Aspektorientierung und Programmfamilien im Betriebssystembau

Olaf Spinczyk

os@aspectc.org

Abstract: In der hier zusammengefassten Dissertation geht es um die Analyse, den Entwurf und die Implementierung von maßgeschneiderter Systemsoftware, wie sie beispielsweise im Bereich kleinster eingebetteter Systeme benötigt wird, um Hardware-Anforderungen zu reduzieren und damit Kosten zu sparen. Dabei wurde der Gedanke der Betriebssystemfamilie aufgegriffen und mit dem modernen Ansatz der aspektorientierten Programmierung (AOP) kombiniert. Die so entstandenen Methoden, Werkzeuge und Fallstudien zeigen neue Perspektiven für diese Klasse von Betriebssystemen auf und lassen sich auch auf andere Bereiche sinnvoll übertragen. Die vollständige Fassung der Arbeit ist elektronisch über die Universitätbibliothek Magdeburg (http://diglib.uni-magdeburg.de/) verfügbar .

1 Einleitung

Wer heutzutage das Wort Betriebssysteme hört, denkt im Allgemeinen zuerst an Systeme für Arbeitsplatzrechner wie Windows oder Linux. Dabei gibt es bedeutende Anwendungsgebiete, für die diese Vielzwecksysteme schlecht oder gar nicht geeignet sind. Leistungsstarke Parallelrechner, wie sie zum Beispiel für aufwendige Simulationen benötigt werden, gehören ebenso dazu wie sicherheitskritische Bankrechner oder die in immer mehr Geräten enthaltenen eingebetteten Systeme. Ihnen allen ist gemeinsam, dass sie verglichen mit Arbeitsplatzrechnern ein eingeschränktes Aufgabenfeld haben. Dafür werden an sie aber besondere Ansprüche zum Beispiel bezüglich der Ausnutzung der Rechenleistung oder der Ausfallsicherheit gestellt.

Dies gilt insbesondere für die sogenannten eingebetteten Systeme (engl. *embedded systems*). Das sind Rechnersysteme inklusive dazugehöriger Software, die in ein bestimmtes Produkt integriert sind. Dort haben sie in der Regel Steuerungsaufgaben auszuführen. Beispiele sind mikrocontroller-basierte Rechnersysteme in modernen Haushaltsgeräten oder im Automobil.

Während das Betriebssystem für einen typischen Vielzweck-Arbeitsplatzrechner auf eine große Menge potentiell angeschlossener Geräte und ein sehr weites Anwendungsspektrum eingerichtet sein muss, ist bei einem eingebetteten System die zu betreibenden Hardware von Anfang an ebenso bekannt wie die zu erfüllende Aufgabe. Dafür haben die Entwickler oft mit extremer Ressourcenknappheit zu kämpfen. Ganz besonders gilt dies für den

wichtigen Bereich der kleinsten (“tiefst”) eingebetteten Systeme, wo vorwiegend 4 und 8 Bit Mikrocontroller mit nur wenigen KBytes Hauptspeicher eingesetzt werden[1]. In Anbetracht der Tatsache, dass bereits ein einfaches “hello, world” Programm auf einem PC System mehrere hundert KBytes[2] Speicher einnimmt, wird klar, das es besonderer Techniken bedarf, um für solche Systeme überhaupt Software entwickeln zu können.

“Betriebssysteme von der Stange”, die alle denkbaren Anwendungen unterstützen, sind hier wegen ihres Ressourcenverbrauchs nicht einsetzbar. Stattdessen werden anwendungsspezifisch “maßgeschneiderte Systeme” benötigt. Solche Systemsoftware wird häufig als in-house Lösung für spezielle Projekte entwickelt [FSH+01]. Da trotz der Unterschiede aber auch viele Gemeinsamkeiten zwischen diesen Systemen existieren, muss leider davon ausgegangen werden, dass hier das Rad immer wieder neu erfunden wird.

Ein besserer Weg wäre die Entwicklung eines für jedes Projekt individuell “maßschneiderbaren” Systems, dessen Teilfunktionen immer wiederverwendet werden. Dabei sollten sich die Konfigurierungsmaßnahmen nicht nur auf grobe Systembausteine wie Gerätetreiber oder Dateisysteme auswirken, sondern feingranular auf alle Funktionen und Eigenschaften des Systems, um im Ressourcenverbrauch mit Spezialimplementierungen konkurrieren zu können.

