fall einer Kurzschlussauswertung nicht vorhandene, Argument zugegriffen wird.

Diese werden - wie in dem Quelltext 4.6 zu sehen - in der Argumenten-Liste des Advice deklariert und sind dadurch im Advice-Quelltext verfügbar. Da dafür die Werte der Argumente und nicht wie das Join-Point-Objekt die Zeiger zur Verfügung stellt, darf diese Pointcut-Funktion bei vordefinierte Operatoren mit Kurzschlussauswertung nur für das erste Argument[11] verwendet werden. Die Verwendung für das zweite und eventuelle dritte Argument ist nicht erlaubt, da sonst im Fall einer Kurzschlussauswertung im von dem AspectC++-Compilers eingewebtem AspectC++-Quelltext ein Null-Zeiger dereferenziert würde. Daher darf ein solcher Advice an Join-Points mit Kurzschlussauswertung nicht eingewebt werden.
Als zuätzliche Erweiterung sollte in solchen Fällen der Advice nicht eingewebt und eine entsprechende Warnung für den Benutzer generiert werden.

[11] Für das erste Argument ist immer ein Wert vorhanden, da der Ausdruck des ersten Argumentes in allen Fällen ausgewertet wird.

4.3.8 Kopier-Zuweisungs-Operator

Der Kopier-Zuweisungs-Operator oder engl. *copy-assignment-operator* wird aufgerufen, wenn ein Objekt, das aus einer Klasse instanziiert wurde, während einer Zuweisung kopiert werden muss[6, Kap. 12.8:1]. Falls der Benutzer für eine Klasse keinen Kopier-Zuweisungs-Operator definiert hat, wird von dem C++-Compiler automatisch eine Definition erstellt, in der das Objekt memberweise kopiert wird[6, Kap. 12.8:28].
Im Clang-AST werden Kopier-Zuweisungs-Operatoren als Knoten der Klasse *clang::CXXOperatorCallExpr* und nicht wie Zuweisungs-Operatoren auf Objekten, die nicht aus einer Klasse instanziiert werden[12], als Knoten der Klasse *clang::BinaryOperator* dargestellt.
Für die AST-Knoten der Klasse *clang::CXXOperatorCallExpr* existiert im bisherigem AspectC++-Compiler bereits eine Traverse-Member-Funktion, allerdings wurden die Aufrufe von den von dem Compiler generierten und somit vordefinierten Kopier-Zuweisungs-Operatoren bisher nicht weiter verarbeitet.
Dies wurde geändert, sodass auch Aufrufe der vordefinierten Kopier-Zuweisungs-Operatoren im AspectC++-Modell registriert werden. Da die Klasse *clang::CXXOperatorCallExpr* von der Klasse *clang::CallExpr* erbt, wird dafür die bereits vorhandene Registrier-Member-Funktion *ClangModelBuilder::register_call* verwendet.

4.3.9 Implizite Dereferenzierungen

In der Programmiersprache C++ existieren neben dem expliziten Dereferenzierungs-Operator `*` andere vordefinierte Operatoren, in denen implizit eine Dereferenzierung durchgeführt wird. Im Folgenden sind diese zusammen mit einer äquivalenten Schreibweise, in der die implizite Dereferenzierung explizit zu sehen ist, aufgeführt.

- Member-Zugriff-Operator `->` (`a->b` ⇔ `(*a).b`)
- Array-Index-Operator `[]` (`a[b]` ⇔ `*(a + b)`)
- Zeiger-zu-Member-Operator `->*` (`a->*b` ⇔ `(*(a)).*b`)
- Zeiger-zu-Member-Operator `.*` (<äquivalente Schreibweise nicht vorhanden>)
- Aufruf eines Funktions-Zeigers (`a()` ⇔ `(*a)()`)

Da die Implementierung in dieser Arbeit unter anderem zur Erhöhung der Fehlertoleranz durch Überprüfung von Zeigern vor deren Dereferenzierung führen soll, ist es wünschenswert, an möglichst vielen Dereferenzierungen den Zeiger-Validierungs-Quelltext einzuweben.
Das Einweben von Advice an einem Aufruf des Member-Zugriff-Operators `->`[6, Kap. 5.2.5] oder an einem Funktionsaufruf[6, Kap. 5.2.2] ist im Gegesatz zu den anderen, oben genannten vordefinierten Operatoren mit impliziten Dereferenzierungen aktuell nicht

[12] Diese Nicht-Klassen-Typen sind Fundamentale Typen[6, Kap. 3.9.1], Array-Typen[6, Kap. 8.3.4], Referenz-Typen[6, Kap. 8.5.3], Zeiger-Typen[6, Kap. 8.3.1] oder Aufzählungs-Typen[6, Kap. 7.2].

möglich. Daher besteht auch aktuell keine Möglichkeit, die Zeiger, die implizit dereferenziert werden, zu erhalten. Eine Möglichkeit wäre, das Einweben in den beiden genannten Fällen komplett zu ermöglichen. Dies zu implementieren ist allerdings sehr aufwendig und nur mit Einschränkungen möglich, da zum Beispiel im Fall des Operators `->` der Typ *Member* bzw. *Member-Funktion* des zweiten Arguments nur implizit existiert und somit nicht zwischengespeichert werden kann. Auch im Fall des Funktionsaufrufs wären diese Typen ein Problem. Aus diesem Grund wird in dieser Arbeit nur das Einweben von Advice an der impliziten Dereferenzierung ermöglicht.

Im Fall des Member-Zugriff-Operators `->` wird dazu der Ausdruck `a->b()`[13] gedanklich in die äquivalente Form `(a*).b()` umgeformt und anschließend die Dereferenzierung im Teilausdruck `*a` im AspectC++-Modell registriert. Falls dann an diesem Join-Point ein Advice eingewebt werden soll, wird der Teilausdruck `a` als Argument des eingewebtem Aufruf-Wrapper-Aufrufs an den eingewebten Aufruf-Wrapper übergeben, sodass sich vereinfacht `wrapper_deref(a).b()` ergibt. Wie zu erkennen ist, wird außerdem der Pfeil(`->`) zu einem Punkt(`.`) transformiert, da das Ergebnis des Aufruf-Wrappers bereits der dereferenzierte Zeiger ist. Diese Transformation wird durch ein Entfernen des Quelltextes „->“ und einem Einfügen des Quelltextes „.“ erreicht. Allerdings werden in dem AspectC++-Compiler Änderungen, die C++-Quelltext entfernen direkt, und Änderungen, die C++-Quelltext einfügen, erst nach allen entfernenden Änderungen durchgeführt. Falls nun zuätzlich auch an dem Aufruf der Member-Funktion `b()` ein Advice eingewebt werden soll, müsste dieser zweite Einwebevorgang den durch das Einweben der impliziten Dereferenzierung eingefügten Punkt(`.`) wieder entfernen, damit sich `wrapper_memfunc(wrapper_deref(a))` ergibt[14]. Da der Punkt aber noch nicht eingefügt worden ist, wird dieser auch nicht entfernt, sondern zusammen mit allen anderen Einfügungen am Ende eingefügt, sodass sich `wrapper_memfunc(wrapper_deref(a).)` ergibt. Da dies aufgrund des Punktes keinen gültigen C++-Quelltext darstellt, dürfen an dem Aufruf des Member-Zugriff-Operators `->` nicht ein Advice für die implizite Dereferenzierung **und** den Aufruf der Member-Funktion eingewebt werden.
Der Aufbau und die Funktionsweise des Aufruf-Wrappers für eine implizite Dereferenzierung entspricht dem Aufbau und der Funktionalität eines Aufruf-Wrapper für eine explizite Dereferenzierung. Daraus folgt, dass das Ergebnis des Gesamtausdruckes `a->b` durch das Einweben eines Advice nicht verändert wird[15] und dass zusätzlich das erste Argument über einen Advice verfügbar ist.

Im Fall des Aufrufs eines Funktions-Zeigers wird der Ausdruck `a()` ebenfalls gedanklich in die äquivalente Form `(*a)()` umgeformt und die implizite Dereferenzierung im Teilausdruck `*a` im AspectC++-Modell registriert. Wenn ein Advice für diesen Join-Point eingewebt werden soll, wird der Ausdruck `a` über den Aufruf des Aufruf-Wrappers an den Aufruf-Wrapper übergeben. Innerhalb des Aufruf-Wrappers wird der Funktions-Zeiger wie bei einer expliziten Dereferenzierung eines Funktions-Zeigers behandelt, was dazu führt, dass das Ergebnis des Aufruf-Wrapper-Aufrufs ein Funktions-Objekt ist.
Durch das Einweben ist folglich der Typ des Ausdruckes `a` von einem Funktions-Zeiger-

[13] Der Fall `a->b` verläuft analog.
[14] Die Member-Funktion `b` wird innerhalb des Aufruf-Wrappers `wrapper_memfunc` aufgerufen.
[15] Vorausgesetzt, der Advice-Quelltext an sich verändert das Ergebnis nicht.

Typ zu einem Funktions-Objekt-Typ verändert worden. Da die Programmiersprache C++ aber sowohl das Aufrufen eines Funktions-Objekts als auch das Aufrufen eines Funktions-Zeigers selbst erlaubt und beide Aufrufe zu dem gleichen Ergebnis führen[6, Kap. 5.2.2], wird der Gesamtausdruck durch das Einweben nicht verändert.

5 Evaluation

Nach der Implementierung wird die erweiterte Variante[1] des AspectC++-Compilers evaluiert. Dazu werden in den folgenden zwei Abschnitten die Aspekte vorgestellt, die zur Valdierung verwendet werden können.
Im darauf folgendem dritten Abschnitt werden die zur Bewertung der Fehlertoleranz verwendeten Methoden und Werkzeuge vorgestellt, bevor in den anderen Abschnitten die Ergebnisse der Evaluation präsentiert werden.
Evaluiert werden die eCos-Kernel-Tests des Betriebssystems eCos, die Qt-Beispiele sowie die Puma-Bibliothek. Während die eCos-Kernel-Tests bezüglich des Kompiliervorganges, der Größe und der Laufzeit-Faktoren sowie der Fehlertoleranz untersucht werden, werden die QT-Beispiele und die Puma-Bibliothek aus Zeitgründen und weil keine Umgebung zur Messung der Fehlertoleranz vorhanden ist, nur bezüglich des Kompiliervorganges bewertet.

5.1 Zeiger-Validierung

Wie bereits im Kapitel 2.1 beschrieben, ist eine Zeiger-Validierung aufgrund des eingeschränkten Wertebereiches eines Zeigers möglich. Um dies genauer zu beschreiben, wird zuerst das in Tabelle 5.1 gezeigte Speicherlayout einer für das Betriebssystem *eCos* (siehe Abschnitt 5.4) erstellten Anwendung vorgestellt. Der Unterschied dieses Layouts zu den meisten anderen Speicherlayouten ist, dass sich das *stack*-Segment, in dem die Programm-Stacks gespeichert werden, innerhalb des *data*-Segments befindet. Der Speicherbereich der Anwendung teilt sich, wie in Tabelle 5.1 gezeigt, in die Speichersegmente *text*, *data* und *heap* auf. Zusätzlich wird ein Teil des Speichers freigelassen, damit der Heap in diesen Bereich hineinwachsen kann.
Das *text*-Segment beinhaltet die Programminstruktionen, die durch den Compiler aus den (Member-)Funktionen eines Quelltextes kompiliert werden. Im *data*-Segment hingegen werden Variablen gespeichert. Ein Teil dieses Segmentes wird in die Bereiche *bss* und *stack* aufgeteilt, sodass eine Dreiteilung wie in Tabelle 5.1 entsteht. Durch diese wird es ermöglicht, die verschiedenen Arten von Variablen getrennt zu speichern. So wird das *data*-Segment für initialisierten und das *bss*-Segment für die uninitialisieren globalen und statischen Variablen verwendet. Das *stack*-Segment hingegen speichert die lokalen Variablen und insbesondere die Argumente eines Funktionsaufrufes.
Mit Kenntnis dieser Informationen über das Speicherlayout kann ein Zeiger weiter differenziert werden. Im Folgenden wird zwischen einem Funktions-Zeiger und einem Objekt-

[1] Die im Rahmen dieser Arbeit entwickelte Variante des AspectC++-Compilers, die die Unterstützung für vordefinierte Operatoren bietet, wird als ***neue***, ***erweiterte*** oder ***angepasste*** Variante bezeichnet.

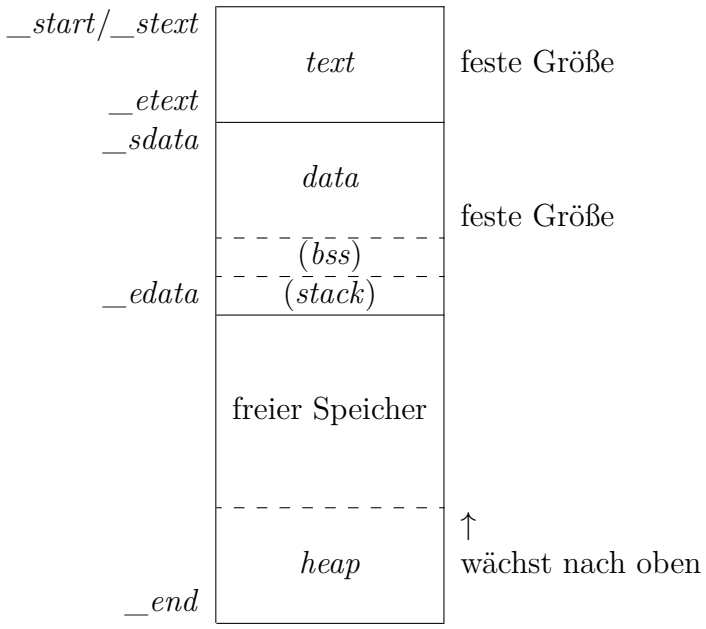

Tabelle 5.1: Speichersegmente[25, Kap. 10.4] in eCos und Symbole, die die Grenzen dieser Speichersegmente markieren

Zeiger unterschieden. Ersterer darf nur auf Funktionen oder statische Member-Funktionen und somit nur in das *text*-Segment zeigen. Der Objekt-Zeiger hingegen zeigt auf eine Variable oder den Heap und darf somit in das *data*- oder *heap*-Segment zeigen. Ob es sich bei einem gegebenen Zeiger um einen Funktions- oder Objekt-Zeiger handelt, kann mittels Templates zur Kompilierzeit aus dem Kontext des Zeigers abgeleitet werden.
In dieser Arbeit wird dazu der bereits fertig implementierte *TypeTraits*-Namensraum[7, Kap. 6] verwendet. In diesem Namensraum befindet sich für jede gängige Zeiger-Kategorie und auch für Funktions- und Objekt-Zeiger eine Strukur oder engl. *struct* mit entsprechenden Template-Argumenten und einem Aufzählungswert, anhand dem die vom Compiler ausgewählte Klasse und somit die Kategorie des Zeigers erkannt werden kann. Da diese Auswahl der Zeiger-Kategorie und die daraus resultierende Validierung bereits zur Kompilierzeit feststeht, können die zur Laufzeit auftretenden Soft-Errors nur den Zeiger an sich und nicht die Validierung verändern.
Eine solche Validierung besteht anschließend darin, mit einer simplen Bereichs-Überprüfung festzustellen, ob der jeweilige Zeiger in ein erlaubtes Segment zeigt.

