Rekonfiguration durch dynamische aspektorientierte Programmierung

Martin Gumbrecht
FAU Erlangen-Nürnberg
martin.gumbrecht@gmail.com

ABSTRACT

Softwaresysteme benötigen oft eine stetige Weiterentwicklung sowie Fehlerbehebungen. Oft soll dabei jedoch ein Neustart vermieden werden oder ist sogar völlig unmöglich. Dynamische aspektorientierte Programmierung hat sich dafür als geeignetes Konzept herausgestellt, mit dem Anwendungsfälle vom Nachladen einzelner Funktionen bis zum Austausch ganzer Komponenten gelöst werden können. Dabei gibt es sehr unterschiedliche Realisierungsansätze, die zum Teil plattformspezifische Entwicklungswerkzeuge benötigen oder aber umfangreiche zusätzliche Ressourcen in der Ausführung erfordern. Auch die Integration in bestehende Softwaresysteme ist nicht immer trivial zu lösen.
Es konnte festgestellt werden, dass noch keine universelle Lösung für alle Anwendungsfälle existiert, aber dennoch auch bei speziellen Anforderungen, wie bei Betriebssystemen, eingebetteten Systemen oder mehrfädigen Anwendungssystemen eine geeignete Implementierung möglich ist.

1. EINLEITUNG

Die Änderung und Anpassung ist ein integraler Bestandteil im Entwicklungsprozess von Softwaresystemen. Dies beinhaltet Fehlerbehebungen, Sicherheitsaktualisierungen sowie die Integration neuer Funktionen zur Anpassung an geänderte Nutzungsszenarien. Dabei kann es vorkommen, dass ein Neustart z. B. aufgrund von Verfügbarkeitsanforderungen gar nicht oder nicht zeitnah erfolgen kann. In diesem Fall ist es erforderlich das System zur Laufzeit zu aktualisieren.
Dynamische aspektorientierte Programmierung bietet Konzepte zum Nachladen und Einbinden von Code und ist damit prinzipiell geeignet dieses Problem zu lösen. Je nach Art des Softwaresystems z. B. für eingebettete Systeme, Betriebssysteme oder mehrfädige Anwendungssysteme sind dabei jedoch unterschiedliche Anforderungen zu erfüllen, die verschiedene Techniken und Implementierungen verlangen.
In diesem Artikel sollen die Konzepte der dynamischen aspektorientierten Programmierung zur Rekonfiguration von Softwaresystemen untersucht und eingeordnet werden. Dazu wird in Kapitel 2 die statische aspektorientierte Programmierung analysiert. In Kapitel 3 werden die Konzepte der dynamischen aspektorientierten Programmierung untersucht und anschließend mehrere domänenspezifische Implementierungen analysiert. In Kapitel 5 werden die Ansätze weitergehend evaluiert und bewertet.

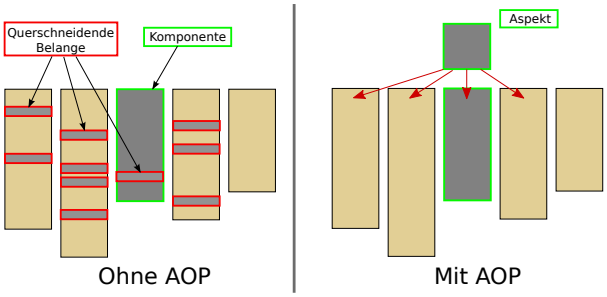

Abbildung 1: Modularisierung durch AOP

2. ASPEKTORIENTIERTE PROGRAMMIERUNG

Die prozedurale und objektorientierte Programmierung bietet mit Funktionen und Klassen Konzepte zur Kapselung von wiederverwendbaren Codeabschnitten. Einige Funktionen wie z. B. Logging, Tracing, Fehlerbehandlung oder Datenvalidierung müssen dabei an diversen Stellen wiederholt eingefügt werden, was oft zu Problemen u. a. in der Lesbarkeit und Wartbarkeit führt. Durch aspektorientierte Programmierung (AOP) können solche querschneidende Belange als *Aspekt* modularisiert und an definierten Stellen im Code automatisiert eingefügt werden (siehe Abbildung 1) [4].
Die Definition eines Aspekts beinhaltet den *Pointcut*, also die Stellen, an welchen der Aspekt eingefügt werden soll, und den *Advice*, welcher die Funktionalität beinhaltet. Ein konkreter Punkt, an dem Aspekt-Code ausgeführt wird, nennt sich *Join-Point*. Durch den *Weaver*, der z. B. ein Precompiler sein kann, werden die Aspekte schließlich in den Code eingefügt [10].
Pointcuts werden in der Regel durch Ausdrücke definiert, die es, je nach Implementierung, erlauben verschiedene Klassen, Methoden oder Datenstrukturaufrufe als Join-Points festzulegen. Es werden verschiedene Advice-Typen unterschieden. Durch *before*- und *after*-Advices wird zusätzlicher Code vor oder nach einer Methodenausführung eingefügt. *Around*-Advices werden anstelle des Methodenaufrufes ausgeführt, wobei die urspüngliche Methode innerhalb des Advices aufgerufen werden kann [10].
Insgesamt fördert aspektorientierte Programmierung die Modularisierung und damit auch die Wartbarkeit von Softwaresystemen, da Redundanz von Code vermieden wird und somit Änderungen an einer zentralen Stelle durchgeführt werden können.