Kommerzielle Systeme bieten diese feingranulare Konfigurierbarkeit derzeit noch nicht, da die Variantenvielfalt schnell unbeherrschbar wird. Daher ist es ein seit mehreren Jahren verfolgtes Ziel der Arbeitsgruppe für Betriebssysteme und Verteilte Systeme an der Universität Magdeburg gewesen, Methoden und Werkzeuge zu entwickeln, die dieses Problem mindern, und die praktische Nutzbarkeit durch die exemplarische Entwicklung der PURE Betriebssystemfamilie [BGP+99] zu demonstrieren.

Die hier zusammengefasste Dissertation ist in diesem Kontext zu sehen. Dabei gilt das besondere Augenmerk der Konfigurierbarkeit solcher Systemeigenschaften, deren Implementierung nicht oder nur sehr schwer modularisierbar ist und somit über weite Teile des Systems verstreut vorliegt. Beispiele sind Synchronisationscode oder Schutz im Mehrbenutzerbetrieb. Wo solcher Code eingebaut wird und was er zu tun hat, gehört zu den strategischen Entwurfsentscheidungen beim Bau eines Betriebssystems. Die Konfigurierbarkeit solcher Eigenschaften stellt eine besondere Herausforderung dar.

2 Programmfamilien

Die mit PURE verfolgte Entwicklung orientierte sich anfangs ausschließlich am Konzept der Programmfamilie, das bereits Mitte der 70er Jahre vor dem Hintergrund von Betriebssystementwicklungen entstand [HFC76][Par76]. Der wesentliche Grundbegriff des familienbasierten Entwurfs ist die Funktion des zu realisierenden Systems. Eine Funktion ist in diesem Kontext eine Leistung oder ein Dienst, der von einem Softwaresystem erbracht wird. Üblicherweise können Funktionen in Teilfunktionen zerlegt werden (funk-

[1] im Jahr 2000 wurden etwa 80% der hergestellten Mikroprozessoren bzw. Mikrocontroller für solche 4 oder 8 Bit Systeme verwendet[Tur02].

[2] 352456 Bytes mit gcc 2.96 unter RedHat Linux 7.1 für x86

tionale Dekomposition). Die Mitglieder einer Programmfamilie unterscheiden sich durch die Teilfunktionen aus denen sie gebildet werden und damit in ihrer Gesamtfunktion.

Der Vorteil bei der Zerlegung einer Funktion in Teilfunktionen, die sich gegenseitig benutzen, besteht in der Möglichkeit der Wiederverwendung. Bei Programmfamilien spielt dies eine besonders große Rolle, da neue Familienmitglieder mit möglichst wenig Aufwand realisiert werden sollen, indem Funktionen von existierenden Familienmitgliedern wiederverwendet werden. Dies setzt aber eine entsprechende Strukturierung der Software voraus.

Um ein hohes Maß an Wiederverwendung zu erzielen und Redundanz im Code zu vermeiden, wird die gegenseitige Benutzung von Funktionen beim familienbasierten Entwurf so beschränkt, dass keine Zyklen im Abhängigkeitsgraphen vorliegen. So entstehende Graphen werden als funktionale Hierarchie bezeichnet. Als Beispiel zeigt Abbildung 1 die funktionale Hierarchie des FAMOS Systems von N. Habermann [HFC76], das sich für die Erläuterung der Idee besonders gut eignet.

