AspectC++

Benjamin Polak <ben@polak.de>

6. Juli 2003

Zusammenfassung

AspectC++ ist eine Spracherweiterung für aspektorientierte Programmierung mit C++. Die wesentlichen Sprachelemente werden kurz vorgestellt. Darüber hinaus wird die Eignung von AspectC++ für den Betriebssystembau anhand zweier Fallstudien exemplarisch gezeigt.

1 AspectC++

AspectC++ erweitert die Programmiersprache C++ um Sprachkonstrukte für aspektorientierte Programmierung. Die AspectC++-Spracherweiterung ist angelehnt an AspectJ, weist jedoch naturgemäß einige Unterschiede auf, bedingt durch die unterschiedlichen zugrundeliegenden Sprachen C++ und Java. Im folgenden werden die grundlegenden Konzepte und Sprachkonstrukte von AspectC++ vorgestellt.

1.1 Join Points und Pointcuts

Mit Hilfe von Aspekten werden sog. "*cross-cutting concerns*" realisiert. "*Cross-cutting*" bedeutet, dass Aspekt-Code an mehreren verschiedenen Stellen innerhalb des Anwendungs- oder System-Codes einwirkt. Man benötigt also einen Formalismus, um diese Stellen zu spezifizieren. AspectC++ stellt hierfür die folgenden Konzepte und Sprachmittel zur Verfügung:

Join Points sind Stellen im Anwendungscode, an denen Aspekte greifen können. Join Points beziehen sich auf Methoden, Attribute, Typen, Objekte, oder Stellen im Kontrollfluss, von denen aus auf andere Join Points zugegriffen wird.

Pointcuts sind Mengen von Join Points, die jeweils durch eine **Pointcut Expression** beschrieben werden. Dabei kann eine Pointcut Expression zusammengesetzt sein aus Match Expressions, Pointcut Functions, und algebraischen Operatoren.

Match Expressions sind Zeichenketten, die als Suchmuster dienen und optional Wildcard-Operatoren enthalten, nämlich "`%`" als Wildcard für Namensteile und "`...`" als Wildcard für eine beliebige Anzahl von Parametern in einer Funktionssignatur. Zur Verdeutlichung folgende Beispiele:

`int` wählt den gleichnamigen primitiven Datentypen von C++ aus

`%List` wählt alle Klassen, struct's, oder union's aus, deren Name auf "List" endet

`% printf(...)` wählt alle Funktionen namens "printf" mit beliebigen Rückgabetypen und Parametern aus

Pointcut Functions sind vordefinierte Funktionen zum Herausfiltern oder Abbilden von Join Points aus Pointcuts. Hier einige Beispiele:

`derived("List")` wählt alle Klassen aus, die von der Klasse "List" abgeleitet sind

`call("% printf(...)")`
wählt alle Aufrufstellen der Funktion "printf" aus

`execution("% printf(...)")`
wählt die Implementierung der Funktion "printf" aus

Algebraische Operatoren dienen zur Verknüpfung von Pointcuts. Mit "`&&`" wird die Schnittmenge zweier Pointcuts gebildet, mit "`||`" die Vereinigungsmenge, und mit "`!`" die Komplementmenge. Beispiele:

`"%List" || "%Queue"`
wählt alle Klassen aus, deren Name auf "List" oder "Queue" endet

`call("void draw()") && within("Shape")`
wählt alle Aufrufe der Funktion "draw()" aus, die innerhalb der Klasse "Shape" durchgeführt werden

Es ist möglich, Pointcuts zu deklarieren. Eine Pointcut-Deklaration wird eingeleitet durch das Schlüsselwort "`pointcut`". Beispiel:

```
pointcut lists() = derived("List");
```

Pointcuts können auch als "`virtual`" deklariert werden. Damit lassen sich wiederverwendbare Aspektimplementierungen erstellen (mehr dazu weiter unten). Beispiel:

```
pointcut virtual methods() = 0;
```