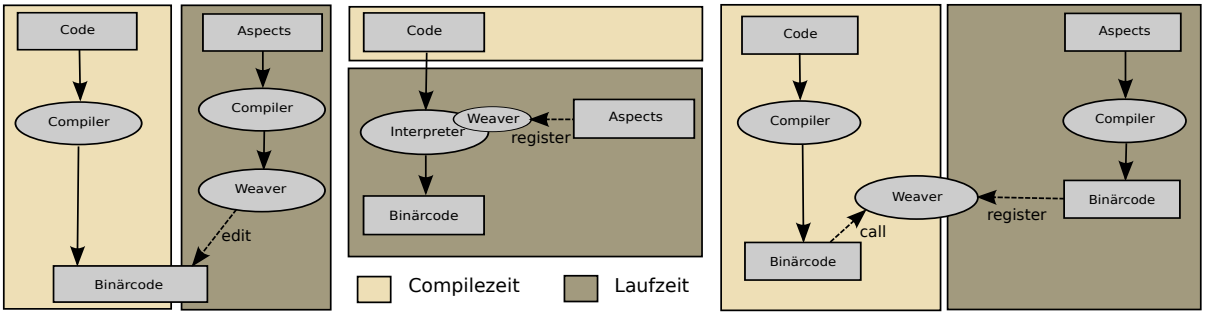

Abbildung 2: Dynamischer Weaver Links: Binärcodebasiert, Mitte: Interpreterbasiert, Rechts: Quellcode-Instrumentierung

3. DYNAMISCHE ASPEKTORIENTIERTE PROGRAMMIERUNG

In der statischen aspektorientierten Programmierung müssen alle Aspekte zur Compilezeit vorliegen und werden fest in den Code eingewebt. Dies ermöglicht eine effiziente Ausführung ohne nennenswerten zusätzlichen Speicher- oder Zeitbedarf [2]. Mit der dynamischen aspektorientierten Programmierung wird es möglich, das ein Einbinden der Aspekte bis zur Laufzeit zu verzögern.
Für dynamische Softwareupdates ergeben sich dadurch zwei Nutzungsszenarien.

- Es können weitere querschneidende Belange implementiert werden und damit zusätzliche Funktionalität hinzugefügt werden, die eine Vielzahl von Join-Points betrifft.
- Es können konkrete Methoden ersetzt werden, indem durch einen Around-Advice die Ausführung verhindert und stattdessen auf eine alternative Implementierung umgeleitet wird.

Ein dynamisches AOP-System besteht im Wesentlichen aus einem dynamischen Weaver, der das Nachladen und Einfügen des Aspekt-Codes übernimmt. Dieser kann unter der Verwendung von drei Konzepten implementiert werden, die im Folgenden analysiert und verglichen werden.

3.1 Binärcodemanipulation

Ein Ansatz zur Realisierung von dynamischer AOP ist die nachträgliche Manipulation des Binärcodes, so dass dieser um entsprechende Aspekte erweitert wird. Dabei werden *call*-Instruktionen so überschrieben, dass stattdessen in neuen, nachgeladenen Code gesprungen wird. Dort werden die entsprechenden Advices ausgeführt und die eigentliche Funktion aufgerufen. Anschließend wird der Rücksprung in den Aufrufer durchgeführt [3].
Zum Auffinden der Join-Points werden dazu Symboltabellen und Debugginginformationen genutzt. Während des Überschreibens müssen inkonsistente Zustände durch Synchronisation ausgeschlossen werden [1].
Durch die Verwendung von Symboltabellen und direkter Änderungen des Binärcodes ist dieses Konzept compiler- und plattformabhängig, ermöglicht allerdings eine sehr zeit- und speichereffiziente Umsetzung. Sowohl Join-Points als auch Advices können zur Laufzeit definiert werden. Für objektorientierte Programmiersprachen ist dieser Ansatz in der Regel jedoch nicht realisierbar, da Optimierungen wie Code-Inlining oder Symbol-Stripping in gängigen Compilern nicht abgeschaltet werden können [2].