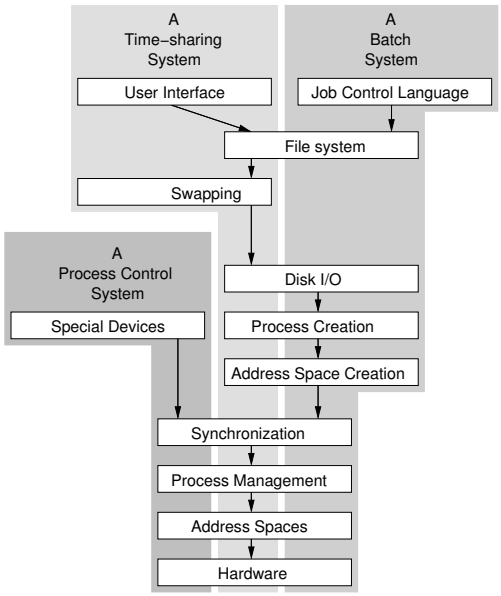

Abbildung 1: Die Funktionale Hierarchie des FAMOS Systems

FAMOS war ein aus Schichten aufgebautes System, von denen jede nur die darunterliegenden Schichten kennen und zur Erfüllung ihrer Aufgabe benutzen durfte. Die beschrifteten Rechtecke in der Abbildung repräsentieren die Funktionen des Systems. Durch einfache Konfigurierungsmaßnahmen konnten daraus drei verschiedene Familienmitglieder erzeugt werden: Ein *Process Control* System, ein *Time-sharing* System und ein *Batch* System. Durch die Abhängigkeitsbeziehungen in der funktionalen Hierarchie konnte beispielsweise modelliert werden, dass für die *Special Devices* in der *Process Control System* Variante von FAMOS auch die Funktionen *Synchronization*, *Process Management*, *Address Spaces* und *Hardware* gebraucht werden. Diese vier Basisfunktionen werden ebenfalls für die

anderen beiden Familienmitglieder benötigt. Dafür kann dort auf die Funktion *Special Devices* verzichtet werden.

Auf den ersten Blick mag die Zerlegung einer Software in Schichten, wie sie hier vorgenommen wurde, trivial wirken. Tatsächlich erfordert dies jedoch ein großes Maß an Erfahrung. Zum Beispiel wurden bei FAMOS die Funktionen Prozessverwaltung (*Process Management*) und Prozesserzeugung (*Process Creation*) voneinander getrennt, obwohl beide letztlich mit den gleichen Datenstrukturen arbeiten. Dadurch wird erreicht, dass in der *Process Control* Systemkonfiguration kein Ballast in Form ungenutzten Programmcodes entsteht, denn eine dynamische Prozesserzeugung wird hier nicht gebraucht. Beim Entwurf jeder einzelnen Funktion bzw. Schicht muss also versucht werden, Entwurfsentscheidungen zu vermeiden, die der Erweiterung der Familie im Wege stehen könnten. Dies verlangt viel Weitblick und Disziplin.

3 Aspektorientierte Programmierung

Die beschriebenen funktionalen Hierarchien eignen sich nicht, um Eigenschaften wie "Schutz im Mehrbenutzerbetrieb" zu erfassen. Es handelt sich dabei um ein sogenanntes *Crosscutting Concern*, das sich bezogen auf Abbildung 1 unter anderem auf die Funktionen *File System*, *Disk I/O* und *User Interface* auswirkt. Dort muss zum Beispiel überprüft werden, ob der aktuelle Benutzer berechtigt ist, eine bestimmte Datei oder ein bestimmtes Gerät zu benutzen. Die entsprechenden Zugriffsrechte müssen in Datenstrukturen verwaltet werden und die Benutzerschnittstelle muss Operationen zur Manipulation dieser Daten bereitstellen.

Crosscutting Concerns stellen grundsätzlich eine Gefahr für die Wiederverwendbarkeit und Erweiterbarkeit von Programmcode dar. Daher ist das Ziel der noch vergleichsweise jungen Disziplin der aspektorientieren Programmierung [KLM+97], im Sinne des Prinzips *Separation of Concerns*, Code, der ein *Crosscutting Concern* implementiert (z.B. Schutz), vom Code, der die eigentliche Funktionalität erbringt (z.B. *Disk I/O*) zu trennen. Mit Hilfe programmiersprachlicher Mittel wird dieser Code zu einem eigenständigen Modul zusammengefasst. Entsprechende Module werden als Aspekte bezeichnet. Welche Sprachmittel dies sind, wird in Abschnitt 4 beschrieben.

Durch aspektorientierte Programmierung wird die Konfigurierung derartiger Eigenschaften wesentlich erleichtert. Dieser Gedanke war der Ausgangspunkt für die hier beschriebene Arbeit. Wenn man *Crosscutting Concerns* in einer funktionalen Hierarchie erfassen könnte, wäre es beispielsweise möglich, alternative Varianten des "Schutzes im Mehrbenutzerbetrieb" zu erfassen, von denen wie bei herkömmlichen Funktionen eine entsprechend der Anwendungsanforderungen gewählt werden kann. Diese Varianten könnten völlig verschiedene Punkte im System betreffen, so dass durch eine einzelne Konfigurierungsentscheidung eine sehr große Wirkung erzielt würde.