1.2 Advices

Bei Advice Code handelt es sich um Code, der an Join Points gebunden wird. Eine Advice-Deklaration wird eingeleitet durch das Schlüsselwort "`advice`". Darauffolgend werden die Join Points spezifiziert, entweder direkt durch eine Pointcut Expression oder durch Referenz auf einen deklarierten Pointcut. Der Advice Code wird aktiviert, wenn die entsprechenden Join Points bei der Ausführung der Anwendung oder des Systems erreicht werden. Die Aktivierung kann dabei

jeweils vor dem Join Point, nach dem Join Point, oder statt des Join Points erfolgen. Um dies zu bestimmen, gibt es die drei Ausdrücke "`before()`", "`after()`" und "`around()`". Beispielsweise würde die folgende Advice-Deklaration bewirken, dass vor der Ausführung der "`login()`"-Methode eine Meldung ausgegeben wird:

```
advice execution("void login(...)") : before() {
    cout << "Logging in." << endl;
}
```

Statt der drei oben erwähnten Ausdrücke (`before()`, `after()`, und `around()`) kann auch eine beliebige gültige C++-Deklaration angegeben werde, diese wird dann an den entsprechenden Join Points eingebunden. Der Advice wird in diesem Fall als **Introduction** bezeichnet. Im folgenden Beispiel werden per Introduction ein neues Attribut und eine neue Methode in zwei Klassen eingefügt:

```
pointcut classes() = "Circle" || "Rect";

advice classes() : bool shaded;

advice classes() : void set_shaded(bool state) {
    shaded = state;
}
```

1.3 Aspects

Um logisch zusammengehörige Advices, Introductions, und deren Zustandsinformationen zu modularisieren und in eine Einheit zu kapseln dient das "`aspect`"-Sprachelement. Dabei handelt es sich um eine Erweiterung des Klassenkonzeptes von C++. Aspektdeklarationen sind nach dem gleichen Schema aufgebaut wie Klassendeklarationen, Aspekte können genau wie Klassen auch Attribute und Methoden enthalten, Aspekte können von anderen Aspekten und Klassen erben. Hier ein Beispiel:

```
aspect InstantiationCounter {

    static int count;

    InstantiationCounter() : count(0) { }

    pointcut counted() = "Circle" || "Rect";

    advice counted() : class CounterHelper {
        CounterHelper() {
            InstantiationCounter::count++;
        }
    } helper;
```

```
    advice execution("% main(...)") : after() {
        cout << "Final instantiation count: "
             << InstantiationCounter::count << endl;
    }

};
```

Dieses Beispiel realisiert einen Instantiierungszähler für die beiden Klassen "`Circle`" und "`Rect`". Eine Verbesserungsmöglichkeit wäre es, diesen Aspekt wiederverwendbar zu machen. Dies lässt sich bewerkstelligen, indem der Pointcut "`counted()`" als rein virtuell deklariert wird:

```
aspect InstantiationCounter {
    // ...
    pointcut virtual counted() = 0;
    // ...
};
```

Soll nun für eine konkrete Menge von Klassen ein Instantiierungszähler angelegt werden, muss der Aspekt abgeleitet werden und der virtuelle Pointcut "`counted()`" definiert werden:

```
aspect ShapesCounter : public InstantiationCounter {
    pointcut counted() = derived("Shape");
};
```

1.4 Join Point API

Oftmals werden im Advice-Code Informationen über den Join Point benötigt, der bei der Aktivierung des Advice-Codes vorliegt. Hierzu stellt AspectC++ eine Join Point API zur Verfügung. Damit lassen sich z.B., im Falle einer Funktion als Join Point, die Meta-Informationen wie Parameter- und Rückgabetypen, oder auch konkrete Informationen wie Parameter- und Rückgabewerte abfragen.