3.2 Interpreterbasiert

Bei interpretierten Programmiersprachen kann das Einbinden von Aspekten naturgemäß bis zur Laufzeit verzögert werden. Eine angepasste Laufzeitumgebung ermöglicht es, Join-Points dynamisch zu identifizieren und nachgeladenen Aspekt-Code auszuführen [3].
Insbesondere bei Bytecode-Interpretern, wie beispielsweise der JVM, kann es notwendig sein, das Programm im Debugging-Modus auszuführen um Optimierungen zu unterbinden. Dies kann zu einem beträchtlichem zusätzlichem Ressourcenbedarf führen. Außerdem ist es erforderlich, dass die angepasste Laufzeitumgebung für die verwendete Plattform verfügbar ist [5]. Oft scheidet die Verwendung von interpreterbasierten Sprachen auf Systemen mit eingeschränkten Ressourcen sowieso aus.

3.3 Quellcode-Instrumentierung

Durch das Festlegen von potentiellen Join-Points zur Compilezeit kann an dieser Stelle ein Aufruf an ein Unterprogramm erfolgen, welches den Punkt auf eventuell nachgeladene Aspekte überprüft und diese ausführt. Dies kann durch zwei Techniken erreicht werden.

- Durch einen Precompiler kann an definierten Punkten generierter Code eingefügt werden.
- Es werden Proxy-Objekte erzeugt, die vor bzw. nach den Methodenaufrufen Aspekt-Code ausführen.

In beiden Varianten wird an fest definierten Stellen ein entsprechendes Modul aufgerufen, das für den aktuellen Join-Point entsprechende Aspekte aufruft und bei dem zur Laufzeit nachgeladene Aspekte registriert werden können [8][2].
Durch diesen Ansatz entsteht potentiell ein großer zusätzlicher Zeitbedarf, da um das System ausreichend flexibel gestalten zu können eine Vielzahl von Join-Points definiert werden muss, die eventuell ungenutzt bleiben. Da der Weaver jedoch als Bibliothek vorliegt, ist eine hardwareneutrale Lösung möglich.

3.4 Vergleich und Einordnung

Es hat sich gezeigt, dass Binärcodemanipulation ein effizientes Konzept ist, das jedoch auf bestimmte Programmiersprachen eingeschränkt ist und an die Existenz von plattformspezifischen Entwicklungswerkzeugen gebunden ist.
Bei interpreterbasierten Verfahren ist der Weaver fest in die Laufzeitumgebung eingebunden, was auch hier spezielle Tools erforderlich macht. Im Allgemeinen ist dabei von einem hohen Ressourcenverbrauch auszugehen, was vor allem für eingebettete Systeme kritisch sein kann.
Bei der Quellcode-Instrumentierung liegt der Weaver dagegen in einer Bibliothek vor, was prinzipiell das Übersetzen für jede Architektur ermöglicht. Die Performanz ist dabei stark abhängig von der jeweiligen Implementierung. Nachteilig ist, dass Join-Points zur Compilezeit festgelegt werden müssen.
Abbildung 2 zeigt die unterschiedliche Realisierung der Weaver sowie die Aufteilung der Aktivitäten auf Compile- und Laufzeit.

4. IMPLEMENTIERUNGEN

Es existieren diverse Implementierungen der oben genannten Konzepte. Diese sind in der Regel domänenspezifisch und unterscheiden sich im Umfang ihrer Funktionalität. In diesem Kapitel werden drei Frameworks analysiert, die einen Rekonfigurationsmechanismus für Betriebssysteme, eingebettete Systeme und mehrfädige Anwendungsprogramme implementieren.

4.1 Betriebssysteme

Chen, Su und Chou stellen in ihrem Artikel[1] einen Ansatz zur dynamischen Aktualisierung von Betriebssystemen vor. Er nutzt den Ansatz der Binärcodemanipulation und ist beispielhaft für Linux implementiert worden.

4.1.1 Anforderungen

Bei Betriebssystemen ist, z. B. im Serverbetrieb, ein Neustart des Systems zur Rekonfiguration oft unerwünscht. Um kritische Fehler zu beseitigen wird daher ein Mechanismus benötigt, welcher das Austauschen und Ändern von Funktionen zur Laufzeit unterstützt. Dabei sollen die Advice-Typen *before*, *after* und *around* unterstützt werden und es soll möglich sein, mehrere Advices pro Join-Point registrieren zu können. Damit wird das Ergänzen von Funktionen sowie deren kompletter Austausch möglich. Das Framework benutzt das Konzept der Kernel-Module zum Nachladen von Programmcode [1].