Leider sind die Beziehungen von *Crosscutting Concerns* zu herkömmlichen Funktionen und auch ihre Beziehungen untereinander anders geartet als die Beziehungen zwischen Funktionen. Daher wurden im Rahmen der Arbeit diese Beziehungen analysiert und es

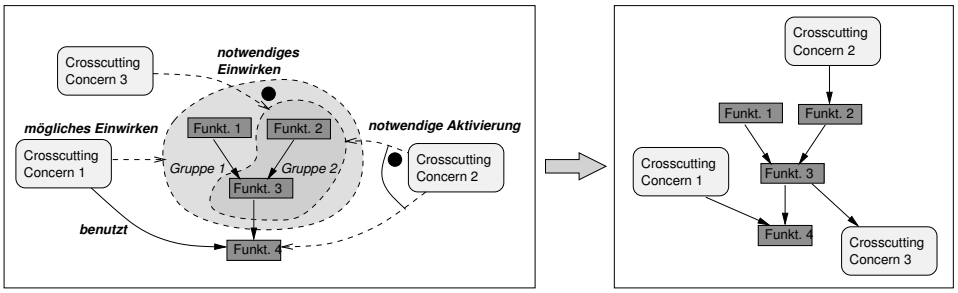

Abbildung 2: Eine *Concern*-Hierarchie und deren Abbildung auf Abhängigkeitsbeziehungen

wurde das Modell der sogenannten *Concern*-Hierarchie entwickelt, das die funktionalen Hierarchien um die notwendigen Ausdrucksmittel erweitert. Abbildung 2 zeigt links eine solche *Concern*-Hierarchie. Sie zeigt unter anderem das "Einwirken" von *Crosscutting Concerns* auf Gruppen von Funktionen. Der Pfeil soll andeuten, dass sich eine derartige Struktur automatisch in einen Abhängigkeitsgraphen überführen lässt, der für die Ermittlung von zulässigen Konfigurationsvarianten benötigt wird. Der entsprechende Graph ist auf der rechten Seite zu sehen.

Concern-Hierarchien sind als Hilfsmittel für die Modellierung aspektorientierter Programmfamilien zu sehen und bilden eine wichtige Grundlage für die methodische Erstellung hochgradig konfigurierbarer Programmfamilien.

4 AspectC++

Die aspektorientierte Programmierung wird durch Sprachmittel ermöglicht, mit deren Hilfe ein Programmmodul auf andere Module einwirken kann, ohne dass diese Module dafür in irgendeiner Weise präpariert sein müssten [EFB01]. Die bekannteste Sprache, die dies unterstützt, ist die Java Erweiterung AspectJ [KHH+01].

Leider ist eine Java-basierte Sprache im Kontext tiefst eingebetteter Systeme nicht einsetzbar und eine Betrachtung möglicher Alternativen im C/C++ Umfeld zeigte, dass auch dort brauchbare Sprachen nicht existierten. Daher wurde im Rahmen der Dissertation AspectC++ [SGSP02] entworfen und ein Übersetzer dafür implementiert[3]. Wie der Name andeutet, ist AspectC++ eine an die Syntax und Semantik von AspectJ angelehnte, aspektorientierte Spracherweiterung zu C++.

In AspectC++ werden die Verbindungspunkte, an denen ein Aspekt auf andere Module wirkt, mit Hilfe sogenannter *Pointcut*-Ausdrücke beschrieben, die, wie im folgenden Codefragment zu sehen ist, auch einen Namen bekommen können:

```
pointcut IRQ_level_call(int level) =
  call("void IRQ::level(int)") && args(level);
```

[3] siehe `http://www.aspectc.org/`

Diese Definition des *Pointcuts* `IRQ_level_call` beschreibt beispielsweise alle Punkte im System, an denen die Funktion `IRQ::level(int)` aufgerufen wird. Mit Hilfe von sogenanntem *Advice*-Code kann nun ein Aspekt bestimmen, was an diesen Verbindungspunkten an zusätzlichen Aktionen geschehen soll:

```
aspect Debug {
  int _last;
  advice IRQ_level_call(l) : after (int l) {
    printf ("IRQ-level: %d->%d", _last, l);
    _last = l;
  }
};
```