Mit Hilfe der Join Point API kann z.B. ein wiederverwendbarer Aspekt zur Kontrollflussverfolgung ("`Tracer`") realisiert werden:

```
aspect Tracer {
    pointcut virtual functions() = 0;
    advice execution(functions()) : around() {
        cout << JoinPoint::signature() << "(";
        // print parameters
        for (unsigned i=0; i<JoinPoint::args(); i++) {
            cout << (i ? "," : "");
            print_value(JoinPoint::argtype(i),
                    thisJoinPoint->arg(i));
        }
        cout << ")";
```

```
        // execute function
        thisJoinPoint->action().trigger();
        // print result
        cout << ", result=";
        print_value(JoinPoint::resulttype(),
                thisJoinPoint->result());
    }
}
```

1.5 AspectC++ Compiler

Der AspectC++ Compiler ist als Präprozessor realisiert, der AspectC++-Code in C++-Code transformiert, und dabei die Aspekte einwebt. Das Transformationsergebnis kann dann mit einem normalen C++-Compiler wie GNU g++ oder Microsoft VisualC++ kompiliert werden.

2 Aspekte in der Betriebssystemfamilie PURE

Die Anwendung von aspektorientierter Programmierung im Rahmen der Betriebssystementwicklung soll hier aufgezeigt werden anhand zweier Fallstudien an der existierenden Betriebssystemfamilie PURE[3].

2.1 Unterbrechungssynchronisation

In vielen Teilen des Betriebssystems (z.B. beim *Memory Management*, *I/O*, *Scheduling*) gibt es kritische Bereiche, die geschützt werden müssen vor überlappter Ausführung, wie sie durch Unterbrechungen hervorgerufen werden kann. Bei der Unterbrechungssynchronisation handelt es sich daher um einen *cross-cutting concern*.

Innerhalb der Betriebssystemfamilie PURE wird das Prolog/Epilog-Modell zur Synchronisation von Unterbrechungen verwendet. Dabei werden die kritischen Bereiche eingerahmt durch Aufrufe der Form "`lock.enter()`" und "`lock.leave()`". Treten dazwischen Unterbrechungen auf, werden die mit der Unterbrechungsbehandlung verbundenen kritischen Operationen entsprechend verzögert.

2.1.1 Probleme der herkömmlichen Implementierung

Die Synchronisationsaufrufe sind sehr zahlreich und quer durch das System verstreut, was in mancher Hinsicht problematisch ist:

- Von der Unterbrechungssynchronisation abhängige Module sind schlecht wiederverwendbar in Umgebungen mit keinem Synchronisationsbedarf oder anderen verwendeten Synchronisationsverfahren, da die Synchronisationsaufrufe ja fest in die Module einkodiert sind.

- Die Strategie der Synchronisation (z.B. hinsichtlich Granularität) ist nicht flexibel änderbar, da dies eine aufwendige und fehleranfällige Anpassung der zahlreichen Synchronisationsaufrufe erfordern würde.

2.1.2 Verwendung von AOP

Im Rahmen einer Fallstudie wurde die Prolog/Epilog-Unterbrechungssynchronisation in PURE mittels AOP reimplementiert. Wie sich zeigte, ergaben sich daraus eine Reihe von Vorteilen im Vergleich zu der herkömmlichen Implementierung:

- Die verstreuten Synchronisationsaufrufe konnten aus allen Modulen entfernt werden, was deren Wiederverwendbarkeit deutlich verbesserte.
- Da die Unterbrechungssynchronisation vollständig in einem Aspekt modularisiert wurde, können hier auch einfach zentral Änderungen an der Synchronisationsstrategie vorgenommen werden.
- Der Quellcode-Umfang sowie der Speicherplatzverbrauch des Kompilats konnten reduziert werden, da einige Wrapper-Klassen, Schnittstellen-Klassen, u.ä. entfernt werden konnten.