4.1.2 Architektur

Das Framework besteht aus drei Komponenten:

- Der *Aspect Maker* ist ein Werkzeug zur Entwicklung und Übersetzung der Advices.
- Der *Aspect Registrar* verwaltet die Pointcuts mit den zugehörigen Advices.
- Der *Aspect Manager* wird zur Laufzeit an einem Join-Point angesprochen. Er identifiziert die entsprechenden Advices und ruft diese auf.

4.1.3 Umsetzung

Aspekt-Maker
Ein Aspekt wird in einem C-Dialekt geschrieben und beinhaltet den Advice-Typ, den Advice-Code sowie den Pointcut (siehe Listing 1). Der Aspekt wird schließlich zu einem Kernel-Modul übersetzt (siehe Listing 2). Dabei wird die Adresse des Pointcuts beim Übersetzen in der Symboltabelle nachgeschlagen (Zeile 1). Beim Laden des Kernel-Moduls wird der Advice beim Aspect-Registrar angemeldet (Zeile 7) und im Binärcode an der entsprechenden Stelle ein Aufruf des Aspekt-Managers eingefügt (Zeile 8). Dazu wird an der nachgeschlagenen Adresse die erste Anweisung mit einem Aufruf des Aspekt-Managers ersetzt. Diese wird später vom Aspekt-Manager selbst ausgeführt. Beim Entladen werden diese Schritte wieder rückgängig gemacht [1].

```
1  aspect access{
2    before():execution("sys_mkdir");{
3      printk("this is a before-advice\n");
4    }
5  }
```

Listing 1: Aspekt-Code

```
1  #define ADDRESS 0xdeadbeef
2  #define TYPE before
3  void advice (void){
4    printk("this is a before-advice\n");
5  }
6  int init_module(){
7    register(advice, TYPE, ADDRESS);
8    weave(advice);
9  }
10 void cleanup_module(){
11   unweave(advice);
12   unregister(advice, TYPE, ADDRESS);
13 }
```

Listing 2: Übersetzter Aspekt-Code (vereinfacht)

Aspekt-Registrar
Der Aspekt-Registrar verwaltet alle Pointcuts als Liste, wobei jeder Pointcut selbst eine Datenstruktur ist, die alle entsprechenden Advices enthält. Zu jedem Advice kann eine Prioriät angegeben werden, welche die Reihenfolge beeinflusst. Beim Registrieren eines Advices wird der Pointcut in der Liste gesucht und, wenn noch nicht vorhanden, eingefügt. Unter dem Pointcut wird schließlich der Advice entsprechend der Priorität angehängt [1].

Aspekt-Manager
Bei einem Aufruf einer Funktion, die mit einem Advice versehen worden ist, erfolgt ein Aufruf des Aspect-Managers, der den Pointcut in der Liste identifiziert und entsprechende Advices aufruft (siehe Abbildung 3). Vor dem Aufruf der eigentlichen Funktion wird die ersetzte erste Instruktion vom Aspektmanager selbst ausgeführt und an die Stelle der zweiten Anweisung gesprungen. Nach der Rückkehr in den Aspekt-Manager werden eventuelle After-Advices aufgerufen und zum Schluss direkt in den ursprünglichen Aufrufer zurrückgesprungen [1].

4.1.4 Analyse

Die beschriebene Implementierung kann einen Linux-Kernel ohne weitere Anpassungen zur Laufzeit erweitern. Lediglich