Ein Aspekt, der die beschriebenen Vorgänge zur Überprüfung eines Zeigers in einen bestehenden Quelltext einwebt, ist in dem Quelltext 5.1 definiert[2].
Um die benötigten Grenzen der Speichersegmente dynamisch zu erkennen, werden die in Tabelle 5.1 markierten Symbole *_stext*, *_etext* und *_end* verwendet. Anschließend werden diese je nach Zeiger-Kategorie mit dem Zeiger verglichen.

[2]Damit dieser auf eine Seite passt wurde er an einigen Stellen gekürzt. Die vollständige Version befindet sich in dem Anhang in dem Quelltext A.2

```
1  #include "PtrType.h" // TypeTraits
2  extern char _stext, _etext, _end; // Segmentgrenzen
3  void ecc_panic() // Fehler-Behandler
4  // Aspekt, der Zeiger validiert bevor diese dereferenziert werden:
5  aspect PointerValidator {
6    // Vor den Aufrufen von vordef. Dereferenzierungs-Operatoren:
7    advice call("% operator *(%*)") : before() {
8      typedef JoinPoint::template Arg<0>::ReferredType PointerType;
9      const PointerType pointer = *(tjp->arg<0>());
10     // Zeiger-Kategorie mittels TypeTraits[7] ermitteln:
11     const TypeTraits::result_type pointer_category = TypeTraits::↩
           pointerType<PointerType>::result;
12     // Segmentgrenzen für Vergleich vorbereiten:
13     const PointerType text_start_addr = (PointerType)&_stext;
14     const PointerType text_end_addr = (PointerType)&_etext;
15     const PointerType program_end_addr = (PointerType)&_end;
16     // Unterscheidung anhand der Zeiger-Kategorie:
17     switch(pointer_category) {
18       case /* Kein Funktions- oder Objekt-Zeiger */:
19         ecc_panic(); // Fehler signalisieren:
20       case /* Objekt-Zeiger */:
21         // Zeiger muss in das data- oder heap-Segment zeigen:
22         if(pointer < text_end_addr || pointer > program_end_addr) {
23           ecc_panic(); // Fehler signalisieren:
24         }
25         break;
26       case /* Funktions-Zeiger */:
27         // Zeiger muss in das text-Segment zeigen:
28         if(pointer < text_start_addr || pointer > text_end_addr) {
29           ecc_panic(); // Fehler signalisieren:
30         }
31         break;
32 } } }
33 ;
```

Quelltext 5.1: Aspekt zur Zeiger-Validierung

5.2 Arithmetik-Validierung

Die Arithmetik-Validierung umfasst die Überprüfung aller ganzzahligen Additionen, Subtraktionen, Multiplikationen und Divisionen. Um solche Berechnungen zu validieren, können die sogenannten *Redidue*-Codes[8, S. 9] verwendet werden, die auf der Formel 5.1 basieren.

$$(N_1 \oplus N_2)\%A \stackrel{?}{=} (N_1\%A \oplus N_2\%A)\%A \qquad N_i, A \in \mathbb{Z}; A \neq 0 \tag{5.1}$$

In dieser Gleichung stellen N_1, N_2 die ganzzahligen Operanden und somit die Eingabe der Berechnung dar. $\oplus$ symbolisiert den Platzhalter für den gewünschten Operator ($+$, $-$, $*$ oder $/$) und A stellt eine Konstante dar, die zur Bildung des Modulo-Wertes verwendet wird. Der Wert der Konstante A sollte allerdings nicht beliebig gewählt werden, denn durch einen Einzel-Bit-Flip innerhalb einer binären Zahl wird der Wert dieser Zahl immer um eine Zweierpotenz erhöht oder erniedrigt. Falls somit die Konstante A genau dieser Zweierpotenz entsprechen würde, würde der Modulo-Wert der fehlerfreien und der fehlerhaften Zahl mit dieser Konstante identisch sein. Eine Fehlererkennung wäre in einem solchen Fall nicht möglich. Um diese Situationen auszuschließen darf die Konstante keine Zweierpotenz sein.
Falls die Gleichung 5.1 nicht gilt, ist mit Sicherheit ein Fehler aufgetreten. In dem anderen Fall kann nur vermutet werden, dass kein Fehler aufgetreten ist.
Um nun die beispielsweise durch *Soft-Errors* entstandenen Einzel-Bit-Flips während einer Berechnung in einer Anwendung zu erkennen, können diese Berechnungen mit den *Residue*-Codes validiert werden. Die für eine Validierung einer Addition notwendigen Änderungen an einem Quelltext können mit dem Aspekt in Quelltext 5.2 beschrieben werden.

```
1  // Fehler-Behandler:
2  void ecc_panic() __attribute__ ((noreturn, noinline));
3
4  aspect CheckCalculation {
5    // Konstante für die Bildung des Modulo-Wertes:
6    const static int A = 10;
7
8    // Pointcut für ganzzahlige Plus-Operatoren:
9    pointcut built_in_integer_plus() =
10       "int operator +(int,int)" ||
11       "unsigned int operator +(unsigned int,unsigned int)" ||
12       // ... (unsigned) short int || (unsigned) long int ||
13       // (unsigned) long long int || (unsigned) char;
14
15   // Call-Advice für alle ganzzahligen Additionen:
16   advice call(built_in_integer_plus()) : after() {
17     // Typ der Argumente und des Rückgabewertes:
18     typedef JoinPoint::template Arg<0>::ReferredType Type;
19     // Werte der Argumente und Rückgabewert:
20     Type summand_one = *(tjp->arg<0>());
21     Type summand_two = *(tjp->arg<0>());
22     Type result = *static_cast<Type*>(tjp->result());
23     // Validierung durch den Residue Code:
24     if(result % A != (summand_one % A + summand_two % A) % A) {
25       // Fehler signalisieren:
26       ecc_panic();
27     }
```

```
28     }
29 };
```

Quelltext 5.2: Aspekt zur Validierung einer ganzzahligen Addition
Advice für Subtraktions-, Multiplikations- und Divisions-Operatoren werden analog zu diesem erstellt.

Der Pointcut in Zeile 9 beschreibt alle vordefinierten Additionsoperatoren, die mit ganzen Zahlen arbeiten. Hinter dem Aufruf eines solchen Operators wird durch den in Zeile 16 definierten Advice eine Überprüfung eingewebt, die die Summe unter Verwendung der Formel 5.1 und der Summanden auf Fehler untersucht. Der Wert der Konstante A wurde in diesem Beispiel als 10 gewählt.
Die weiteren Advice und Pointcuts für die Subtraktions-, Multiplikations- sowie Divisions-Operatoren werden analog gebildet und an dieser Stelle aus Gründen der Übersichtlichkeit vernachlässigt.

5.3 Bewertung von Fehlertoleranz

Um die Fehlertoleranz einer Anwendung oder eines Systems zu bewerten, muss das Verhalten der Anwendung oder des Systems im Falle eines Fehlers beobachtet und ausgewertet werden. Fehler können in vielen Stellen eines Systems, wie zum Beispiel dem Arbeitsspeicher, der CPU oder einem Bus, injiziert werden. Im Rahmen dieser Arbeit beschränken sich die Fehlerinjektionen allerdings nur auf den *data*-Bereich einer Anwendung, da für diese Fehlerinjektionen bereits eine funktionierende Umgebung vorhanden ist und Implementierung einer eigenen Fehlerinjektions-Umgebung für die ALU oder andere Teile eines Systems den Rahmen dieser Arbeit sprengen würde. Daraus folgt, dass nur die in Abschnitt 5.1 beschriebene Zeiger-Validierung und nicht die in Abschnitt 5.2 vorgestellte Arithmetik-Validierung in Bezug auf die Fehlertoleranz evaluiert wird.
Weiterhin werden in dieser Arbeit zur Fehlerinjektion nur Einzel-Bit-Flips oder engl. *single-bit-flips* innerhalb des *data*-Segments einer Anwendung durchgeführt. Diese Einzel-Bit-Flips lassen sich durch die Anwendung des logischen XOR-Operators auf ein Bit des Speichers beschreiben. Die Position einer solchen Fehlerinjektion lässt sich mittels des Zeitpunktes der Ausführung und des Ortes in dem Speicher eindeutig beschreiben. Der durch diese endlichen Zeitpunkte und endliche Orte aufgespannte endliche Raum nennt sich Fehlerraum und ist beispielhaft in Abbildung 5.2 dargestellt.
Um die Fehlertoleranz einer Anwendung zu messen, wird gleichverteilt für jede Kombination aus Zeitpunkt der Ausführung und Ort im Speicher, d.h. für jeden Punkt in dem Fehlerraum, ein sogenanntes Fehlerinjektions-Experiment durchgeführt. Während eines solchen Fehlerinjektions-Experimentes wird die Anwendung für eine zu dem entspechenden Punkt im Fehlerraum gehörende Anzahl von Taktzyklen ausgeführt und anschließend das zu dem Punkt im Fehlerraum gehörende Bit geflippt. Danach wird beobachtet, wie sich die Anwendung nach der Injektion des Fehlers verhält.

- Stürzt die Anwendung beispielsweise durch eine Speicherzugriffsverletzung oder eine Division durch Null ab, wird das Ergebnis als ***TRAP*** bezeichnet.

- Ein ***TIMEOUT*** tritt auf, falls die Anwendung durch die Fehlerinjektion eine längere Laufzeit als ohne Fehlerinjektion benötigte und daher beendet werden musste.
- Falls die Anwendung zwar bis zum Ende durchläuft, aber fehlerhafte Ausgaben erzeugt, ist das Ergebnis des Fehlerinjektions-Experimentes eine *Silent Data Corruption*(***SDC***).
- Das Ergebnis ***OK*** markiert, dass die Fehlerinjektion keinen Einfluss auf das Verhalten der Anwendung hat.

Nach Ausführung aller Fehlerinjektions-Experimente kann anhand des Verhältisses von fehlerhaften Ergebnissen (*TRAP*, *TIMEOUT*, *SDC*) zu fehlerfreien Ergebnissen (*OK*) eine Aussage über die Fehlertoleranz einer Anwendung getroffen werden.

5.3.1 FAIL*-Framework

Um die oben genannten Fehlerinjektions-Experimente automatisiert durchzuführen, wird in dieser Arbeit das Fehlerinjektionssystem *FAIL**[3][26] verwendet. Das Backend von FAIL*, in dem die Fehler injiziert werden, kann durch eine lose Kopplung, die mit Hilfe einer API und der Verwendung von Aspekten erreicht wird, leicht ausgetauscht werden[26, S. 4]. Im Rahmen dieser Arbeit wird allerdings als einziges Backend der x86 PC-Simulator *Bochs*[27] eingesetzt.
Die grundlegende Funktionsweise von FAIL* ist in Abbildung 5.1 schematisch dargestellt. Wie zu erkennen ist, gliedert sich FAIL* grob in zwei Komponenten. Die in der Abbildung oben dargestellte Komponente ist den Kampagnen-Kontroll-Server, der die sogenannten Kampagnen und deren Ergebnisse verwaltet. Eine Kampagne enthält Informationen zu der zu untersuchenden Anwendung sowie eine Menge von Beschreibungen von Fehlerinjektions-Experimenten. Aus diesen Kampagnen erstellt der Kampagnen-Kontroll-Server eine Liste von Aufträgen, in der jeder Auftrag Informationen für genau ein Fehlerinjektions-Experiment einer Anwendung beinhaltet.
Die zweite Komponente im unteren Teil der Abbildung 5.1 besteht aus den Instanzen des FAIL*-Clients. Ein solcher Client holt sich von dem Kampagnen-Kontroll-Server einen Auftrag ab und führt anschließend ein Fehlerinjektions-Experiment an der durch den Auftrag beschriebenen Position im Fehlerraum der Anwendung durch. Das Ergebnis des Fehlerinjektions-Experimentes wird danach an den Kampagnen-Kontroll-Server übertragen, der diese für die anschließende Analyse in einer Datenbank speichert.
Damit nicht jede einzelne Instanz des FAIL*-Client für jedes Fehlerinjektions-Experiment den Simulator initialisieren und eCos hochfahren muss, wird von FAIL* der Zustand des initialisierten Simulators inklusive hochgefahrenem Betriebssystem gespeichert. Die FAIL*-Clients können anschließend diesen Zustand laden und direkt mit dem Fehlerinjektions-Experiment beginnen. Eine weitere Optimierung kann durch die Reduzierung der Fehlerinjektions-Experimente erreicht werden. Dies ist zum einen möglich, da fehlerhafte Bits in einem Speicher unproblematisch sind, solange sie nicht gelesen werden.

[3] **FA**ult **I**njection **L**everaged; der Platzhalter ***** steht für das austauschbare Backend, in dem die Fehlerinjektionen stattfinden.