Die Realisierung des Aspekts sah dabei wie folgt aus:

```
// Beschreibung der PURE Subsysteme
pointcut osek()   = derived("osekBase") || "osekOS";
pointcut thread() = derived("Schemer") && !osek();
pointcut case()   = derived("Counter");
// ...

// Aspekt zur grobgranularen Interruptsynchronisation
aspect IntSync {

    // Zu synchronisierende Region
    pointcut kernel() = osek() || thread() || case();

    // Eintritte in die zu synchronisierende Region
    pointcut locked() = call(kernel()) &&
            !within(kernel());

    // Austritte aus der zu synchronisierenden Region
    pointcut unlocked() = within(kernel()) &&
            call("void Triplet::action()");

    // Advice-Code zum Sperren und Erlauben von Epilogen
    advice unlocked() : before() { lock.leave(); }
    advice unlocked() : after()  { lock.enter(); }
    advice locked()   : before() { lock.enter(); }
    advice locked()   : after()  { lock.leave(); }

}
```

2.2 Fadensynchronisation

Nebenläufig ausgeführte Fäden müssen synchronisiert werden, falls sie gemeinsam auf Betriebsmittel wie z.B. Gerätetreiber zugreifen. In PURE sind Gerätetreiber unsynchronisiert, d.h. die Fäden sind hierbei selbst für Synchronisation verantwortlich.

Inwieweit sich eine Fadensynchronisation auch mittels AOP realisieren lässt, wurde in einer weiteren Fallstudie am Beispiel eines Floppy-Treibers untersucht.

Eine Fadensynchronisation mittels gegenseitigem Ausschluss wurde durch folgende Aspekte realisiert:

```
// Abstrakter Aspekt fuer gegenseitigen Ausschluss
aspect Mutex {
    Semaphore mutex;
    pointcut virtual funcs() = 0;
    Mutex() : mutex(1) {} // Semaphor-Initialisierung
    // Abfangen und Schutz aller Aufrufe
    advice funcs() : before() { mutex.wait(); }
    advice funcs() : after()  { mutex.signal(); }
};

// Konkreter Aspekt fuer gegenseiteigen Ausschluss in
// Floppy- und DMA-Treiber
aspect FloppyAndDMAMutex : public Mutex {
    pointcut inst() = "DMA8237A"||"PCFloppyController";
    pointcut floppy() = derived("PCFloppyController") &&
                        base("PCFloppyDriver");
    pointcut funcs() =
        (call("DMA8237A") && !within("DMA8237A")) ||
        (call(floppy()) && !within(floppy()));
    advice inst() : FloppyAndDMAMutex __inst;
    static FloppyAndDMAMutex *aspectOf() {
        return &thisJoinPoint->target()->__inst;
    }
};
```

3 Fazit

Die Umsetzung des Konzepts der aspektorientierten Programmierung scheint bei AspectC++ gelungen. Die Vorteile, wie sie die AOP verspricht, werden hier tatsächlich erreicht. Dies belegt der erfolgreiche Einsatz von AspectC++ beim Betriebssystembau, wie im Rahmen von Fallstudien an der existierenden Betriebssystemfamilie PURE gezeigt.

Literatur

[1] pure-systems GmbH, Matthias Urban, Olaf Spinczyk, "AspectC++ Language Reference", Juni 2003

[2] pure-systems GmbH, Olaf Spinczyk, "AspectC++ Compiler Manual", Juni 2003

[3] Olaf Spinczyk, "Aspektorientierung und Programmfamilien im Betriebssystembau", Dissertation, Dezember 2002

[4] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat, "An Aspect-Orientied Implementation of Interrupt Synchronization in the PURE Operating System Family", Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems, Malaga, Spain, June 11th, 2002, ISBN 84-699-8733-X

[5] Olaf Spinczyk, Andreas Gal, "Aspektorientierung und Betriebssysteme", Tutorium im Rahmen des Herbsttreffens der Fachgruppen "Betriebssysteme" der GI und ITG (Berlin, 7.11.2002)