Der Aspekt bildet lediglich eine modulare Einheit, um ähnlich wie eine C++ Klasse zusammengehörige Daten und Operationen zu kapseln. Die *Advice*-Definition besagt, dass jeweils nach ("after") dem Aufruf der Funktion `IRQ::level(int)` das angegebene Codefragment auszuführen ist, das durch eine Ausgabe des jeweils letzten und aktuellen Argumentwertes der so beobachteten Aufrufe zur Fehlersuche ("Debug") beiträgt.

Neben den hier gezeigten Sprachelementen bietet AspectC++ noch wesentlich weiter reichende Möglichkeiten, zum Beispiel zur Beschreibung von Verbindungspunkten, weitere Arten von *Advice* und eine "auf Anforderung" generierte Laufzeitunterstützung, um im *Advice*-Code weitere Kontextinformationen vom aktuellen Verbindungspunkt zu erhalten. Details sind dem Referenzhandbuch zu entnehmen.

Obwohl AspectC++ nach dem "Vorbild" AspectJ entworfen wurde, sind im Laufe der Arbeit auch viele neue Konzepte, z.B. der Ordnungsmechanismus und ein erweitertes *Joinpoint*-Modell, entstanden, die über die Fähigkeiten von AspectJ hinausgehen. Die Implementierung des Übersetzers umfasst wegen der Komplexität der Sprache C++ etwa 80000 Zeilen C++ Code.

5 Fallstudien

Um die praktische Umsetzbarkeit der Konzepte, die Funktionstüchtigkeit der Werkzeuge und den generellen Nutzen des in der Dissertation verfolgten Ansatzes nachzuweisen, wurden im Rahmen der Arbeit mehrere Fallstudien durchgeführt, von denen zwei im Folgenden kurz vorgestellt werden.

Unterbrechungssynchronisation

Die erste Fallstudie stellt die aspektorientierte Implementierung der Unterbrechungssynchronisation in PURE dar, die ursprünglich noch herkömmlich implementiert vorlag. Es handelt sich dabei um ein *Crosscutting Concern*, das mehrere Subsysteme übergreift. Eine Analyse der ursprünglichen Quelltexte zeigte, dass es 166 Aufrufe der Synchronisationprimitiven in PURE gab, die sich auf 15 Klassen verteilten. Abbildung 3 zeigt einen vereinfachten Ausschnitt aus der Klassenhierarchie von PURE, die 8 dieser 15 Klassen

enthält. Sie sind durch eine dunklere Unterlegung markiert. Hier zeigt sich, wie viele verschiedene Klassen in unterschiedlichen Subsystemen (*Osek*, *Thread* und *Case*) von der Unterbrechungssynchronisation betroffen sind.