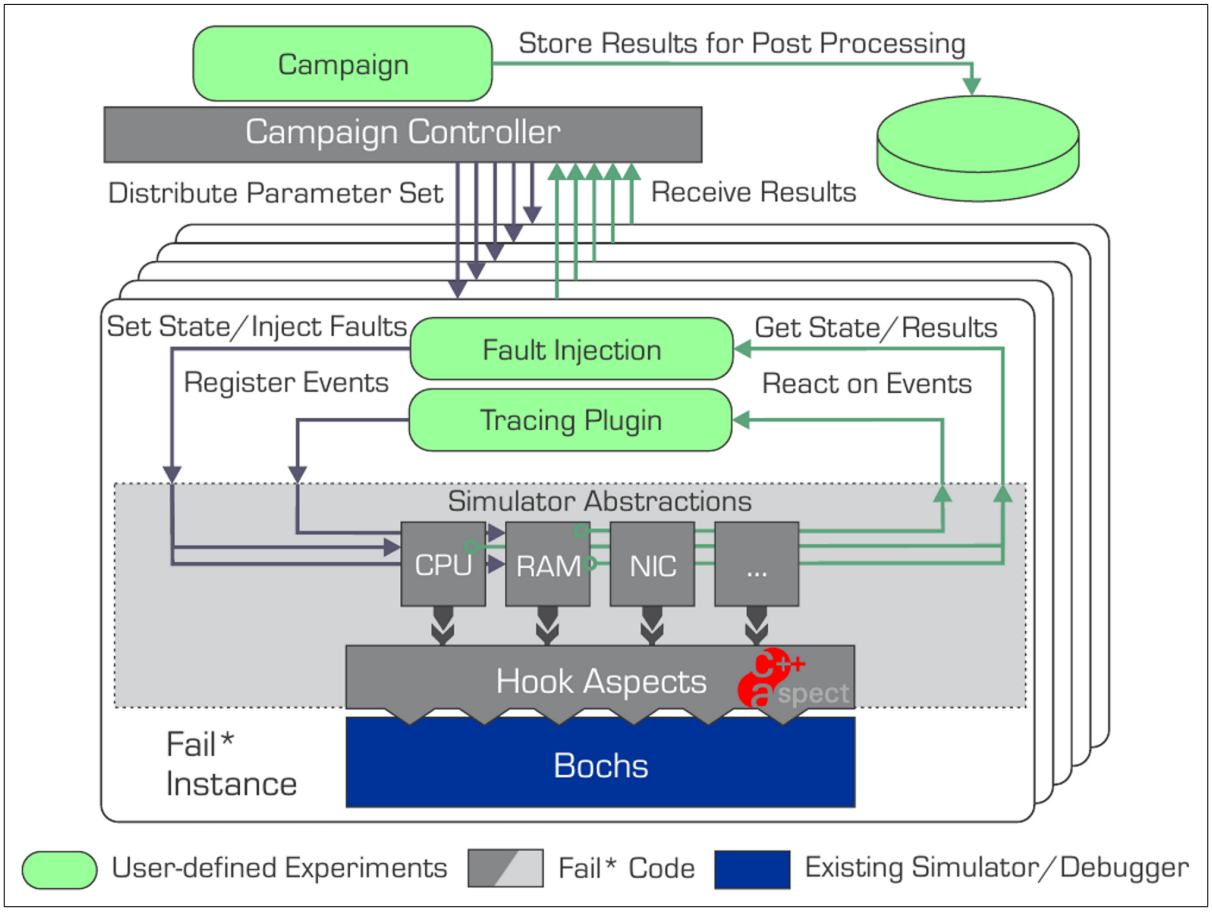

Abbildung 5.1: Übersicht der Funktionsweise von FAIL*
Der Kampagnen-Kontroll-Server („Campaign Controller“) verwaltet zentral die Aufträge mit den Informationen, an welchen Position im Fehlerraum ein Fehler injiziert werden soll. Die einzelnen Instanzen des FAIL*-Client holen sich von dem Kampagnen-Kontroll-Server einen Auftrag mit Informationen zu dem Fehlerinjektions-Experiment ab, führen das durch den Auftrag beschriebene Fehlerinjektions-Experiment durch und geben das Ergebnis an den Server zurück, der dieses zentral speichert. (Quelle: [26, S. 4])

Solche Situationen stellen die Fehlerinjektion 5 und 6 in Abbildung 5.2 dar. Wie an diesen Beispielen zu sehen, ist das Ergebnis eines Fehlerinjektions-Experiment immer *OK*, falls das durch die Fehlerinjektion geflippte Bit nach der Fehlerinjektion von der Anwendung nicht gelesen wird. Daher können solche Fehlerinjektions-Experiment an solchen Punkten im Fehlerraum direkt das Ergebnis *OK* zurückgeben.
Eine weitere Optimierung soll anhand des Beispiels in Abbildung 5.2 verdeutlicht werden. In diesem werden in dem Ausführungszeitraum τ_1 bis τ_3 die Bits σ_1 bis σ_3 sequentiell beschrieben und anschließend in dem Ausführungszeitraum τ_4 bis τ_6 wieder sequentiell gelesen. Die Anzahl Taktzyklen, die zwischen dem Schreiben und dem Lesen eines bestimmten Bits σ_i liegt, wird als Λ bezeichnet. Für das Bit σ_2 wird zwischen dem Schreiben zum Ausführungszeitpunkt τ_2 und dem Lesen zum Ausführungszeitpunkt τ_5 zu $\tau_5 - \tau_2 = \Lambda$ Ausführungszeitpunkten eine Fehlerinjektion durchgeführt.

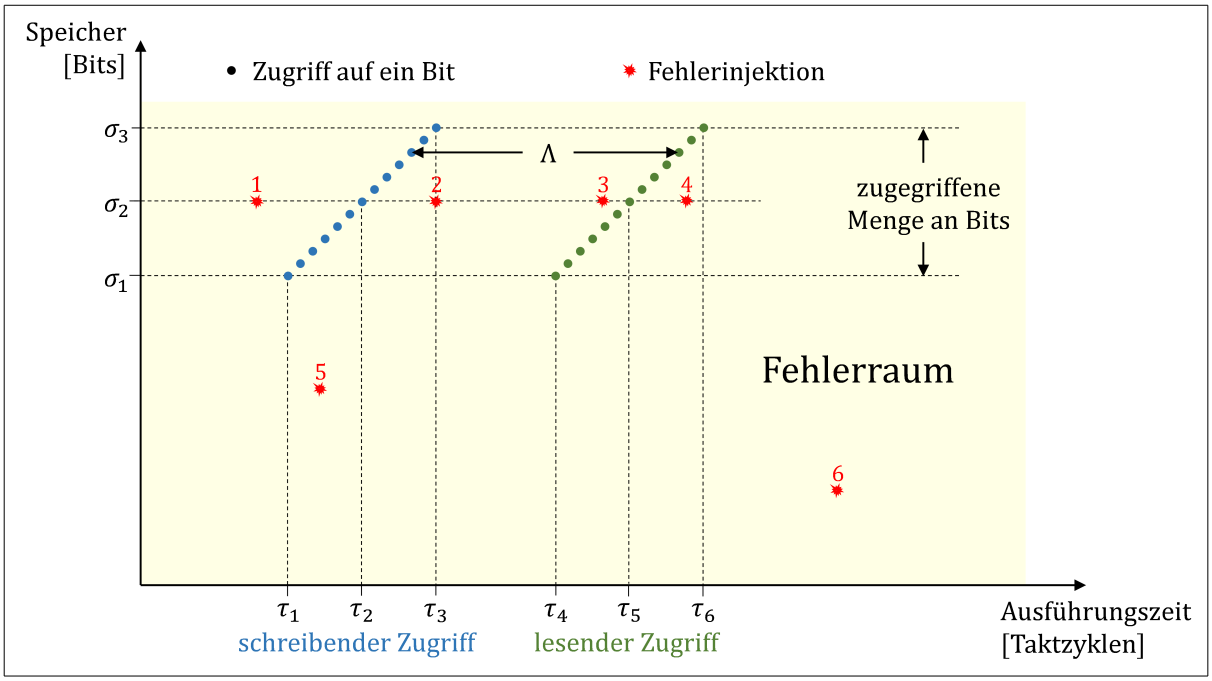

Abbildung 5.2: Fehlerraum mit schreibenden und lesenden Speicherzugriffen
Die Speicherzugriffe werden durch die farbigen Punkte und Fehlerinjektionen durch rote Explosionen dargestellt. Λ = Anzahl Taktzyklen zwischen dem Schreiben und dem Lesen eines Bits im Speicher.

In diesem Ausführungszeitraum ist es jedoch unerheblich, zu welchem Ausführungszeitpunkt die Fehlerinjektion stattfindet, denn diese Fehlerinjektion macht sich erst während des Lesens zum Ausführungszeitpunkt τ_5 bemerkbar. Für das Beispiel in der Abbildung 5.2 bedeutet dies, dass die Fehlerinjektionen 2 und 3 zu dem gleichen Ergebnis des Fehlerinjektions-Experimentes führen. Daher genügt es, in dem Bereich zwischen dem Schreiben und Lesen eines Bits nur ein einziges Fehlerinjektions-Experiment durchzuführen und das Ergebnis Λ-fach zu werten. Um die in den soeben genanten Optimierungen beschriebenen Situationen zu erkennen, sind detaillierte Informationen über das Ausführungsverhalten bezüglich der Speicherzugriffe der Anwendung nötig. FAIL* verwendet für die Ermittlung dieser Informationen das *Tracing*-Modul, welches bereits im Abschnitt 5.4.3 beschrieben wurde.

5.4 eCos

eCos(**e**mbedded **C**onfigurable **o**perating **s**ystem) ist ein kleines und hochkonfigurierbares Echtzeitbetriebssystem oder engl. *real time operating system*(RTOS) für eingebette Systeme[28, S. 7][29]. eCos ist, wie durch den Namen angedeutet, durch seine vielen Komponenten, die einzeln mit der durch das Werkzeug *eCos Configuration Tool* bereitgestellten grafischen Overfläche verwaltet werden können, hochgradig konfigurierbar. Dieses Werkzeug erstellt aus der ausgewählten Konfiguration entsprechende Header- und

Make-Dateien, sodass die gewünschten Komponenten während des Kompiliervorganges der eCos-Bibliothek mit Hilfe von bedingten Einsetzungen des Präprozessors[6, Kap. 16] statisch ausgewählt werden.
Die den Komponenten zugrunde liegende Programmiersprache ist nicht einheitlich, sondern ein Mix aus den Programmiersprachen C++, C, C-Präprozessor und Assembler. Der eCos-Kernel an sich ist allerdings in der Programmiersprache C++ implementiert[28, S. 7].

5.4.1 eCos-Kernel-Testsuite

Für den eCos-Kernel existieren eine Reihe von Tests, die die in der eCos-Bibliothek verfügbaren Komponenten gründlich überprüfen und deren korrekte Ausführung bestätigen[30]. Diese Tests werden als *eCos-Kernel-Testsuite* bezeichnet und testen die Funktionalität der Semaphoren, der Mutexe, der Threads, der Ausnahmen, des Schedulers sowie der Echtzeituhr[31, S. 8]. Die in dieser Arbeit verwendeteten Tests der eCos-Kernel-Testsuite werden im Folgenden auch als *eCos-Kernel-Tests* oder nur als *Tests* bezeichnet und sind in Tabelle 5.2 inklusive einer kurzen Beschreibung des jeweiligen Testobjektes aufgeführt.

Test	Beschreibung des Testobjektes	Anzahl Threads
bin_sem1	Binäre Semaphoren	2 Threads
bin_sem2	Verklemmung („Speisende Philosophen“)	15 Threads
bin_sem3	Timeout-Funktionalität von binären Semaphoren	2 Threads
cnt_sem1	Zählenden Semaphoren	2 Threads
flag1	Basis-Funktionalität und Synchronisation von Flags	3 Threads
kill	Thread-Funktionen `kill()` und `reinitalize()`	3 Threads
mbox1	Nachrichten-Box-Funktionen `put()` und `tryput()`	2 Threads
mqueue1	Nachrichtenwarteschlange (inkl. Semaphoren)	2 Threads
mutex1	Basis-Funktionalität von Mutexen	3 Threads
mutex2	Freigabe von Mutexen	4 Threads
mutex3	Prioritäts-Vererbung von Mutexen	7 Threads
release	Thread-Funktion `release()`	2 Threads
sched1	Basis-Funktionalität des Schedulers	2 Threads
sync2	Verschiedene Sperr-Mechanismen	4 Threads
sync3	Prioritäten und Prioritäts-Vererbung	3 Threads
thread0	Thread-Konstruktor und -Destruktor	1 Thread
thread1	Basic-Funktionalität von Threads	2 Threads
thread2	Scheduler- und Thread-Prioritäten	3 Threads

Tabelle 5.2: Die in dieser Arbeit verwendeten Tests der eCos-Kernel-Testsuite. Quellen: [31, S. 8] und [7, S. 83]

Diese Tests werden zusammen mit der eCos-Bibliothek im Folgenden für die Evaluation der angepassten Variante des AspectC++-Compilers verwendet.

5.4.2 Kompiliervorgang

Während des Kompiliervorganges werden die eCos-Bibliothek und anschließend die Tests der eCos-Kernel-Testsuite kompiliert. Zur Evaluation dieses Vorganges wird dieser für verschiedene Evaluations-Kombinationen, die durch die Merkmale *verwendete Variante des AspectC++Compilers* und *eingewebte Aspekte* charakterisiert werden, durchgeführt und vermessen.
Zum einen wurde die CPU-Zeit, während der sich der Prozess im Benutzer-Modus (*user*) befand und zum anderen die CPU-Zeit, während der sich der Prozess in dem Kernel-Modus (*sys*) befand, gemessen[4]. Zum anderen wurde die Größe des AspectC++-Repository (siehe Kapitel 3.1.2), sowie die Anzahl der Join-Points, an denen während des Kompiliervorganges ein Advice eingewebt wurde, ausgewertet.
Die Ergebnisse dieser Messungen sind in Tabelle 5.3 dargestellt. Die Beschreibung der Variante des AspectC++-Compilers wurde aus Platzgründen in die Tabellenbeschriftung verschoben und mittels der Symbole ▲, ■ und ★ verknüpft.