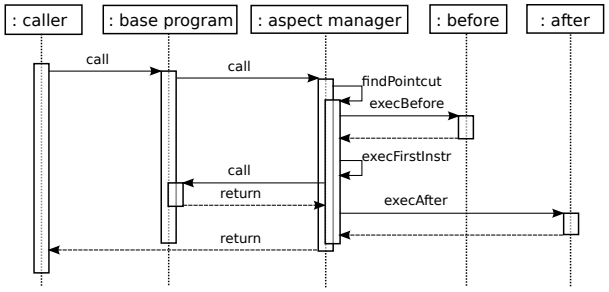

Abbildung 3: Aufrufsemantik im Aspekt-Manager

die Unterstützung für Kernel-Module ist Voraussetzung.
Es sind jedoch keine Mechanismen zur Änderung von Datenstrukturen enthalten. Damit eignet sich der Ansatz lediglich für kleinere Fehlerbehebungen oder Erweiterungen, jedoch nicht für umfassende Änderungen an bestehender Software.
Die Autoren erwähnen richtigerweise, dass der Schreibzugriff auf den Binärcode geschützt werden muss, stellen aber keine Implementierung vor.
Ebenso fehlt eine Analyse des Zeitbedarfs beim Aufruf des Aspekt-Managers. Es ist jedoch davon auszugehen, dass dieser linear mit der Anzahl der registrierten Pointcuts wächst, da diese als Liste verwaltet werden. Andere Implementierungen benötigen hierfür lediglich $\mathcal{O}(logN)$ durch Verwendung eines balancierten Baumes.
Im Gegensatz zu anderen Ansätzen werden hier Pointcuts statt Join-Points verwaltet, was die Größe der Datenstruktur stark verringert, da ein Pointcut in der Regel aus einer Vielzahl von Join-Points besteht. Aus dem Artikel geht nicht hervor, ob in der beschriebenen Implementierung die Definition eines Pointcuts mehrere Funktionen betreffen kann. Wenn Pointcut A beispielsweise Funktion F betrifft und Pointcut B die Funktionen F und G, müssen beim Betreten von F alle entsprechenden Pointcuts (hier A und B) gesucht werden und somit in jedem Fall durch die komplette Liste iteriert werden. Dies würde die oben genannte effizientere Implementierung unmöglich machen.

4.2 Eingebettete Systeme

Gilani u. a. stellen in ihrem Artikel[2] eine Erweiterung des Frameworks AspectC++ um dynamische Aspekte vor. Bei der Implementierung wurde besonders auf die Eignung für eingebettete Systeme Wert gelegt.

4.2.1 Anforderungen

Eingebettete Systeme müssen oft mit sehr knappen Ressourcen auskommen. Nichtsdestotrotz kann auch dabei eine Rekonfiguration benötigt werden um das System weiterzuentwickeln. Dafür wurde eine Vorgehensweise entworfen, um statische und dynamische Aspekte gleichartig entwickeln zu können. Dadurch kann die Entscheidung, ob ein Aspekt dynamisch oder statisch eingebunden wird, beim Deployment abhängig von den verfügbaren Ressourcen getroffen werden. Dies ermöglicht einerseits Funktionen wie z. B. Tracing lediglich im Fehlerfall zu aktivieren, aber auch neu entwickelte Aspekte zur Laufzeit nachzuladen. Der Aufruf von dynamischen Aspekten soll dabei so effizient wie möglich implementiert werden um den Einsatz in eingebetteten System zu ermöglichen [2].

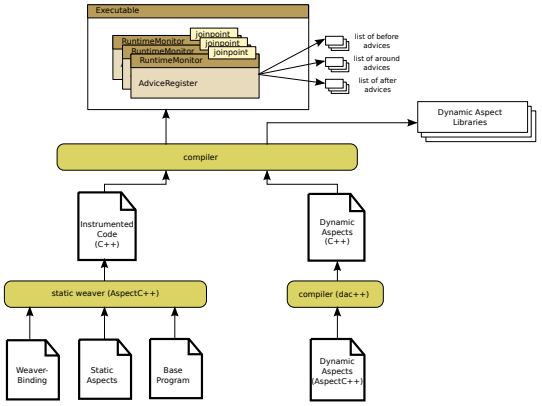

Abbildung 4: Architektur von Dynamic AspectC++

4.2.2 Architektur

Das Framework besteht aus drei Teilen, dem *Weaver-Binding*, dem *Runtime-Monitor* und einer Build-Umgebung für dynamische Aspekte (siehe Abbildung 4). Der Weaver-Binding ruft durch Quellcode-Instumentierung an definierten Stellen den Runtime-Manager auf, welcher die dyanmischen Aspekte verwaltet. Um den Zeitbedarf konstant zu halten, wird für jeden Join-Point ein eigener Runtime-Monitor verwendet. Das Framework ist in der Funktionalität und den unterstützten AOP-Features, wie beispielsweise den Advice-Typen, so konfigurierbar, dass eine möglichst kleine, anwendungsspezifische Variante zusammengestellt werden kann [2].

4.2.3 Umsetzung

Weaver-Binding
Zur Instrumentierung wird ein statischer Aspekt verwendet. Listing 3 zeigt dies am Beispiel des Before-Advices. Der Pointcut wird pure-virtual definiert und kann bei Verwendung mit eigenen Werten überschrieben werden (Zeile 2). Dadurch wird definiert, an welchen Stellen dynamische Aspekte möglich sind.
Im Advice werden Informationen wie übergebene Parameter extrahiert (Zeile 5, 6), die in Aspekten verwendet werden können. In Zeile 7 wird schließlich, anhand der Join-Point-Id, der spezifische Runtime-Manager aufgerufen, der zur Laufzeit dynamische Advices ausführt.
Diese Implementierung verursacht einen zusätzlichen Speicherbedarf von lediglich 12 Byte pro Join-Point. Die mächtige Expression-Sprache von AspektC++ ermöglicht es, Pointcuts flexibel zu konfigurieren, so dass die Anzahl der nicht genutzten Join-Points und damit der Ressourcenverbrauch reduziert werden kann [3][2].

```
1  aspect instrument{
2    pointcut virtual dynamicJPS = 0;
3    public:
4      dynamicJPS():before(){
5        ArgsJnPnt<JoinPoint::ARGS> jp;
6        jp.joinpointName = JoinPoint::signature();
7        monitor<JoinPoint::JPID>::BeforeAdvice(&jp);
8    }
9  };
```