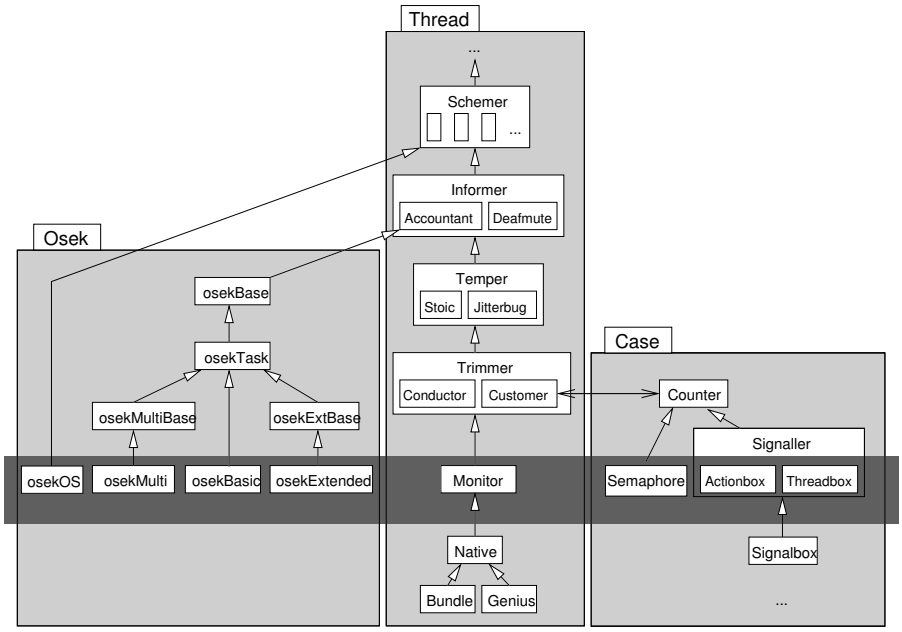

Abbildung 3: Zugriffe auf die globale Sperrvariable in PURE

Eine Analyse der vorliegenden Implementierung ergab zahlreiche Schwächen hinsichtlich der Änder- und Erweiterbarkeit. Insbesondere sind die Synchronisationspunkte praktisch unumstößlich in der Implementierung verankert. Der Übergang zu einer anderen Unterbrechungsgranularität wäre so mit massiven fehlerträchtigen manuellen Eingriffen verbunden.

Abbildung 4 zeigt nun die Implementierung eines Aspekts, der an denselben Punkten für Unterbrechungssynchronisation sorgt, auch wenn in den betroffenen Klassen kein einziger Aufruf der Primitiven mehr direkt zu finden ist. Die Idee hinter dieser Implementierung ist, zunächst alle Klassen zu beschreiben, die zur zu synchronisierenden Region gehören (*Pointcut* `kernel`). Der *Pointcut* `locked` beschreibt nun darauf aufbauend die Menge aller Aufrufe in diesen Bereich hinein, die sich selbst nicht in dem Bereich befinden.

Durch den Austausch dieses Aspekts durch einen anderen ließe sich nun vergleichsweise leicht eine andere Synchronisationgranularität implementieren. Messungen der Codegröße haben darüber hinaus ergeben, dass es zu keinem sprachbedingten zusätzlichen Ressourcenverbrauch kam. Außerdem wurde bei den Implementierungsarbeiten festgestellt, dass an bestimmten Punkten in der ursprünglichen Implementierung fehlerhafterweise keine Synchronisation erfolgte. Die Wahrscheinlichkeit solcher Fehler ist durch die zentrale Beschreibung der Punkte auf einem nun höheren Abstraktionsniveau geringer geworden.

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

// Aspekt zur grobgranularen Unterbrechungssynchr.
aspect IntSync2 {
  // Eintritte in die zu synchronisierende Region
  pointcut kernel() = osek() || thread() || ca-
se() /* || ... */;
  pointcut locked() = call(kernel()) && !within(kernel());
  advice locked() : before() { lock.enter (); }
  advice locked() : after()  { lock.leave (); }
};
```

Abbildung 4: Ein Aspekt zur Unterbrechungssynchronisation (vereinfacht)

Fadensynchronisation in Gerätetreibern

Mit der zweiten Fallstudie über Fadensynchronisation beim Gerätetreiberzugriff wurde gezeigt, welchen Einfluss Aspekte auf die Architektur eines Betriebssystems haben können. Gerätetreiber verwalten Zustandsinformationen, deren unkontrollierte nebenläufige Manipulation zu Fehlern führen könnte, weshalb Synchronisationscode unvermeidbar ist. In einer Betriebssystemfamilie sollte dieser jedoch von den eigentlichen Treiber separiert werden, da es zum Beispiel einfädige Systemvarianten geben könnte, bei denen jeglicher Synchronisationscode Ressourcenverschwendung wäre. Aber auch im vielfädigen Betrieb kann die Synchronisation unter Umständen unnötig sein. Etwa dann, wenn ein Treiber in dieser Konfiguration nur indirekt durch einen anderen Treiber angesprochen wird, bei dem bereits für gegenseitigen Ausschluss gesorgt wurde.