	Eingewebter Aspekt		**Kompilierzeit (Sekunden)**				**Repository-**	
	Aspekt	Anz.	*user*	*sys*	Gesamt		**Größe (Bytes)**	
▲	*keiner*	0	49,96	4,16	54,11		1.573.817	
★	*keiner*	0	76,46	9,42	85,89	+59%	4.533.237	+188%
■	*keiner*	0	70,32	7,05	77,37	+43%	4.120.882	+162%
■	Arithmetik-Valid.	805	81,30	9,17	90,48	+67%	4.291.685	+173%
■	Zeiger-Validierung	750	85,65	7,71	93,36	+73%	4.276.939	+172%
★	Zeiger-Validierung	2.867	107,67	9,72	117,39	+117%	5.135.472	+226%

Tabelle 5.3: Kompilierzeit für die eCos-Bibliothek und die in Tabelle 5.2 genannten eCos-Kernel-Tests für verschiedene Evaluations-Kombinationen aus AspectC++-Compiler-Variante und eingewebtem Aspekt. Zusätzlich ist die Anzahl der Join-Points, an denen ein Advice eingewebt wurde, sowie die Größe des AspectC++-Repositories angegeben. Die Prozentangaben zeigen die Veränderung bezüglich der ersten Kombination.
Variante des AspectC++-Compilers:
▲: Ohne Erweiterung für vordefinierte Operatoren (Version vom 5.12.2014)
■: Mit Erweiterung für explizite vordefinierte Operatoren aus Tabelle A.1
★: Mit Erweiterung für alle vordefinierten Operatoren aus Tabelle A.1 (inklusive implizite Dereferenzierungen)

Die sechs Evaluations-Kombinationen wurden so gewählt, dass für jede Kombination eine andere Kombination existiert, die sich nur durch ein Merkmal von dieser unterscheidet. Dadurch kann mit Hilfe eines Vergleichs ein Rückschluss auf den Einfluss der einzelnen Merkmale gezogen werden. Die erste Kombination mit der bisherigen Variante des AspectC++-Compilers und keinen eingewebten Aspekten wird im Folgenden als *Ausgangs-Kombination* oder *Ausgangs-Evaluations-Kombination* bezeichnet.

[4]Linux `time`-Befehl; Testsystem mit acht Intel® Xeon® E5-4640 CPUs mit jeweils acht Kernen (2.40 GHz pro Kern) und 252,37 GiB Arbeitsspeicher; keine Nebenläufigkeit während des Kompiliervorganges; zehn Wiederholungen und anschließende Bildung des Mittelwertes

Aus den Ergebnissen folgt, dass bereits durch die Verwendung der um explizite und implizite vordefinierte Operatoren erweiterten Variante des AspectC++-Compilers ohne eingewebte Aspekte eine 59 % längere Kompilierzeit vorhanden ist. Dies kann durch die um 162 % deutlich angestiegene Größe des AspectC++-Repositories erklärt werden. Da die von den Join-Points unabhängigen Daten[5] mit einer Größe von rund 53 KiB nur ca. 1 % der Größe des AspectC++-Repositories ausmachen, kommt die gestiegene Größe des AspectC++-Repository wiederum durch die zusätzlichen Join-Points an Aufrufen von vordefinierten Operatoren zustande.
So existieren mit der bisherigen Variante des AspectC++-Compilers 3349 Join-Points an Aufrufen von (Member-)Funktionen, wohingegen mit der erweiterten Variante 16634 - das entspricht beinahe einer Verfünffachung - Join-Points an Aufrufen von (Member-)Funktionen und vordefinierten Operatoren existieren[6].
Wenn der in Abschnitt 5.2 erläuterte Arithmetik-Validierungs-Aspekt eingewebt wird, ergeben sich 805 zusätzliche Join-Points.
Falls stattdessen der in Abschnitt 5.1 beschriebene Zeiger-Validierungs-Aspekt an den 750 Aufrufen einer expliziten Dereferenzierung eingewebt wird, steigt die Kompilierzeit im Vergleich zur Kompilierzeit ohne den eingewebten Aspekt um weitere 21 % und im Vergleich zur Ausgangs-Kombination um 73 %. Der Zuwachs der Größe des AspectC++-Repository um annähernd 152 KiB ensteht, da nun zusätzlich gespeichert werden muss, an welchen Join-Points der Advice des Aspektes eingewebt werden muss.
Infolge der Verwendung der Variante des AspectC++-Compilers, die auch zusätzlich implizite Dereferenzierungen unterstützt, entstehen 2117 weitere Join-Points, an denen ein impliziter Dereferenzierungs-Operator aufgerufen und der Zeiger-Validierungs-Aspekt eingewebt wird. Im Vergleich der Anzahl der Join-Points an Aufrufen ergibt sich für die drei Varianten des AspectC++-Compilers demzufolge das Diagramm in Abbildung 5.3.

[5] Diese Daten umfassen Informationen zu den Übersetzungseinheiten und Header-Dateien des C++-Quelltexts.

[6] In dem Quelltext der eCos-Bibliothek sowie der verwendeten eCos-Kernel-Tests wird demzufolge 13285-mal ein (in dieser Arbeit betrachteter expliziter) vordefinierter Operator aufgerufen.

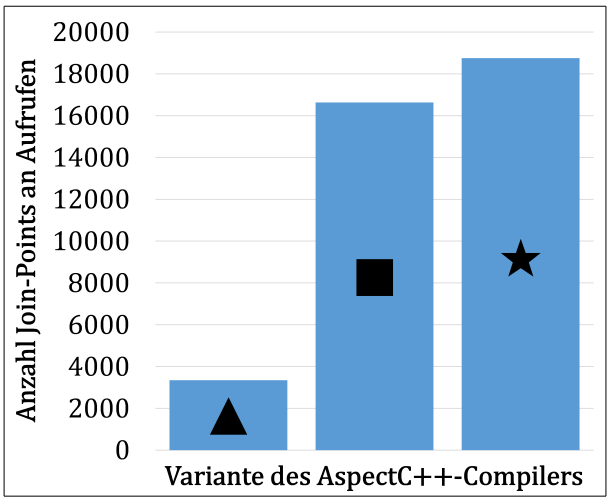

Abbildung 5.3: Anzahl der Join-Points an Aufrufen in dem AspectC++-Repository
Variante des AspectC++-Compilers:
▲: Ohne Erweiterung für vordefinierte Operatoren (Version vom 5.12.2014)
■: Mit Erweiterung für explizite vordefinierte Operatoren aus Tabelle A.1
★: Mit Erweiterung für alle vordefinierten Operatoren aus Tabelle A.1 (inklusive implizite Dereferenzierungen)

Die Verwendung der Variante des AspectC++-Compilers, die auch implizite Dereferenzierungen unterstützt, macht sich zusätzlich sowohl in der Kompilierzeit, die sich um weitere 25 % verlängert, als auch am AspectC++-Repository, das sich um weitere 20 % vergrößert, bemerkbar. Insgesamt entsteht durch die Verwendung der erweiterten Variante des AspectC++-Compilers im Vergleich zur bisherigen Variante ein erheblicher Overhead von - im Fall der Repository-Größe - um bis zu 226 %. Dieser Overhead geht allerdings mit einer über 500 % größeren Anzahl von Join-Points und dem zusätzlich eingewebten Zeiger-Validierungs-Aspekt einher, durch den eine Verbesserung der Fehlertoleranz möglich ist.

5.4.3 Größe und Laufzeit-Faktoren

Die Größe der ausführbaren *.elf*-Datei der einzelnen Tests ist besonders in Systemen mit geringem Speicherplatz ein entscheidender Faktor. Daher wurde diese für die folgenden drei Evalutions-Kombinationen gemessen[7].
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte

②: Erweiterte Variante des AspectC++-Compilers und eingewebter Zeiger-Validierungs-Aspekt an expliziten Dereferenzierungen (Dereferenzierungs-Operator)

③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und

[7]Linux `size`-Befehl

expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

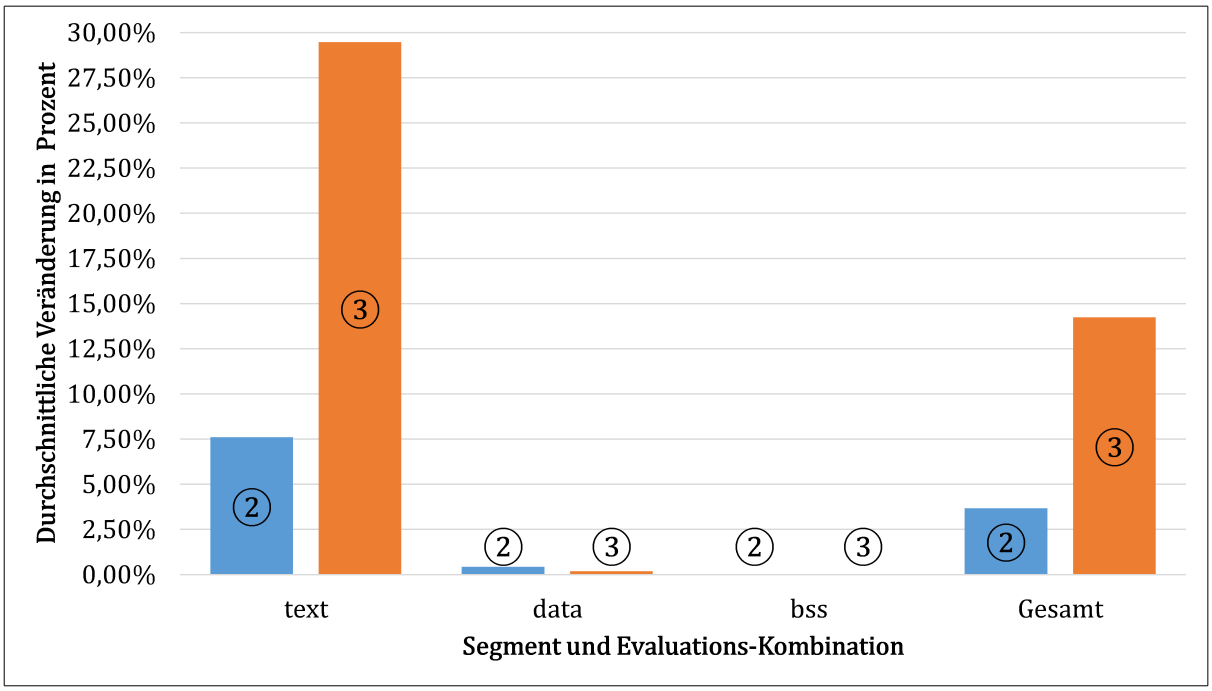

Abbildung 5.4: Durchschnittliche Veränderung der Größen der Segmente der *.elf*-Datei im Vergleich zur Ausgangs-Kombination ①
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
②: Erweiterte Variante des AspectC++-Compilers und eingewebter Zeiger-Validierungs-Aspekt an expliziten Dereferenzierungen (Dereferenzierungs-Operator)
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

Die Abbildung 5.4 zeigt die durchschnittliche Veränderung der gemessenen Größen[8] für die Evaluations-Kombinationen ② und ③ im Vergleich zur Ausgangs-Kombination ①. Es wird deutlich, dass das *text*-Segment ausschlaggebend für die Veränderung der Gesamtgröße ist. Dies liegt daran, dass die (Member-)Funktionen durch den zusätzlich eingewebten Advice-Quelltext aus mehr Anweisungen bestehen und somit auch mehr Speicherplatz benötigen.

Um auch den Einfluss auf die zur Laufzeit verbrauchten Ressourcen messen zu können, wird das Tracing-Plugin des in Abschnitt 5.3.1 vorgestellten FAIL* verwendet. Dieses

[8] In den Tabellen A.2 bis A.4 des Anhangs befinden sich die einzelnen Ergebnissen für jeden Test und jede Evaluations-Kombination.

Plugin verwendet den Backend-Simulator von FAIL*, um eine Anwendung - in diesem Fall ein eCos-Kernel-Test - auszuführen und währendessen zu jeder Ausführungszeit die aktuelle Instruktion und die Speicherzugriffe zu protokollieren. Aus diesen protokollierten Daten lassen sich mit einem weiteren FAIL*-Plugin Angaben bezüglich der Gesamtanzahl der Instruktionen, der Taktzyklen, der lesenden und schreibenden Speicherzugriffe sowie der gelesenen und geschriebenen Speichermenge ermitteln. Die durch dieses Vorgehen ermittelten Werte der Laufzeit-Faktoren befinden sich in den Tabellen A.2 bis A.4, die auf Grund ihrer Größe im Anhang zu finden sind.
Im Fall der Evaluations-Kombination ② unterscheiden sich die Werte der einzelnen Laufzeit-Faktoren der eCos-Kernel-Tests um jeweils weniger als 1 % von den Werten der Ausgangs-Kombination ①. Die Werte der Laufzeit-Faktoren der Evaluations-Kombination ③ unterscheiden sich im Gegensatz dazu in Extremfällen um rund -5 % oder 7 % von der den Werten der Ausgangs-Kombination. Diese Extremfälle treten für die eCos-Kernel-Tests *release* und *bin_sem2* auf und sind inklusive der anderen Laufzeit-Faktoren in dem in Abbildung 5.5 dargestellten Diagramm visualisiert. Der durchschnittliche Laufzeit-Overhead für alle eCos-Kernel-Tests liegt allerdings bei nur ungefähr 0,7 %.