Listing 3: Intrumentationsaspekt (vereinfacht)

Runtime-Monitor
Mit Hilfe von Templates unter der Verwendung der eindeutigen Join-Point-Id wird für jeden Join-Point ein eigener Runtime-Monitor erzeugt, in welchem lediglich die konkret zugehörigen Advices verwaltet werden.
Durch diese Implementierung wird konstante Zeit zum Auffinden der Advices benötigt, im Gegensatz zu $\mathcal{O}(logN)$ bei einer zentralen Instanz [2].

Dynamic AspectC++ Compiler
Wie bereits erwähnt, wird sowohl für statische als auch für dynamische Aspekte die C++-Erweiterung AspectC++ verwendet. Aus dem dynamischen Aspect-Code wird mit einem speziellen Compiler (dac++) gewöhnlicher C++-Code erzeugt, welcher wiederum zu Bibliotheken übersetzt wird, die zur Laufzeit nachgeladen werden können. Diese werden schließlich durch den Runtime-Monitor ausgeführt [9].

4.2.4 Analyse
Die beschriebene Implementierung bietet eine umfangreiche Umsetzung der dynamischen AOP für C++. Durch geringe Anpassungen zur Compilezeit kann dabei auch ein bestehendes System um dynamische Rekonfigurationsmöglichkeiten erweitert werden. Insbesondere bei AspectC++-Anwendungen können bestehende Aspekte ohne nennenswerten zusätzlichen Programmieraufwand als dynamische Aspekte deployed werden.
Das Framework wurde umfassend auf niedrigen Ressourcenbedarf hin optimiert. Dabei konnte pro dynamischem Join-Point ein zusätzlicher Speicherverbrauch von lediglich 12 Byte und ein konstanter Zeitbedarf erreicht werden.
Zur weiteren Analyse wurde das Betriebssystem eCos mit dem oben beschriebenen Framework erweitert und eine dynamischer Aspekt beispielhaft implementiert. Dabei konnte festgestellt werden, das der benötigte RAM und ROM bei optimal definierten Pointcuts nur geringfügig zunimmt, während bei flexiblerer Pointcut-Definition, wie sie für weitreichende Rekonfigurationsmöglichkeiten benötigt wird, ein relevanter zusätzlicher Ressourcenverbrauch zu verzeichnen ist. Dieses Resultat konnte bei Laufzeitmessungen ebenfalls bestätigt werden [2].
Insgesamt lässt sich damit schließen, dass der Ansatz für eingebettete Systeme geeignet ist, jedoch flexible Rekonfigurationsmöglichkeiten durch zusätzliche Ressourcen erkauft werden müssen.

4.3 Mehrfädige komponentenbasierte Anwendungen
In ihrem Artikel[7] beschreiben Rasche, Schult und Polze die dynamische Rekonfiguration von mehrfädigen Anwendungssystemen mithilfe des selbstenwickelten Frameworks Rapier LOOM.NET. Sie nutzen dabei die aspektorientierte Programmierung zur Weiterentwicklung von Komponenten in objektorientierten Softwaresystemen.

4.3.1 Anforderungen
Viele moderne Anwendungssysteme, wie beispielsweise Enterprise-Anwendungen, unterliegen einer stetigen Weiterentwicklung zur Anpassung an neue Nutzungsszenarien. Sie laufen dabei oft als Client-Server-Systeme, wobei jeder Ausfall des Servers während einer Rekonfiguration hohe Kosten verursachen kann. Ein Ansatz für dynamische Softwareupdates muss also das Austauschen, Hinzufügen und Entfernen von Komponenten erlauben. Beim Austausch muss dabei auch ein Zustandstransfer zwischen alter und neuer Instanz vorgenommen werden. Eine Komponente wird dabei definiert als eine Klasse mit öffentlicher Schnittstelle. Lediglich diese Schnittstelle darf bei der Aktualisierung nicht verändert werden.
Insbesondere ist für diesen Anwendungsfall auch die Rekonfiguration von mehrfädigen Anwendungen notwendig. Dabei muss bei vielen gleichzeitigen Aufrufen ein geeigneter Rekonfigurationszeitpunkt gefunden werden [7].