```
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 (); }
};
```

Abbildung 5: Ein wiederverwendbarer Aspekt zum gegenseitigen Aussschluss

Um den Synchronisationscode von den Treibern zu trennen, wurden zunächst verschiedene wiederverwendbare Synchronisationsaspekte, wie der in Abbildung 5, erstellt. Dieser Aspekt sorgt durch Aufrufe der Semaphoroperationen `wait()` und `signal()` vor und nach dem Erreichen bestimmter Verbindungspunkte für gegenseitigen Ausschluss. Dabei wird lediglich der Synchronisationsmechanismus beschrieben. Die konkreten Synchronisationspunkte müssen in Form des virtuellen *Pointcuts* `funcs` durch einen abgeleiteten Aspekt später festgelegt werden. Mit Hilfe dieser AspectC++ Spracheigenschaft konnte ei-

ne ganze Bibliothek wiederverwendbarer Synchronisationsaspekte umgesetzt werden. So wurden auch Aspekte zur Lösung des ersten Leser-/Schreiber Problems sowie Varianten davon, die nicht auf Semaphoren basieren, implementiert. Diese alternativen Synchronisationsaspekte nutzen blockierende Kommunikationsprimitiven. So erzeugt der entsprechende Aspekt für gegenseitigen Ausschluss zum Beispiel einen *Server*-Faden, der einer zu synchronisierenden Region zugeordnet wird. Alle Aufrufe von Methoden der Klassen dieser Region werden durch den Aspekt in Nachrichten umgewandelt, die an den Faden geschickt werden. In der Nachrichtenwarteschlange dieses Fadens werden die Aufrufe sequentialisiert und der Reihe nach abgearbeitet. Die Auswahl der Synchronisationsvariante und der Synchronisationspunkte erfolgte bei der durchgeführten aspektorientierten Implementierung an einer zentralen Stelle, an der Konfigurationswissen vorhanden war, so dass Ressourcenverschwendung in allen Konfigurationen vermieden werden konnte.

Messungen der Codegröße und der Laufzeiten zeigten einen deutlich größeren Ressourcenverbrauch bei den Systemvarianten, bei denen die Synchronisation auf blockierenden Kommunikationsprimitiven basierte. Dennoch haben solche Konfigurationen ihre Berechtigung. Zum Beispiel könnten ergänzende Robustheitseigenschaften wie Adressraumtrennung eine fadenorientierte Architektur notwendig machen. Diese Überlegungen zeigen die enge Verwandtschaft zwischen der Betriebssystemarchitektur (z.B. Mikrokern oder Monolith) zu globalen *Crosscutting Concerns* im Betriebssystem.

6 Verwandte Arbeiten

Vergleichbare Arbeiten sind bisher noch wenig zu finden. So wurden einige Experimente mit AOP am *Prefetching* und der NFS Implementierung des FreeBSD Kerns durchgeführt [CKFS01, CKF+01]. Die Stoßrichtung dieser Arbeiten waren Vielzweckbetriebssysteme. Mögliche positive Folgen, die das Konfigurieren von Aspekt- oder Komponentencode in einer Betriebssystemfamilie mit sich bringen könnte, wurden außer Acht gelassen.

Einige weitere Arbeiten [NCBE00] stellen ein Rahmenwerk für die aspektorientierte Betriebssystementwicklung vor, das auf dem “Aspektmoderator” Konzept beruht und ohne spezielle Sprachunterstützung auskommt. Zwar unterstützt dieser Ansatz auch das Hinzufügen von Aspekten zur Laufzeit, doch lässt der Moderatoransatz erhebliche Effizienzeinbußen erwarten, was in den Papieren nicht durch Messungen widerlegt wurde.

Eine Vorgängerarbeit aus dem Umfeld der Arbeitsgruppe des Autors beschreibt die anwendungsorientierten Entwurfsprinzipien der EPOS Betriebssystemfamilie [Frö01]. Obwohl der Schwerpunkt dieser Arbeit eher im Bereich der Systemanpassung und dem Entwurf generativer Komponenten liegt, wurde dort bereits die Nützlichkeit der aspektorientierten Programmierung für Betriebssystemfamilien erkannt. Zur Implementierung der Aspekte wurde aus Mangel an echter sprachlicher Unterstützung C++ *Template*-Metaprogrammierung eingesetzt.

Der Gedanke Betriebssysteme als Familien bzw. Produktlinien zu bauen, wurde auch im GeneSys Projekt (Universität Kaiserslautern) verfolgt [Bau99]. AOP kam dort allerdings noch nicht zum Einsatz.

7 Fazit