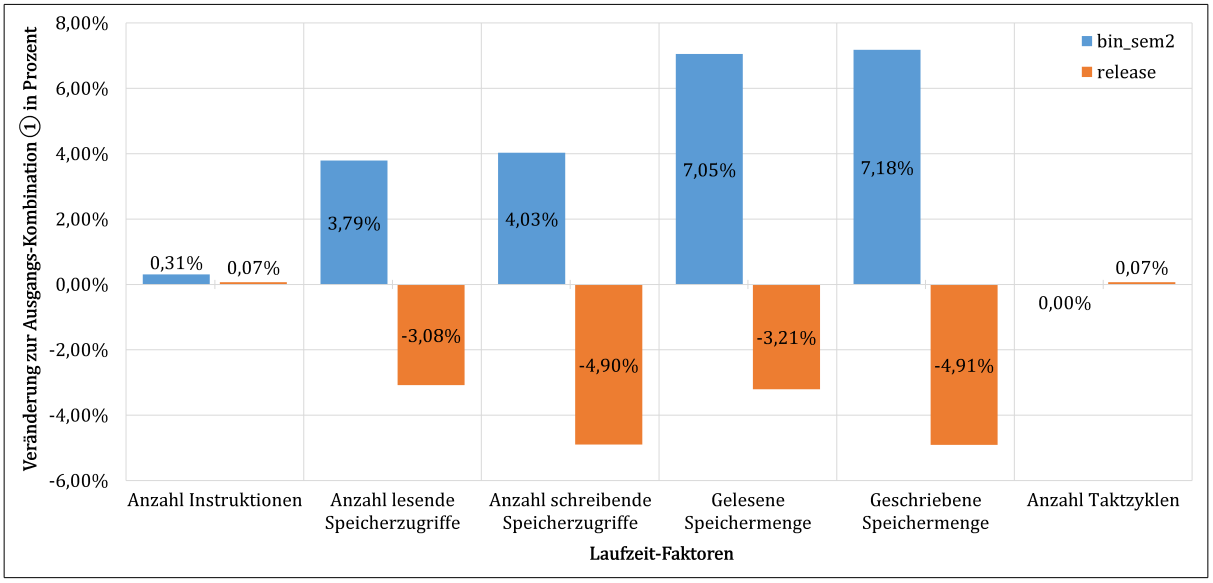

Abbildung 5.5: Veränderung der Werte der Laufzeit-Faktoren der eCos-Kernel-Tests *release* und *bin_sem2* in der Evaluations-Kombination ③ im Vergleich zur Ausgangs-Kombination ①
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

Durch das Einweben des Zeiger-Validierungs-Aspekt haben sich im Fall des eCos-Kernel-Testes *release* sowohl die Anzahl der Speicherzugriffe als auch die zugegriffene Speichermenge reduziert. Da diese Verbesserungen allerdings mit einer Verschlech-

terung der benötigten Taktzyklen einhergeht, kann insgesamt und unabhängig von dem Anwendungsfall nicht von einer Verbesserung gesprochen werden. Die unterschiedlichen Abweichungen in diesen zwei Fällen von der Ausgangs-Kombination sind höchstwahrscheinlich durch unterschiedliche Optimierungs-Abwägungen des verwendeten *g++*-Compilers[9] entstanden. So war es für den Compiler wahrscheinlich optimal, die durch den Zeiger-Validierungs-Aspekt überprüften Zeiger auf Kosten von Taktzyklen in CPU-Registern statt im Speicher zwischenzuspeichern. Weitergehende Untersuchungen wurden allerdings aus Zeitgründen unterlassen.
Ein gegensätzliches Ergebnis zeigt sich in den Werten des eCos-Kernel-Testes *bin_sem2*, in denen keine Verbesserung erkannt werden kann. Die, vor allem bezüglich des Speichers gestiegenen, Werte sind durch die die Große Anzahl von 15 Threads zu erklären. Denn während des Kontextwechsels zwischen den Threads müssen z.B. CPU-Register in einem langsameren Speicher, wie beispielsweise der Arbeitsspeicher oder ein Cache, gesichert und vor der weiteren Ausführung des Threads wiederhergestellt werden. Da durch den eingewebten Zeiger-Validierungs-Aspekt die in CPU-Registern vorhandenen Zeiger aufgrund der Validierung länger dort verbleiben, müssen diese häufiger während eines Kontextwechsels in einen langsameren Speicher ausgelagert und später von dort aus wieder eingelagert werden.

5.4.4 Fehlertoleranz

Zur Evaluation der eCos-Kernel-Tests wird das in Abschnitt 5.3.1 eingeführte Fehlerinjektionssystem FAIL* mit dem *Bochs*-Simulator als Backend verwendet. Als Betriebssystem des Simulators wird eCos eingesetzt. Außerdem werden die zur Fehlerinjektion verwendeten Einzel-Bit-Flips nur in dem *data*-Segment des jeweiligen eCos-Kernel-Tests durchgeführt.

Durch die großen Fehlerräume und die daraus resultierende große Anzahl an Fehlerinjektions-Experimenten nimmt die Fehlertoleranz-Evaluation eines eCos-Kernel-Tests einige Zeit in Anspruch. Aus diesem Grund kann diese Evaluation nicht für jeden Test und jede Evaluations-Kombination durchgeführt werden. Deswegen wurden durch drei Fehlerinjektions-Kampagnen erste Werte für die drei bereits im Abschnitt 5.4.3 vorgestellten Evalutions-Kombinationen und den eCos-Kernel-Tests *sched1* bestimmt, anhand derer über das weitere Vorgehen entschieden werden soll. Der Test *sched1* wurde gewählt, weil dessen Fehlerraum durch die im Vergleich mit den anderen Tests geringe gelesene Speichermenge und Anzahl Taktzyklen relativ klein ist und sich somit schnell Ergebnisse ergeben.

[9] Version 4.7.2

	Anzahl fehlerhafte Ergebnisse	
①	1.075.060.654	
②	1.037.898.940	-3,46%
③	817.220.061	-23,98%

Tabelle 5.4: Anzahl der fehlerhaften Ergebnisse der Fehlerinjektions-Experimente für den eCos-Kernel-Tests *sched1* und verschiedene Evaluations-Kombinationen
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
②: Erweiterte Variante des AspectC++-Compilers und eingewebter Zeiger-Validierungs-Aspekt an expliziten Dereferenzierungen (Dereferenzierungs-Operator)
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

Das in Tabelle 5.4 aufgeführte Resultat zeigt, dass das Einweben des Zeiger-Validierungs-Aspekt an Aufrufen des expliziten Dereferenzierungs-Operators (②) eine Wirkung zeigt und die Fehlertoleranz dadurch im Vergleich zur Ausgangs-Kombination ① um ungefähr 3,5% gestiegen ist. Wie im Abschnitt 5.4.3 bereits beschrieben ist die Größe des *.elf*-Datei des Tests währendessen um 4 % gestiegenen und ein Laufzeit-Overhead mit weniger als 1 % quasi nicht vorhanden.
Wird der Zeiger-Validierungs-Aspekt zusätzlich an Aufrufen von den in Tabelle A.1 angegebenen impliziten Dereferenzierungs-Operatoren eingewebt(③), steigt die Fehlertoleranz insgesamt um etwa 24 % im Vergleich zur Ausgangs-Kombination. Diese Verbesserung geht mit einem um ungefähr 0,5 % gestiegenen Laufzeit-Overhead und einer um rund 15 % größeren ausführbaren Datei einher.
Da die Evaluations-Kombination ③ durch die zuätzlichen impliziten Dereferenzierungen eine höhere Verbesserung der Fehlertoleranz bietet, wird diese im Folgenden bezüglich der Fehlertoleranz im Vergleich zur Ausgangs-Kombination genauer untersucht.
Dazu werden für weitere Tests der eCos-Kernel-Testsuite Fehlerinjektions-Kampagnen durchgeführt und die Ergebnisse der Fehlerinjektions-Experimente bezüglich ihrer Art (*TRAP*, *TIMEOUT*, *SDC*, *OK*, *DETECTED*) differenziert, sodass die in Tabelle A.5[10] enthaltenen Werte bestimmt werden. Falls diese Werte als Säulendiagramm interpretiert werden, ergibt sich die Abbildung 5.6.
Jeweils zwei hintereinanderfolgende Säulen gehören zu dem gleich eCos-Kernel-Test. Die jeweils erste dieser beiden Säulen entsteht aus den Werten der Ausgangs-Evaluations-Kombination ①, während die Zweite der erweiterten Variante des AspectC++-Compilers mit an Aufrufen von expliziten und impliziten Dereferenzierungs-Operatoren Zeiger-Validierungs-Aspekten (③) entspricht. Die in roten und orangen Farbtönen gefüllten Bereiche bezeichnen die fehlerhafte Ergebnisse. Wie durch einen jeweiligen Vergleich der Säulen ① und ③ zu erkennen ist, sinkt die Anzahl der fehlerhaften Ergebnisse vor allem durch die, durch das Einweben des Zeiger-Validierungs-Aspektes geringere Anzahl

[10]Die Tabelle befindet sich aus Platzgründen im Anhang.

von Abstürzen (*TRAP*). Währendessen steigt die Anzahl der detektierten Fehler um ungefähr die Menge der gesunkenen fehlerhaften Ergebnisse an, sodass die Anzahl der fehlerfreien Ergebnisse (*OK*) bis auf kleine Schwankungen stagniert.

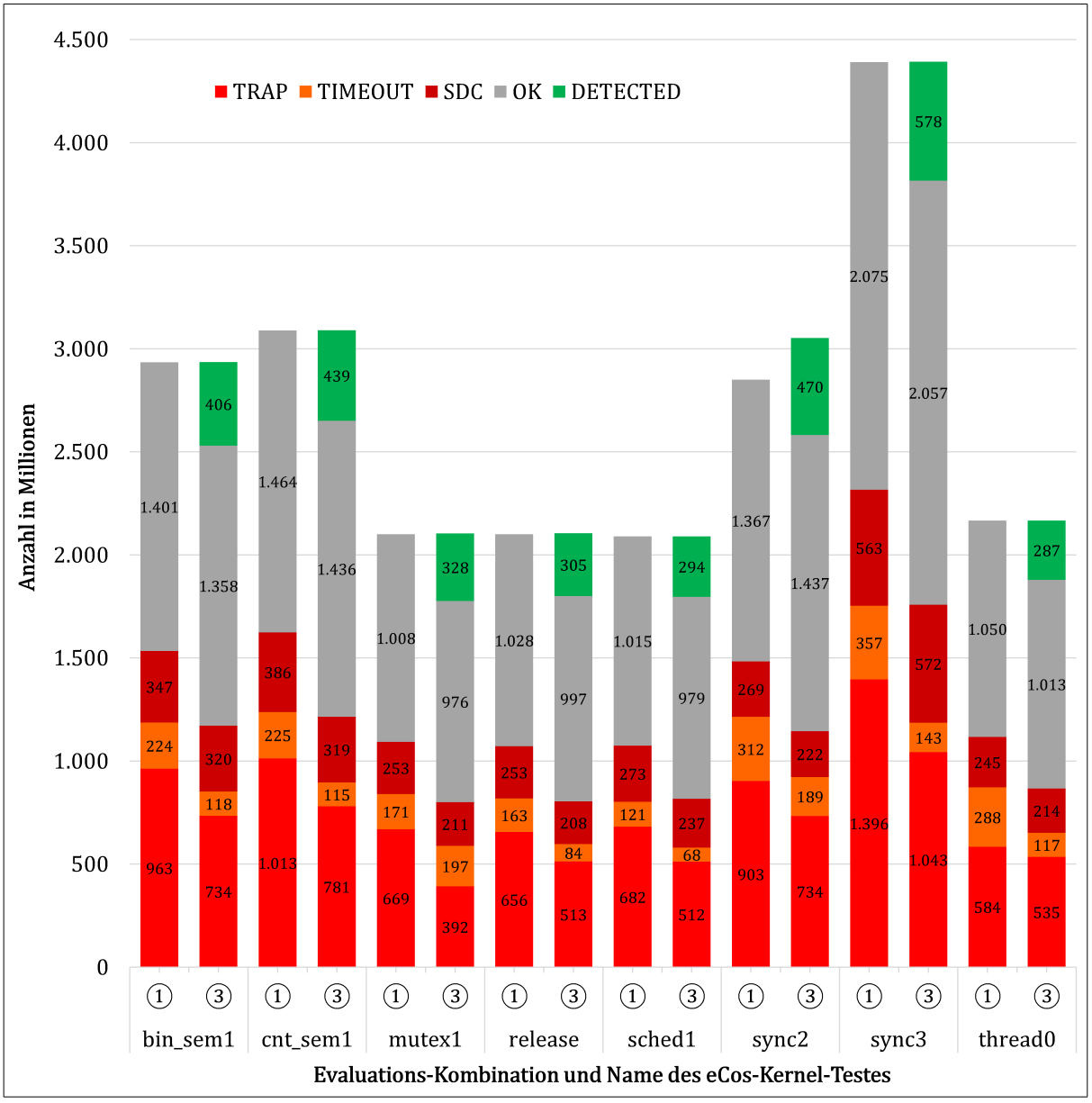

Abbildung 5.6: Vergleich der Ergebnisse der Fehlertoleranz-Benchmarks für die Tests der eCos-Kernel-Testsuite
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

In Tabelle 5.5 ist zusätzlich die Veränderung der aggregierten fehlerhaften Ergebnissen

	Test	**Größe des Fehlerraumes / Summe Fehlerinjektionen**		**Summe fehlerhafte Ergebnisse**	
①	bin_sem1	2.934.468.064		1.533.580.103	
	bin_sem3	827.926.682.048		425.523.608.317	
	cnt_sem1	3.088.810.592		1.624.357.558	
	mutex1	2.101.017.712		1.093.075.497	
	release	2.100.303.584		1.072.008.823	
	sched1	2.090.234.560		1.075.060.654	
	sync2	2.850.154.456		1.483.533.741	
	sync3	4.391.265.800		2.316.310.367	
	thread0	2.166.429.424		1.116.886.975	
	thread1	55.064.961.376		25.579.061.797	
③	bin_sem1	2.935.325.496	+0,03%	1.171.980.321	-23,58%
	bin_sem3	827.919.445.880	±0,00%	227.613.525.767	-46,51%
	cnt_sem1	3.090.137.816	+0,04%	1.214.746.889	-25,22%
	mutex1	2.104.123.000	+0,15%	800.156.043	-26,80%
	release	2.105.642.768	+0,25%	804.428.023	-24,96%
	sched1	2.090.393.024	+0,01%	817.220.061	-23,98%
	sync2	3.052.366.120	+7,09%	1.145.063.524	-22,82%
	sync3	4.393.438.896	+0,05%	1.758.400.899	-24,09%
	thread0	2.166.459.128	±0,00%	866.435.895	-22,42%
	thread1	55.065.376.296	±0,00%	13.489.437.211	-47,26%