4.3.2 Architektur
Komponenten, die rekonfigurierbar gehalten werden sollen, werden mit einem *Rekonfigurationsproxy* versehen. Ein *Rekonfigurationsmanager* erzeugt zur Laufzeit die Instanz einer neuen Komponente, sorgt für den Zustandstransfer und aktualisiert den Proxy.

4.3.3 Umsetzung

Rekonfigurationsproxy
Der Rekonfigurationsaspekt beinhaltet eine Schnittstelle für den Austausch von Komponenten und einen Advice, der Aufrufe an die entsprechende Version der Komponenten weiterleitet (siehe Listing 4).
In Zeile 5 wird dazu ein Around-Advice definiert, wodurch anstelle aller Methodenaufrufe der betroffenen Klassen (Zeile 6) die Proxy-Methode aufgerufen wird. In dieser wird, durch einen Reader-Writer-Lock (s. u.) geschützt, die entsprechende Methode auf der aktuellen Komponentenversion aufgerufen. Falls noch keine Rekonfiguration stattgefunden hat ist dies das Objekt selbst (Zeile 10), im anderen Fall die unter `target` gehaltene Referenz (Zeile 12) [7].
Die Rekonfigurationsmethode setzt, ebenfalls synchronisiert, die Referenz auf die neue Implementierung der Komponente.
Zu beachten ist, dass in die Methoden `Context.Invoke` und `Context.InvokeOn` (Zeile 10, 12) in LOOM.NET nicht zu einer Reflection-API gehören und dieser Aufruf in konstanter Zeit erfolgt [6].
Der Rekonfigurationsaspekt wird in LOOM.NET bei Erzeugung einer Komponente über eine spezielle Factory angehängt. Es ist zwingend erforderlich, diese anstelle des `new`-Operators zu verwenden, wenn eine Klasse rekonfigurierbar werden soll [8].

```
1  public class ReconfAspect:Aspect,IConfigure{
2    private object target;
3    private RWLock rwlock=new RWLock();

5    [Call(Invoke.Instead)]
6    [IncludeAll]
7    public object Proxy(object[] args){
8      rwlock.AcquireReaderLock(-1);
9      if (target == null){
10       return Context.Invoke(args);
11     } else {
12       Context.InvokeOn(target, args);
13     }
14     rwlock.ReleaseLock();
15   }

17   public void ReplaceReference(object target){
18     rwlock.AcquireWriterLock();
```

```
19      this.target = target;
20      rwlock.ReleaseWriterLock();
21    }
22  }
```

Listing 4: Rekonfigurationsaspekt (vereinfacht)

Rekonfigurationsmanager
Der Rekonfigurationsmanager veranlasst die Rekonfiguration einer oder mehrerer Komponenten gemäß einer gegebenen XML-Datei. Dazu wird eine neue Instanz der Komponente erzeugt, der Zustand wird übertragen und schließlich die Referenz im Proxy ausgetauscht. Die Rekonfiguration mehrerer Komponenten erfolgt nacheinander, wobei die Reihenfolge manuell zu bestimmen ist [7].

Bestimmung des Rekonfigurationszeitpunkts
Um inkonsistente Zustände zu verhindern muss der Austausch einer Komponente atomar stattfinden und die betroffene Komponente darf sich dabei nicht in Ausführung befinden. Dies wird durch einen *Reader-Writer-Lock* erreicht, wobei Methodenaufrufe als lesende Zugriffe zählen, während die Rekonfiguration als schreibender Zugriff gilt.
Innerhalb des Proxies wird dazu vor der Weiterleitung des Aufrufs an die Komponente ein Reader-Lock angefordert (siehe Listing 4, Zeile 8). Durch rekursive Locks wird sichergestellt, dass ein Aufrufer, der bereits eine Ausführung begonnen hat, weitere Lesezugriffe gewährt bekommt, wodurch Deadlocks verhindert werden.
Die Rekonfigurationsmethode fordert einen Writer-Lock an (Zeile 18), woraufhin zukünftige lesende Zugriffe blockiert werden. Sobald alle Leser die Komponente verlassen haben, wird der Schreibzugriff gewährt und die Referenz auf die neue Komponente gesetzt, wodurch die eigentliche Rekonfiguration stattfindet [7].

Zustandstransfer zwischen Komponenten
Der Zustandstransfer kann entweder implizit oder explizit vorgenommen werden. Beim impliziten Zustandstransfer werden alle Member-Variablen rekursiv kopiert. Dies erfordert jedoch, dass sich die Datenstrukturen innerhalb der Komponente beim Update nicht geändert haben. Für den expliziten Zustandstransfer muss eine Komponente eine Methode besitzen um ihren Zustand auszugeben bzw. einzulesen [7].