Mit der hier beschriebenen Arbeit wurden erste Erfahrungen mit dem Einsatz der aspektorientierten Programmierung beim Bau von Betriebssystemfamilien gewonnen. Die durchgeführten Fallstudien zeigen neben der Tragfähigkeit des Ansatzes und der Funktionstüchtigkeit der entwickelten Werkzeuge interessante Perspektiven auf. So deutet sich an, dass die Menge der globalen *Crosscutting Concerns* in einem Betriebssystem nichts anderes ist, als das, was gewöhnlich als Betriebssystemarchitektur bezeichnet wird. An die Konfigurierbarkeit derartiger Eigenschaften war bisher nicht zu denken. Auch Vielzweck-Betriebssysteme könnten davon profitieren, da ein Weg aufgezeigt wurde, wie sich geänderte Anforderungen an die Systemarchitektur mit vertretbarem Aufwand umsetzen ließen.

Ein wichtiges praktisches Ergebnis ist die Implementierung von AspectC++. Inzwischen sind knapp 100 Personen auf der Benutzerliste eingetragen. Neben der freien Version des Übersetzers ist seit kurzem auch eine kommerzielle Version verfügbar. Der größte Teil der Nutzer gehört zu Firmen aus den Bereichen der eingebetteten und mobilen Systeme. So sind bereits Kontakte zu sehr bekannten Konzernen zustande gekommen. Dies lässt den Schluss zu, dass den durchgeführten Arbeiten ein tatsächlicher Bedarf gegenübersteht.

Literatur

[Bau99] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conf. on Advanced Information Systems Engineering (CAISE-99), 6th Doctoral Consortium*, Heidelberg, Germany, June 1999.

[BGP+99] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[CKF+01] Yvonne Coady, Gregor Kiczales, Michael Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring Operating System Aspects. *Communications of the ACM*, pages 79–82, October 2001.

[CKFS01] Yvonne Coady, Gregor Kiczales, Michael Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Internation Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented Programming. *Communications of the ACM*, pages 29–32, October 2001.

[Frö01] Antônio Augusto Medeiros Fröhlich. *Application-Oriented Operating Systems*. Dissertation, Technische Universität Berlin, 2001.

[FSH+01] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A Survey of Configurable Component-based Operating Systems for Embedded Applications. *IEEE Micro*, 21(3), June 2001.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugonin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, pages 59–65, October 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[NCBE00] Paniti Netinant, Constantinos A. Constantinides, Atef Bader, and Tzilla Elrad. Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. In *International Conference on Parallel and Distributed Techniques and Applications (PDPTA 2000)*, Las Vegas, Nevada, June 2000. special session on Aspect-Oriented Programming.

[Par76] D. L. Parnas. Some Hypothesis About the Uses Hierarchy for Operating Systems. Technical report, Technische Hochschule Darmstadt, Fachbereich Informatik, 1976.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, Sydney, Australia, February 2002.

[Tur02] Jim Turley. Embedded Processors. *PC Magazine*, January 2002.

Über den Autor

Olaf Spinczyk wurde im Jahr 1970 in Berlin geboren. Nach dem Abitur und einer Berufsausbildung zum Informatikassistenten bei der Schering AG in Berlin studierte er Informatik an der TU Berlin. Während dieser Zeit arbeitete er halbtags als Informatikassistent bei der Schering AG im VAX/VMS Systemmanagement und an einer Datenbankapplikation. Studienschwerpunkte waren Rechnerentwurf- und Architektur sowie Prozessdatenverarbeitung und Robotik. 1996 beendete er sein Studium mit einer Diplomarbeit bei Prof. W. Giloi, die den Titel "Informationsverbreitung in einem Parallelrechner" hatte. Die Arbeit wurde am damaligen Institut GMD FIRST in Berlin durchgeführt.

Ab 1997 war er Mitarbeiter von Prof. W. Schröder-Preikschat an der Otto-von-Guericke-Universität Magdeburg und half, den dortigen Lehrstuhl für Betriebssysteme und Verteilte Systeme aufzubauen. Seine Forschungsarbeiten adressierten die Bereiche aspektorientierte Programmierung, Betriebssysteme, Programmfamilien und Entwurfsmuster. Nach der Promotion im Jahr 2002 ist der Autor mit Prof. Schröder-Preikschat an die Universität Erlangen-Nürnberg gewechselt und dort heute als wissenschaftlicher Assistent tätig.