Tabelle 5.5: Größe des Fehlerraumes und und die aggregierten fehlerhaften Ergebnisse für die eCos-Kernel-Tests
Außerdem sind die Veränderungen zu der Kombination ① angegeben.
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

(*TRAP*, *TIMEOUT* und *SDC*) der Evaluations-Kombinationen ③ zur Ausgangs-Kombination zu sehen. Die daraus abzulesende Verbesserung der Fehlertoleranz liegt für die meisten Tests bei rund 25 % und erreicht für die Tests *bin_sem3* und *thread1* sogar Werte um die 45 %. Im Durchschnitt aller Tests ergibt sich daher eine rund 29 %-ige Verbesserung der Fehlertoleranz, während der durchschnittliche Laufzeit-Overhead, wie in Abschnitt 5.4.3 gezeigt, bei nur bei ca. 0,7 % liegt. Tabelle 5.5 zeigt außerdem die Summe der Fehlerinjektionen, die, wie in Abschnitt 5.3 gezeigt, der Größe des Fehlerraumes entspricht. Sowohl anhand der Werte in dieser Tabelle als auch an dem oben gezeigten Diagramm kann erkannt werden, dass das Einweben des Zeiger-Validierungs-Aspektes - mit Ausnahme des Testes *sync2* - nahezu keinen Einfluss auf die Größe des Fehlerraumes hat. Der *ecos*-Kernel-Test *sync2* stellt hier eine Ausnahme dar, denn

durch das Einweben des Zeiger-Validierungs-Aspektes ist der Fehleraum um rund 7 % gewachsen. Dies liegt an der im Vergleich zu den anderen eCos-Kernel-Tests deutlich stärker gewachsenen Anzahl der Taktzyklen zwischen der Ausgangs-Kombination und der Evaluations-Kombination ③. Deutlich wird dies vor allem durch das in Abbildung 5.7 gezeigte Diagramm. Warum gerade bei diesem Test die Anzahl der Taktzyklen im

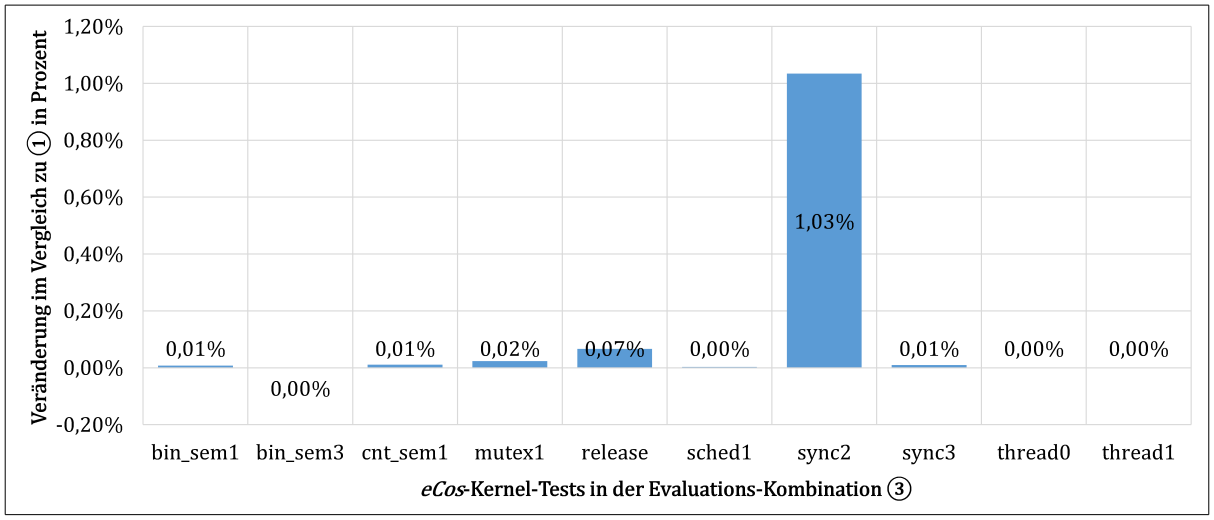

Abbildung 5.7: Veränderung der Taktzyklen der Evalutions-Kombination ③ im Vergleich zur Ausgangs-Kombination ①
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

Vergleich mit den anderen Tests so stark angestiegen ist, konnte im Rahmen dieser Arbeit nicht geklärt werden.

5.5 Qt-Beispiele und Puma-Bibliothek

Zusätzlich zu den eCos-Kernel-Tests werden in diesen Abschnitt weitere Anwendungen bezüglich des Kompiliervorganges evaluiert. Dazu wird allerdings nur die erweiterte Variante des AspectC++-Compilers ohne Dereferenzierungen mit der bisherigen Variante verglichen, da die impliziten Dereferenzierungen zum Zeitpunkt der Messungen noch nicht implementiert waren.

Qt[32] ist eine C++-Bibliothek, durch die grafische Benutzeroberflächen plattformübergreifend realisiert werden können. Die *Qt-Beispiele* sind eine Sammlung von 226 kleinen Anwendungen, die die Funktionalitäten dieser Qt-Bibliothek präsentieren.
Anhand dieser Beispiele soll die Kompilierzeit und die Größe der ausführbaren Dateien evaluiert werden. Dazu wird ein simpler Aspekt, in dem durch einen Advice die Signatur

des jeweiligen Join-Points auf der Standardausgabe ausgegeben wird, an Join-Points von verschiedenenen Pointcuts eingewebt, sodass das Ergebnis in Tabelle 5.6 entsteht.

	Join-Points mit eingewebtem Aspekt	**Kompilierzeit (Sekunden)**		**Gesamt-Größe ausführbare Dateien (Bytes)**		**Gesamt-Größe Repository (Bytes)**	
▲	keine	2.564		13.423.018		16.405.399	
■	keine	2.571	+0,27%	13.423.018	±0%	20.507.649	+25%
■	globale Operatoren	2.603	+1,52%	13.811.633	+3%	23.056.722	+40%
▲	alle	2.671	+4,17%	15.229.088	+13%	23.605.011	+43%
■	alle	2.813	+9,71%	15.473.826	+15%	29.257.428	+78%

Tabelle 5.6: Kompilierzeit, aggregierte Repository-Größe aller Qt-Beispiele sowie Gesamt-Größe aller ausführbaren Dateien der Qt-Beispiele
Variante des AspectC++-Compilers:
▲: Ohne Erweiterung für vordefinierte Operatoren (Version vom 5.12.2014)
■: Mit Erweiterung für explizite vordefinierte Operatoren aus Tabelle A.1

Absolut betrachtet ist die Kompilierzeit der Qt-Beispiele durch die Verwendung der erweiterten Variante des AspectC++-Compilers (ohne implizite Dereferenzierungen) und durch Einweben des Aspektes über vier CPU-Minuten gestiegen. Allerdings ist dies in Anbetracht der auch mit der bisherigen Variante des AspectC++-Compilers sehr langen Kompilierzeit von ungefähr 42 CPU-Minuten nur eine relative Verschlechterung von 9,7 %.
Wie erwartet, steigt die Gesamt-Größe der ausführbaren Dateien ohne den eingewebten Aspekt nicht, da in diesem Fall die zusätzlichen Join-Points nicht verwendet werden. Mit eingewebten Aspekten an allen Join-Points entsteht allerdings eine Vergrößerung um bis zu 15 %.
In der Größe des AspectC++-Repositories machen sich die zusätzlichen Join-Points bereits ohne eingewebte Aspekte deutlich bemerkbar. Ein höherer Größen-Overhead entsteht aber, wenn außerdem an den zusätzlichen Join-Points der oben beschriebene Aspekt eingewebt wird, da in diesem Fall in dem Repository zusätzlich gespeichert werden muss, an welchen Join-Points der Advice des Aspektes eingewebt werden soll.

Für die Puma-Bibliothek, die das alternative Frontend von AspectC++ beinhaltet, wurden aus Zeitgründen nur Messungen ohne eingewebte Aspekte durchgeführt. Die Verwendung der erweiterten Variante des AspectC++-Compilers bewirkt für diese Bibliothek einen Anstieg der Kompilierzeit um ca. 2 % von 470 CPU-Sekunden auf 480 CPU-Sekunden. Außerdem steigt auch die Größe des AspectC++-Repositories von 9 MiB auf 11,6 MiB an, sodass von einer Verschlechterung um rund 30 % gesprochen werden kann.

6 Fazit und Ausblick

Ein Ziel dieser Arbeit war die Realisierung von Advice für vordefinierte Operatoren. Die Kern-Anforderung, die darin bestand, Advice für den Dereferenzierungs-Operator sowie die arithmetischen Operatoren zu ermöglichen, wurde bei weitem übertroffen. Auch das optionale Ziel, für alle in dem Kapitel 13.6 des C++-Standards[6] beschriebenen vordefinierten Operatoren das Einweben von Advice-Quelltext zu implementieren, wurde erreicht. Ingesamt wurden darüber hinaus zusammen mit dem Zeiger-zu-Member Operatoren `.*` und den beiden impliziten Dereferenzierungen Advice für 46 explizite und implizite vordefinierte Operatoren implementiert. Eine Übersicht dieser Operatoren befindet sich in der Tabelle A.1 des Anhangs.

Allerdings konnten nicht alle dieser Operatoren vollständig realisiert werden. Wie bereits im Kapitel 4.3 erläutert wurde, können an bestimmten Aufrufen von einigen vordefinierten Operatoren keine Advice-Quelltexte eingewebt werden.
In einigen Fällen liegt dies an den Restriktionen der Programmiersprache C++. So gibt es Typen, die nur implizit und nicht zur Deklaration einer Variablen verwendet werden. Dies betrifft vor allem die vordefinierten Operatoren, die ein Objekt mit dem nur implizit vorhandenen Typ „*Member*“ oder „*Member-Funktion*“ zwischenspeichern müssen. Andere Restriktionen werden durch neue Sprachelemente gelockert, wodurch in zukünftigen Versionen von AspectC++ beispielsweise die momentan durch die Verwendung von Lambda-Ausdrücken noch optionale links-nach-rechts-Auswertung vollständig unterstützt werden kann. Auch die Implementierung von Advice in konstanten Ausdrücken ist durch die Verwendung des *constexpr*-Spezifizierers in Zukunft zumindest teilweise möglich.
In dem Fall des impliziten Dereferenzierungs-Operators stellen aber auch interne Einschränkungen des AspectC++-Compilers noch Probleme dar. Hier sollte noch eine Funktionalität implementiert werden, durch die ein direktes Einfügen von einem Quelltext so ermöglicht wird, dass folgende Löschvorgänge diesen berücksichtigen.
Allgemein sollte der Benutzer des AspectC++-Compilers allerdings - falls er für einen Join-Point mit Einschränkungen einen Advice definiert - über die entsprechenden Einschränkungen durch einen Hinweis oder eine Warnung informiert werden. Die dafür zusätzlichen Änderungen an den Klassen der Pointcut-Ausdrücke waren jedoch aus Zeitgründen leider nicht mehr möglich, sodass diese Anmerkung vor allem als Ausblick gedacht ist.

Das andere Ziel dieser Arbeit war es, zu evaluieren, ob die Realisierung der vordefinierten Operatoren tatsächlich eine Möglichkeit zur Erkennung der Soft-Errors bietet. Weiterhin sollte der Einfluss dieser Erkennungen auf die Verbesserung der Fehlertoleranz sowie auf den Overhead zur Laufzeit untersucht werden. Dazu wurden die Soft-Errors mit Hilfe des Fehlerinjektionssystems FAIL* in den eCos-Kernel-Tests simuliert und anschlie-

ßend das Verhalten dieser Tests beobachtet. Wie die Evaluation im Kapitel 5 gezeigt hat, ist durch das Einweben des Zeiger-Validierungs-Aspektes in den eCos-Kernel-Tests eine Verbesserung der Fehlertoleranz um bis zu 47 % möglich. Die durchschnittliche Verbesserung der Fehlertoleranz aller eCos-Kernel-Tests in der Evaluations-Kombination ③[1] liegt bei immer noch ausgesprochen guten 29 %. Vor allem im Vergleich mit dem nur maximal rund 7 %-igen und durchschnittlichem 0,7 %-igen (Evaluations-Kombination ③[1]) Laufzeit-Overhead sind diese Verbesserungen sehr zufriedenstellend, sodass die Anforderung an einen geringen Laufzeit-Overhead als erfüllt angesehen werden kann.
Einen Nachteil stellen natürlich die um bis zu 117 % höheren Kompilierzeiten der eCos-Kernel-Tests dar. Auch ohne eingewebte Aspekte stieg die Kompilierzeit dieser Tests um bis zu 59 %, das entspricht immerhin einer halben Minute. Die Kompilierzeit der Qt-Beispiele stieg absolut betrachtet sogar um über vier Minuten. Allerdings ist das in Anbetracht der auch mit der bisherigen Variante des AspectC++-Compilers sehr langen Kompilierzeiten von ungefähr 42 Minuten nur eine relative Verschlechterung von 9,7 %. Diese Laufzeit-Overheads waren jedoch aufgrund der Philosophie von C++ und AspectC++, möglichst viele Auswertungen und Berechnungen bereits statisch zur Kompilierzeit durchzuführen, zu erwarten.
Um den Kompilierzeit-Overhead von bis zu 59 % in Fällen, in denen die zusätzlichen Join-Points an den vordefinierten Operatoren nicht benötigt werden, zu vermeiden, sollte es eine Möglichkeit geben diese Join-Points während des Kompiliervorganges zu ignorieren. Dazu könnten vor dem Parsen des C++-Quelltextes die Pointcut-Ausdrücke daraufhin untersucht werden, ob diese keine Aufrufe von Operatoren beschreiben und somit die Join-Points an vordefinierten Operatoren nicht benötigen. Eine andere Option wäre die Implementierung einer neuen Pointcut-Funktion, mit der die Join-Points an vordefinierten Operatoren explizit aktiviert werden müssen. Dies hätte den Vorteil, dass bestehende Anwendungen durch die Verwendung der erweiterten Variante des AspectC++-Compilers nicht verändert würden.