4.3.4 Analyse
Im Gegensatz zu den beiden anderen Frameworks ist in diesem Ansatz das Ziel im Wesentlichen der Austausch von Komponenten anstatt dem Nachrüsten von Aspekten. Dafür ist die dynamische aspektorientierte Programmierung prinzipiell nicht erforderlich, da der benötigte Rekonfigurationsaspekt bereits zur Compilezeit bekannt ist. Zur Laufzeit wird lediglich die Instanz der neuen Komponente erzeugt und im Proxy ausgetauscht.
Das Konzept setzt ein striktes Komponentenmodell mit eindeutigen, unumgänglichen Schnittstellen vorraus. Insbesondere dürfen keine Referenzen auf interne Objekte nach außen gegeben werden, da diese nach dem Aktualisieren nicht mehr gültig sind. Dies ist bereits zum Design-Zeitpunkt zu berücksichtigen, wodurch das nachträgliche Hinzufügen der Rekonfigurationsfunktionalität unter Umständen schwierig werden kann.
Ohne anstehende Rekonfiguration entsteht im Proxy konstanter zusätzlicher Zeitbedarf. Im Falle einer Rekonfiguration werden alle eingehenden Methodenaufrufe der Komponente bis zum Abschluss verzögert. Die Wartezeit hängt dabei von der Anzahl und Laufzeit der noch zu beendenden Aufrufe vor der Rekonfiguration sowie der Zeit zur Zustandsübertragung ab.
Nach einer Rekonfiguration bleibt immer eine ursprüngliche Variante der Komponente erhalten, wodurch viel zusätzlicher Speicherverbrauch entstehen kann. Dies wäre durch eine geschicktere Implementierung möglicherweise zu verhindern gewesen.

5. FAZIT
Es hat sich gezeigt, dass keiner der Ansätze eine universelle Lösung für beliebige Anwendungsfällte der dynamischen Rekonfiguration bietet.
Durch Binärcodemanipulation werden im Wesentlichen kleinere Fehlerbehebungen ermöglicht, ohne dass die Rekonfigurationsmöglichkeiten bereits im Design des Systems vorrausgeplant werden müssen. Für umfassende Weiterentwicklung scheint sie eher ungeeignet. Zudem ist die Umsetzung an bestimmte Programmiersprachen und Hardware gebunden, was die Flexibilität weiter einschränkt.
Die Quellcode-Instrumentierung hat sich als mächtig und vielseitig herausgestellt. Es sind sowohl Umsetzungen für eingebettete Systeme möglich, als auch für große, mehrfädige Anwendungssysteme. Die Effizienz ist dabei stark von der jeweiligen Implementierung abhängig. Aufgrund der statischen Definition der Pointcuts ist hier eine sehr überlegte Vorauswahl zu treffen.
Insgesamt konnte festgestellt werden, dass dynamische aspektorientierte Programmierung zur Rekonfiguration von Softwaresystemen geeignet ist, die verwendeten Techniken und Frameworks jedoch stark auf die entsprechenden Anforderungen abgestimmt werden müssen. Insbesondere sind auch Möglichkeiten gegeben, die weit über das Hinzufügen und Austauschen von Aspekten hinausgehen und sogar das ändern ganzer Komponenten erlauben.

6. LITERATURVERZEICHNIS

[1] J. Chen, H.-M. Su, and C.-F. Chou. An aspect-oriented framework for operating system evolution. Maerz 2010.

[2] W. Gilani, F. Scheler, D. Lohman, O. Spinczyk, and W. Schroeder-Preikschart. Unification of static and dynamic AOP for evolution in embedded software systems. *HICSS '06 Mini-Track on Adaptive and Evolvable Software Systems, IEEE*, Maerz 2007.

[3] W. Gilani and O. Spinczyk. A family of aspect dynamic weavers.

[4] D. Lohmann. Kongigurierbare Systemsoftware. `https://www4.cs.fau.de/Lehre/SS14/V_KSS/Vorlesung/folien/03-AspectC++_handout.pdf`.

[5] A. Popovivi. *PROSE - A Study on Dynamic AOP.* dissertation.de, 2003.

[6] A. Rasche and W. Schult. Dynamic updates of graphical components in the .NET framework. Maerz 2007.

[7] A. Rasche, W. Schult, and A. Polze. Self-adaptive

multithreaded applications - a case for dynamic aspect weaving. November 2005.

[8] W. Schult. LOOM.NET documentation. `https://loom.codeplex.com/documentation`.

[9] R. Tartler, D. Lohmann, W. Schroeder-Preikschart, and O. Spincyzk. Dynamic AspectC++: Generic advice at any time. November 2009.

[10] L. Wunderlich. *AOP - Aspektorientierte Programmierung in der Praxis*. Entwickler Press, 2005.