Eine weitere Forschungsmöglichkeit könnte zusätzlich die Evaluation des Arithmetik-Validierungs-Aspektes bezüglich Fehlern in einer Berechnungseinheit, wie der ALU, darstellen.

Insgesamt entstand in dieser Arbeit eine Methode, mit der Soft-Errors auch in bestehenden Anwendungen komfortabel durch die Verwendung von AspectC++ erkannt werden können. Dadurch wird ein Beitrag zu einer immer vollständigeren Fehlertoleranz, insbesondere von eingebetten Systemen, geliefert.

[1] ③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

A Anhang

```
1  // Beispiel-Klasse:
2  class ExampleClass {
3    public:
4      int examplevalue;
5      ExampleClass(int value) {
6        examplevalue = value;
7      }
8  };
9
10 // Globale Überladung des binären Plus-Operators:
11 ExampleClass operator +(ExampleClass lhs, ExampleClass rhs) {
12   return ExampleClass(lhs.examplevalue + rhs.examplevalue);
13 }
14
15 int main() {
16   // Lokale Variablen definieren und initialisieren:
17   ExampleClass a = ExampleClass(1), b = ExampleClass(2);
18   int d = 1, e = 2;
19   // Aufruf des überladenen binären Plus-Operators:
20   ExampleClass c = a + b;
21   // Aufruf des vordefinierten binären Plus-Operators:
22   int f = d + e;
23 }
24
25 // Beispiel-Aspekt:
26 aspect ExampleAspect {
27   advice call("% ...::operator +(%)" || "% operator +(%, %)") : ↩
       before() {
28     // Advice-Quelltext: In diesem Beispiel wird die
29     // Signatur des Join-Points ausgegeben:
30     cout << "Signatur: \"" << tjp->signature() << "\"" << endl;
31   }
32 };
```

Quelltext A.1: Beispiel-Quelltext zur Demonstrierung des Webens an Join-Points der überladenen und vordefinierten Operatoren. Das Ergebnis ist im Quelltext 3.3 zu sehen.

```
1  #include "PtrType.h" // TypeTraits
2
3  // Symbole für die Segment-Grenzen:
4  extern char _stext, _etext, _end;
5  // Externe Funktion zur Fehlerbehandlung:
6  void ecc_panic() __attribute__ ((noreturn, noinline));
7
8  // Aspekt, der Zeiger validiert bevor diese dereferenziert werden:
9  aspect PointerValidator {
10   // Pointcut-Ausdruck des Advice beschreibt alle Join-Points an
11   // Aufrufen von vordefinierten Dereferenzierungs-Operatoren.
12   // Validierung findet vor(before) der Dereferenzierung statt:
13   advice call("% operator *(%*)") : before() {
14     // Zeiger-Typ bestimmen:
15     typedef JoinPoint::template Arg<0>::ReferredType PointerType;
16     // Zeiger zwischenspeichern;
17     const PointerType pointer = *(tjp->arg<0>());
18     // Zeiger-Kategorie mittels TypeTraits[7] ermitteln:
19     const TypeTraits::result_type pointer_category = TypeTraits::↩
         pointerType<PointerType>::result;
20     // Adressen der Segment-Grenzen zum Typ des Zeigers
21     // casten und zwischenspeichern:
22     const PointerType text_start_addr = (PointerType)&_stext;
23     const PointerType text_end_addr = (PointerType)&_etext;
24     const PointerType program_end_addr = (PointerType)&_end;
25     // Unterscheidung anhand der Zeiger-Kategorie:
26     switch(pointer_category) {
27       case TypeTraits::NO_POINTER_TYPE:
28       case TypeTraits::REFERENCE_TYPE:
29       case TypeTraits::MEMBER_POINTER_TYPE:
30       case TypeTraits::MEMBER_FUNCTION_POINTER_TYPE:
31       case TypeTraits::_LAST_RESULT_ELEMENT: {
32         // Dereferenzierungen sind nur auf Objekt- und Funktions-
33         // Zeigern erlaubt[6, Kap. 5.3.1:1].
34         // Daher wird in diesem Fall ein Fehler signalisiert:
35         ecc_panic();
36       }
37       case TypeTraits::POINTER_TYPE:
38       case TypeTraits::POINTER_TO_POINTER_TYPE:
39       case TypeTraits::MEMBER_POINTER_TO_POINTER_TYPE: {
40         // Falls der Objekt-Zeiger nicht in das data- oder
41         // heap-Segment zeigt, wird ein Fehler signalisiert:
42         if(pointer < text_end_addr || pointer > program_end_addr) {
43           ecc_panic();
44         }
45         break;
46       }
```

```
47        case TypeTraits::FUNCTION_POINTER_TYPE: {
48          // Falls der Funktions-Zeiger nicht in das text-
49          // Segment zeigt, wird ein Fehler signalisiert:
50          if(pointer < text_start_addr || pointer > text_end_addr) {
51            ecc_panic();
52          }
53          break;
54        }
55      }
56    }
57  };
```

Quelltext A.2: Aspekt zur Zeiger-Validierung

Operator und Beispiel			Typ des Knotens im Clang-AST (clang::...)	Clang-Operator-Art	Besonderheiten
unär	++	a++	UnaryOperator	UnaryPostInc	Postfix-Operator[1]
unär	--	a--	UnaryOperator	UnaryPostDec	Postfix-Operator[1]
unär	++	++a	UnaryOperator	UnaryPreInc	Prefix-Operator
unär	--	--a	UnaryOperator	UnaryPreDec	Prefix-Operator
unär	&	&a	UnaryOperator	UnaryAddrOf	siehe Kap. 4.3.4
unär	*	*a	UnaryOperator	UnaryDeref	
unär	+	+a	UnaryOperator	UnaryPlus	
unär	-	-a	UnaryOperator	UnaryMinus	
unär	∼	∼a	UnaryOperator	UnaryNot	
unär	!	!a	UnaryOperator	UnaryLNot	
binär	.*	a.*b	BinaryOperator	BinPtrMemD	siehe Kap. 4.3.5
binär	->*	a->*b	BinaryOperator	BinPtrMemI	siehe Kap. 4.3.5
binär	*	a * b	BinaryOperator	BinMul	
binär	/	a / b	BinaryOperator	BinDiv	
binär	%	a % b	BinaryOperator	BinRem	
binär	+	a + b	BinaryOperator	BinAdd	
binär	-	a - b	BinaryOperator	BinSub	
binär	<<	a << b	BinaryOperator	BinShl	
binär	>>	a >> b	BinaryOperator	BinShr	
binär	<	a < b	BinaryOperator	BinLT	
binär	>	a > b	BinaryOperator	BinGT	
binär	<=	a <= b	BinaryOperator	BinLE	
binär	>=	a >= b	BinaryOperator	BinGE	
binär	==	a == b	BinaryOperator	BinEQ	
binär	!=	a != b	BinaryOperator	BinNE	

Fortsetzung auf nächster Seite...

[1] Postfix-Operatoren haben ein zweites Argument vom Typ *int*, um zwischen Post- und Prefix-Operatoren zu unterscheiden. Mehr Informationen befinden sich in dem Kapitel 4.3.3.

Operator und Beispiel			Typ des Knotens im Clang-AST (clang::...)	Clang-Operator-Art	Besonderheiten
binär	&	a & b	BinaryOperator	BinAnd	Kurzschluss-Auswertung[2]
binär	^	a ^ b	BinaryOperator	BinXor	
binär	|	a | b	BinaryOperator	BinOr	Kurzschluss-Auswertung[2]
binär	&&	a && b	BinaryOperator	BinLAnd	
binär	||	a || b	BinaryOperator	BinLOr	
binär	=	a = b	BinaryOperator/CXXOperatorCallExpr[3]	BinAssign	siehe Kap. 4.3.8
binär	,	a , b	BinaryOperator	BinComma	Links-nach-rechts-Ausw.[4]
binär	*=	a *= b	CompoundAssignOperator	BinMulAssign	
binär	/=	a /= b	CompoundAssignOperator	BinDivAssign	
binär	%=	a %= b	CompoundAssignOperator	BinRemAssign	
binär	+=	a += b	CompoundAssignOperator	BinAddAssign	
binär	-=	a -= b	CompoundAssignOperator	BinSubAssign	
binär	<<=	a <<= b	CompoundAssignOperator	BinShlAssign	
binär	>>=	a >>= b	CompoundAssignOperator	BinShrAssign	
binär	&=	a &= b	CompoundAssignOperator	BinAndAssign	
binär	|=	a |= b	CompoundAssignOperator	BinOrAssign	
binär	^=	a ^= b	CompoundAssignOperator	BinXorAssign	
binär	[ ]	a[b]	ArraySubscriptExpr		
ternär	? :	a ? b : c	ConditionalOperator		Kurzschluss-Auswertung[2]
Sonderfälle: Implizite Dereferenzierungs-Operatoren:					
Member-Zugriff (a -> b)			MemberExpr		siehe Kap. 4.3.9
Aufruf von Funktions-Zeiger			CallExpr		

Tabelle A.1: Übersicht aller 46 in dieser Arbeit betrachteten vordefinierten Operatoren

[2]Die Problematik der Kurzschluss-Auswertung wird im Kapitel 4.3.7 beschrieben.

[3]Im Fall des Kopier-Zuweisungs-Operator, der im Kapitel 4.3.8 erläutert wird, ist der AST-Knoten ein Objekt vom Typ *clang::CXXOperatorCallExpr*.

[4]Die links-nach-rechts Auswertung und die sich dadurch ergebenen Besonderheiten werden im Kapitel 4.3.6 erläutert.

①	Größe (*.elf*-Datei; Bytes)				Ausführung					
	Programmsegmente			Gesamt	Anzahl In-	Anz. Speicherzugriffe		Speichermenge (Bytes)		Anzahl
Test	*text*	*data*	*bss*		struktionen	lesen	schreib.	lesen	schreiben	Taktzyklen
bin_sem1	25.889	440	24.832	51.161	7.544.277	1.359	1.043	4.848	4.156	7.544.277
bin_sem2	26.125	432	59.904	86.461	7.647.207	33.749	23.305	126.879	91.579	647.397.407
bin_sem3	26.385	424	24.832	51.641	7.548.151	2.696	2.245	10.166	8.956	274.150.991
cnt_sem1	26.221	432	24.832	51.485	7.913.049	1.695	1.307	6.164	5.208	7.913.049
flag1	29.557	424	27.488	57.469	32.018.444	7.553	5.765	27.539	22.975	538.545.974
kill	25.937	456	27.520	53.913	164.930.958	35.548.857	2.398	35.556.462	9.516	218.252.058
mbox1	27.813	440	24.960	53.213	31.829.292	5.547	3.785	19.715	15.099	271.940.452
mqueue1	31.769	448	25.396	57.613	147.899.107	15.371	9.197	49.690	36.579	147.899.107
mutex1	25.349	440	27.552	53.341	5.523.591	2.118	1.724	7.982	6.777	5.523.591
mutex2	26.889	432	30.240	57.561	107.430.679	11.021	6.861	36.010	27.374	107.430.679
mutex3	27.869	544	38.272	66.685	546.997.947	58.042.162	39.501	232.140.124	157.329	3.156.230.747
release	25.045	424	24.832	50.301	5.525.844	2.791	1.796	10.730	7.170	5.525.844
sched1	24.409	448	24.800	49.657	5.519.293	671	519	2.250	2.068	5.519.293
sync2	27.265	440	30.240	57.945	5.577.394	82.811	63.727	329.920	252.288	5.577.394
sync3	25.869	432	27.552	53.853	11.040.240	1.946	1.478	6.919	5.878	11.040.240
thread0	23.873	440	21.984	46.297	5.703.101	626	485	2.056	1.928	5.703.101
thread1	24.765	448	24.800	50.013	5.704.506	1.149	933	4.137	3.712	32.364.766
thread2	26.897	424	27.520	54.841	272.311.269	88.870.142	2.665	355.480.088	10.622	272.311.269

Tabelle A.2: Größe der *.elf*-Datei sowie Werte der Laufzeit-Faktoren der eCos-Kernel-Tests für die Evaluations-Kombination ①
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte

②	Größe (*.elf*-Datei; Bytes)				Ausführung					
	Programmsegmente			Gesamt	Anzahl In-	Anz. Speicherzugriffe		Speichermenge (Bytes)		Anzahl
Test	*text*	*data*	*bss*		struktionen	lesen	schreib.	lesen	schreiben	Taktzyklen
bin_sem1	27.889	424	24.832	53.145	7.544.325	1.359	1.043	4.848	4.156	7.544.325
bin_sem2	28.125	448	59.904	88.477	7.647.707	33.797	23.353	127.071	91.771	647.394.767
bin_sem3	28.385	440	24.832	53.657	7.548.279	2.716	2.265	10.246	9.036	274.148.359
cnt_sem1	28.221	448	24.832	53.501	7.913.097	1.695	1.307	6.164	5.208	7.913.097
flag1	31.557	440	27.488	59.485	32.018.682	7.593	5.805	27.699	23.135	538.546.032
kill	27.937	440	27.520	55.897	164.933.631	35.549.716	2.414	35.557.369	9.580	218.252.061
mbox1	29.813	424	24.960	55.197	31.829.428	5.567	3.805	19.795	15.179	271.940.488
mqueue1	33.961	432	25.396	59.789	147.899.113	15.458	9.239	50.011	36.747	147.899.113
mutex1	27.349	424	27.552	55.325	5.523.659	2.118	1.724	7.982	6.777	5.523.659
mutex2	28.889	448	30.240	59.577	107.430.767	11.029	6.869	36.042	27.406	107.430.767
mutex3	29.869	560	38.272	68.701	546.993.544	58.041.126	39.703	232.135.983	158.158	3.156.233.874
release	27.045	440	24.832	52.317	5.525.892	2.791	1.796	10.730	7.170	5.525.892
sched1	26.409	432	24.800	51.641	5.519.341	671	519	2.250	2.068	5.519.341
sync2	29.265	456	30.240	59.961	5.577.482	82.811	63.727	329.920	252.288	5.577.482
sync3	27.869	448	27.552	55.869	11.040.308	1.946	1.478	6.919	5.878	11.040.308
thread0	25.873	424	21.984	48.281	5.703.149	626	485	2.056	1.928	5.703.149
thread1	26.765	432	24.800	51.997	5.704.562	1.151	935	4.145	3.720	32.364.742
thread2	28.897	440	27.520	56.857	272.308.606	88.869.225	2.685	355.476.420	10.702	272.308.606

Tabelle A.3: Größe der *.elf*-Datei sowie Werte der Laufzeit-Faktoren der eCos-Kernel-Tests für die Evaluations-Kombination ②
②: Erweiterte Variante des AspectC++-Compilers und eingewebter Zeiger-Validierungs-Aspekt an expliziten Dereferenzierungen (Dereferenzierungs-Operator)

③	Größe (*.elf*-Datei; Bytes)				Ausführung					
	Programmsegmente			Gesamt	Anzahl In-	Anz. Speicherzugriffe		Speichermenge (Bytes)		Anzahl
Test	*text*	*data*	*bss*		struktionen	lesen	schreib.	lesen	schreiben	Taktzyklen
bin_sem1	33.457	424	24.832	58.713	7.544.842	1.375	1.055	4.912	4.204	7.544.842
bin_sem2	33.965	432	59.904	94.301	7.670.562	35.028	24.949	131.995	98.155	647.392.442
bin_sem3	34.097	424	24.832	59.353	7.549.546	2.748	2.291	10.374	9.140	274.148.576
cnt_sem1	33.757	448	24.832	59.037	7.913.938	1.705	1.322	6.204	5.268	7.913.938
flag1	37.605	440	27.488	65.533	32.020.890	7.653	5.885	27.939	23.455	538.543.370
kill	33.777	456	27.520	61.753	164.928.686	35.548.561	2.443	35.556.325	9.696	218.249.406
mbox1	35.813	440	24.960	61.213	31.830.582	5.613	3.858	19.979	15.391	271.940.612
mqueue1	41.257	432	25.396	67.085	147.899.123	15.476	9.245	50.083	36.771	147.899.123
mutex1	32.853	424	27.552	60.829	5.524.886	2.128	1.742	8.022	6.849	5.524.886
mutex2	34.553	448	30.240	65.241	107.431.254	11.067	6.897	36.194	27.518	107.431.254
mutex3	35.693	560	38.272	74.525	547.005.764	58.041.790	40.082	232.138.639	159.674	3.156.231.124
release	32.821	424	24.832	58.077	5.529.519	2.705	1.708	10.386	6.818	5.529.519
sched1	31.833	448	24.800	57.081	5.519.475	676	523	2.270	2.084	5.519.475
sync2	34.945	440	30.240	65.625	5.635.079	83.219	64.536	331.552	255.524	5.635.079
sync3	33.581	432	27.552	61.565	11.041.277	1.975	1.501	7.035	5.970	11.041.277
thread0	31.233	440	21.984	53.657	5.703.214	626	485	2.056	1.928	5.703.214
thread1	32.525	432	24.800	57.757	5.705.109	1.160	945	4.181	3.760	32.364.779
thread2	34.817	440	27.520	62.777	272.311.486	88.869.613	2.710	355.477.972	10.802	272.311.486

Tabelle A.4: Größe der *.elf*-Datei sowie Werte der Laufzeit-Faktoren der eCos-Kernel-Tests für die Evaluations-Kombination ③
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

	Test	Anzahl Fehlerinjektions-Experimente mit jeweiligem Ergebnis				
		DETECTED	*OK*	*SDC*	*TIMEOUT*	*TRAP*
①	bin_sem1	0	1.400.887.961	346.571.489	224.390.775	962.617.839
	bin_sem3	0	402.403.073.731	44.925.868.952	155.463.005.114	225.134.734.251
	cnt_sem1	0	1.464.453.034	386.336.845	225.398.731	1.012.621.982
	mutex1	0	1.007.942.215	253.103.684	170.997.316	668.974.497
	release	0	1.028.294.761	252.984.934	163.191.543	655.832.346
	sched1	0	1.015.173.906	272.671.249	120.745.734	681.643.671
	sync2	0	1.366.620.715	268.835.032	311.867.897	902.830.812
	sync3	0	2.074.955.433	562.920.614	357.047.646	1.396.342.107
	thread0	0	1.049.542.449	244.702.519	287.932.695	584.251.761
	thread1	0	29.485.899.579	2.909.041.076	10.259.384.544	12.410.636.177
③	bin_sem1	405.644.565	1.357.700.610	320.196.170	117.516.327	734.267.824
	bin_sem3	214.375.757.529	385.930.162.584	28.273.092.618	91.069.702.779	108.270.730.370
	cnt_sem1	439.015.784	1.436.375.143	318.607.694	115.447.684	780.691.511
	mutex1	327.513.268	976.453.689	211.278.437	196.845.295	392.032.311
	release	304.705.033	996.509.712	207.609.961	83.733.044	513.085.018
	sched1	294.153.434	979.019.529	236.788.073	68.259.916	512.172.072
	sync2	470.332.507	1.436.970.089	222.397.964	188.848.779	733.816.781
	sync3	578.052.313	2.056.985.684	572.446.575	142.921.841	1.043.032.483
	thread0	287.433.017	1.012.590.216	214.328.489	116.985.851	535.121.555
	thread1	14.003.242.132	27.572.696.953	1.652.443.442	6.259.986.331	5.577.007.438

Tabelle A.5: Ergebnisse der Fehlerinjektions-Experimente
①: Bisherige Variante des AspectC++-Compilers und keine eingewebten Aspekte
③: Erweiterte Variante des AspectC++-Compilers inklusive Join-Points an impliziten Dereferenzierungen und eingewebter Zeiger-Validierungs-Aspekt an impliziten und expliziten Dereferenzierungen (Dereferenzierungs-Operator, Member-Zugriffs-Operator `->`, Aufruf eines Funktions-Zeigers)

Literaturverzeichnis

[1] Robert C. Baumann. „Radiation-Induced Soft Errors in Advanced Semiconductor Technologies“. In: *IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY, VOL. 5, NO. 3*. Sep. 2005. URL: http://www.researchgate.net/profile/Robert_Baumann3/publication/3430131_Radiation-induced_soft_errors_in_advanced_semiconductor_technologies/links/544137080cf2a76a3cc7d01d.pdf (besucht am 15. 02. 2015).

[2] Christoph Borchert. „Entwicklung eines aspektorientierten TCP/IP-Stacks für eingebettete Systeme“. Diplomarbeit. Technische Universität Dortmund, Nov. 2010.

[3] Koustav Bhattacharya. „Architectures and algorithms for mitigation of softerrors in nanoscale VLSI circuits“. Diss. Department of Computer Science und Engineering, College of Engineering, University of South Florida, Okt. 2009. URL: http://scholarcommons.usf.edu/cgi/viewcontent.cgi?article=2857&context=etd (besucht am 15. 02. 2015).

[4] Christoph Borchert, Horst Schirmeier und Olaf Spinczyk. „Generative Software-based Memory Error Detection and Correction for Operating System Data Structures“. In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. (Budapest, Hungary). IEEE Computer Society Press, Juni 2013. ISBN: 978-1-4673-6471-3. DOI: 10.1109/DSN.2013.6575308.

[5] Bianca Schroeder, Eduardo Pinheiro und Wolf-Dietrich Weber. „DRAM errors in the wild: a large-scale field study“. In: *ACM SIGMETRICS Performance Evaluation Review*. Bd. 37. 1. ACM. 2009, S. 193–204. URL: http://www.cs.utoronto.ca/~bianca/papers/sigmetrics09.pdf (besucht am 15. 02. 2015).

[6] *ISO/IEC 14882 Programming Languages — C ++, working draft. Internationale Organisation für Normung (ISO), 2013-05-15*. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf (besucht am 24. 01. 2015).

[7] Daniel Wulfert. „Aspektorientierte Implementierung und Bewertung von Fehlertoleranzmechanismen im eingebetteten Betriebssystem eCos“. Diplomarbeit. Technische Universität Dortmund, März 2013.

[8] Ikhwan Lee, Mehmet Basoglu, Michael Sullivan, Doe Hyun Yoon, Larry Kaplan und Mattan Erez. *Survey of Error and Fault Detection Mechanisms*. Techn. Ber. Locality, Parallelism, Hierarchy Group, Department of Electrical und Computer Engineering, The University of Texas at Austin, Apr. 2011.

[9] Tzilla Elrad, Robert E. Filman und Atef Bader. „ASPECT-ORIENTED PROGRAMMING“. In: *COMMUNICATIONS OF THE ACM, Vol. 44, No. 10*. Okt. 2001. URL: http://plg.uwaterloo.ca/~migod/846/papers/aop-intro.pdf (besucht am 15. 02. 2015).

[10] Olaf Spinczyk und Daniel Lohmann. „The Design and Implementation of AspectC++“. In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20.7 (2007), S. 636–651. DOI: 10.1016/j.knosys.2007.05.004.

[11] *"clang"C Language Family Frontend for LLVM*. URL: http://clang.llvm.org/ (besucht am 16. 02. 2015).

[12] *Language Compatibility*. URL: http://clang.llvm.org/compatibility.html (besucht am 16. 02. 2015).

[13] Rainer Leupers. *C-Compiler für Embedded Systems - Technologie und Werkzeuge zur Compilerentwicklung*. Techn. Ber. Lehrstuhl Informatik 12, Universität Dortmund. URL: http://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2000-elektronik.pdf (besucht am 16. 02. 2015).

[14] Matthias Urban, Daniel Lohmann und Olaf Spinczyk. „The aspect-oriented design of the PUMA C/C++ parser framework“. In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. ACM. 2010, S. 217–221. URL: http://aosd.net/2010/files/pdf/industry/urban.pdf (besucht am 16. 02. 2015).

[15] pure-systems GmbH, Olaf Spinczyk und Daniel Lohmann. *AspectC++ Quick Reference*. URL: http://www.aspectc.org/doc/ac-quickref.pdf (besucht am 15. 10. 2014).

[16] pure-systems GmbH und Olaf Spinczyk. *AspectC++ Language Reference*. URL: http://www.aspectc.org/doc/ac-languageref.pdf (besucht am 15. 10. 2014).

[17] *Introduction to the Clang AST — Clang 3.4 documentation*. URL: http://www.llvm.org/releases/3.4/tools/clang/docs/IntroductionToTheClangAST.html (besucht am 19. 01. 2015).

[18] *The Clang AST - A Tutorial by Manuel Klimek*. URL: http://llvm.org/devmtg/2013-04/klimek-slides.pdf (besucht am 19. 01. 2015).

[19] *clang: clang::ASTConsumer Class Reference*. URL: http://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html (besucht am 21. 01. 2015).

[20] *clang: clang::RecursiveASTVisitor< Derived > Class Template Reference*. URL: http://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html (besucht am 21. 01. 2015).

[21] *How to write RecursiveASTVisitor based ASTFrontendActions. — Clang 3.7 documentation*. URL: http://clang.llvm.org/docs/RAVFrontendAction.html (besucht am 21. 01. 2015).

[22] *clang: Cursor manipulations*. URL: http://clang.llvm.org/doxygen/group__CINDEX__CURSOR__MANIP.html#gace7a423874d72b3fdc71d6b0f31830dd (besucht am 21. 01. 2015).

[23] *clang: clang::Stmt Class Reference*. URL: http://clang.llvm.org/doxygen/classclang_1_1Stmt.html (besucht am 23. 01. 2015).

[24] *LLVM Programmer's Manual - The isa<>, cast<> and dyn_cast<> templates — LLVM 3.7 documentation*. URL: http://llvm.org/docs/ProgrammersManual.html#isa (besucht am 24. 01. 2015).

[25] Jason W. Bacon. *Computer Science 315 Lecture Notes*. URL: http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/index.html (besucht am 15. 02. 2015).

[26] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann und Olaf Spinczyk. „FAIL*: Towards a Versatile Fault-Injection Experiment Framework“. In: *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*. Hrsg. von Gero Mühl, Jan Richling und Andreas Herkersdorf. Bd. 200. Lecture Notes in Informatics. Munich, Germany: German Society of Informatics, März 2012, S. 201–210. ISBN: 978-3-88579-294-9.

[27] *bochs: The Open Source IA-32 Emulation Project (Home Page)*. URL: http://bochs.sourceforge.net/ (besucht am 24. 02. 2015).

[28] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk und Wolfgang Schröder-Preikschat. „A Quantitative Analysis of Aspects in the eCos Kernel“. In: *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*. New York, NY, USA: ACM Press, Apr. 2006, S. 191–204. DOI: 10.1145/1218063.1217954.

[29] *About eCos*. URL: http://ecos.sourceware.org/about.html (besucht am 18. 02. 2015).

[30] *Test Suites*. URL: http://ecos.sourceware.org/docs-1.3.1/tutorials/arm/ecos-tutorial.12.html (besucht am 18. 02. 2015).

[31] Christoph Borchert, Horst Schirmeier und Olaf Spinczyk. „Generative Software-based Memory Error Detection and Correction for Operating System Data Structures“. In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. (Budapest, Hungary). IEEE Computer Society Press, Juni 2013. ISBN: 978-1-4673-6471-3. DOI: 10.1109/DSN.2013.6575308.

[32] *Qt Project*. URL: http://qt-project.org/ (besucht am 27. 02. 2015).

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis