“Towards Aspectual Component-Based Real-Time System Development”

Seminar Aspektorientierte Systemprogrammierung

Matthias Körber

Inhalt

1. Motivation....3
2. Einführung....4
2.1. Eingebettete Echtzeitsysteme....4
2.2. Komponentenbasierte Softwareentwicklung....6
3. Design-Kriterien....6
3.1. Geheimnisprinzip....6
3.2. Schnittstellen....7
3.3. Komponentensichten....7
3.4. Zeitliche Eigenschaften....7
3.5. Aufgabenzuweisung....7
3.6. Aspekt-Unterstützung....7
3.7. Aspektweben....7
3.8. Mehrere Schnittstellen....7
3.9. Mehrere Aspekttypen....7
3.10. Unterstützung bei der Konfiguration....8
3.11. Zeitliche Analyse....8
4. ACCORD....8
4.1. Klassifikation von Aspekten in Echtzeitsystemen....8
4.2. Real-Time Component Model....8
4.2.1 Funktionaler Teil von RTCOM....9
4.2.2. Laufzeitsystemabhängiger Teil....10
4.3. RTCOM Schnittstellen....11
4.3.1. Funktionale Schnittstellen....12
4.3.2. Konfigurationsschnittstellen und Kompositionsschnittstellen....12
5. Zusammenfassung und Ausblick....13
6. Bibliographie....13

1. Motivation

Echtzeit- und insbesondere eingebettete Systeme finden in vielen Bereichen unseres täglichen Lebens Verwendung. Dabei handelt es sich nicht immer um kritische Systeme von denen Menschenleben abhängen, wie in Flugzeugen oder Autos. Auch Waschmaschinen oder Getränkeautomaten enthalten häufig eingebettete Systeme, die nichtsdestotrotz korrekt und zuverlässig funktionieren müssen.

Zusätzliche Entwicklungsziele, wie niedrige Entwicklungskosten und möglichst kurze Entwicklungszeiten, erschweren diese Aufgabe und erfordern neue Lösungsansätze. Komponentenbasierte Softwareentwicklung ist einer dieser Ansätze. Ein Softwaresystem wird dabei in mehrere Komponenten zerlegt, die lediglich über fest definierte Schnittstellen miteinander kommunizieren. Die Vorteile sind offensichtlich:

- Niedrigere Entwicklungskosten, da Komponenten wiederverwendet werden können.
- Neue Systeme können schnell aus vorhandenen Komponenten zusammengesetzt werden.
- Bessere Wartbarkeit, da Komponenten beliebig ausgetauscht werden können, vorausgesetzt die Schnittstelle ändert sich nicht.

Trotzdem treten auch hier wieder Probleme auf, die sich durch bloße Kapselung in Komponenten nicht zufriedenstellend lösen lassen. Aspektorientierte Softwareentwicklung begegnet diesen Problemen auf effiziente Art und Weise, indem sogenannte “querschneidende Belange”, wie zum Beispiel Synchronisation, in Aspekte gekapselt werden.

Im Bereich der Echtzeitsysteme führt dies allerdings zu zusätzlichen Anforderungen an aspektorientierte und komponentenbasierte Softwareentwicklung, so müssen zum Beispiel zeitliche Vorgaben eingehalten werden. Aspekte und Komponenten müssen also zeitlich vorhersagbar sein, um ihren Einfluß auf das Gesamtsystem einschätzen zu können.

ACCORD (aspectual component-based real-time system development) ist ein Konzept, mit dem genau das erreicht werden soll.

2. Einführung

2.1. Eingebettete Echtzeitsysteme

Eingebettete Systeme sind Computersysteme, die Bestandteil eines größeren technischen Systems sind und stark anwendungsspezifische Aufgaben erledigen. Wenn die korrekte Funktion des Systems nicht nur von der Korrektheit der Berechnungen eines eingebetteten Systems abhängt, sondern auch von dem Zeitpunkt, zu dem diese beendet, bzw. ihre Ergebnisse verfügbar sind, so handelt es sich um ein eingebettetes Echtzeit-System.

Üblicherweise bestehen diese aus nebenläufigen Programmen, auch Tasks genannt, die bestimmte Task-Deadlines einhalten müssen, um eine korrekte Funktion des Gesamtsystems zu gewährleisten. An dieser Stelle läßt sich unterscheiden zwischen harten und weichen Echtzeitsystemen: kann ein Task seine Deadline nicht einhalten, so hat dies in harten Echtzeitsystemen katastrophale Auswirkungen zur Folge, in weichen Echtzeitsystemen jedoch ist höchstens die Performanz beeinflußt.

Diese Tasks müssen nun in eine Reihenfolge gebracht werden, so dass sie ihre Deadlines erfüllen können. Diese Reihenfolge ist ein Ablaufplan, und um diesen aufstellen zu können muß man in der Lage sein, Voraussagen über die CPU-Zeit, die jeder Task benötigt machen zu können. Die Eigenschaft der worst-case-execution-time (WCET) eines Tasks ist hier sehr nützlich, da sie eine obere Grenze für die CPU-Zeit darstellt.

Die WCET eines bestimmten Tasks läßt sich auf unterschiedliche Weise bestimmen, unter anderem durch Analyse des Quelltexts. Bei herkömmlicher Quelltextanalyse ergibt sich die WCET als Konstante. Symbolische WCET-Analyse jedoch berücksichtigt auch den Kontext, in dem der zu untersuchende Code aufgerufen wird.

Zur Erläuterung ein kleines Beispiel. Die `power`-Funktion (siehe Abb. 1) erwartet zwei Parameter `fkn` und `e`. Zunächst wird entschieden, ob `e` positiv oder negativ ist. Dann wird die Potenz der Zahl `fkn` berechnet, indem eine Schleife `e`-mal durchlaufen wird, während der sich `fkn` jedes Mal mit dem aktuellen Zwischenergebnis multipliziert. Abschließend wird der Kehrwert des Ergebnisses gebildet, falls e negativ war. Die rechte Hälfte der Abbildung zeigt ein Flußdiagramm der Funktion inklusive der benötigten CPU-Zeit für jede Anweisung.

Möchte man nun die WCET für diese Funktion berechnen, so geht man folgendermaßen vor: Zunächst legt man den zulässigen Wertebereich für `e` fest, z.B. [-10; 10]. Anschließend zählt man die Zeit die jede Anweisung benötigt zusammen. An Stellen an denen sich das Flußdiagramm verzweigt wählt man das Maximum der jeweiligen Pfade. Für Schleifen multipliziert man die Zeit für Schleifenrumpf und Testen der Abbruchbedingung, mit der Anzahl der maximal möglichen Schleifendurchgänge.

Somit ergibt sich für die `power`-Funktion folgende WCET:

$$WCET_{power} = 50 + 111 + max(301, 583) + 270 + 10(117 + 317) + 99 + max(560, 0) + 244 = 6374$$

```
1  double power(double fkn, int e) {
2      double res = 0;
3      int times = 0;
4      bool pos = true;
5
6      if e < 0 then
7          times = -e;
8      else
9          pos = false;
10         times = e;
11     end if
12     res = 1.0;
13
14     for(i in 1 ... times loop
15         res = res * fkn;
16     end loop
17
18     if(pos)
19         res = 1 / res;
20
21     return res;
22 end power
```

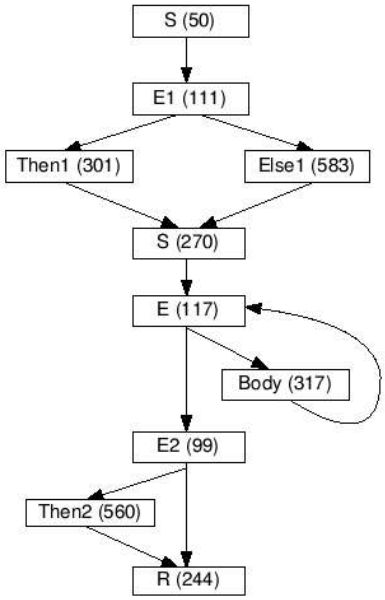

Abb. 1

Dieses Ergebnis ist natürlich viel zu pessimistisch, da der `else`-Zweig (Zeile 8) und die `then`-Anweisung (Zeile 19) nicht beide für das gleiche `e` durchlaufen werden können.

Symbolische WCET-Analyse drückt die WCET abhängig von einem Parameter aus und erlaubt daher genauere Vorhersagen über die maximal benötigte CPU-Zeit der untersuchten Funktion. In Abhängigkeit vom Exponenten `e` lautet die WCET also folgendermaßen:

$$WCET_{power}(e) = 50 + 111 + [e < 0]301 + [e \geq 0]583 + 270 + 117 + \sum_{i=1}^{|e|}(117 + 317) + 99 + [e < 0]560 + [e \geq 0]0 + 244$$

Dies läßt sich vereinfachen zu:

$$WCET_{power} = \begin{cases} 1752 - 434e, e < 0 \\ 1474, e = 0 \\ 1474 + 434e, e > 0 \end{cases}$$

Offensichtlich liegt das Maximum für den angenommenen Wertebereich bei e=-10, woraus eine WCET von 6092 resultiert. Dies ist ein besserer Wert, als der durch traditionelle Analyse errechnete.

2.2. Komponentenbasierte Softwareentwicklung

Traditionelle Softwareentwicklung monolithischer Systeme leidet typischerweise an folgenden Problemen:

- hohe Entwicklungskosten
- unzureichender Support für langlaufende Systeme, bzw. für Änderungen an solchen Systemen
- unbefriedigende Qualität

Flexible Softwaresysteme können mit Hilfe von komponentenbasierter Softwareentwicklung erstellt werden. Hierzu wird ein Softwaresystem aus Komponenten zusammengestellt, die explizit für vielfältige Einsatzmöglichkeiten entwickelt wurden. Dadurch können Kosten und Entwicklungszeit eingespart werden, da auf vorhandene Komponenten zurückgegriffen werden kann. Außerdem können Softwaresysteme leichter erweitert werden, indem einfach neue Komponenten hinzugefügt werden. Schließlich verringern sich auch die Wartungskosten für ein Softwaresystem, da Verbesserungen an einer Komponente allen Systemen, die diese Komponente benutzen zugute kommen.

Clemens Szyperski definierte Softwarekomponenten folgendermaßen:

"A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."[2]

Allgemein läßt sich sagen, dass Komponenten genau definierte Schnittstellen und bestimmte Eigenschaften besitzen. CORBA-Komponenten z.B. sind CORBA-Objekte, mit standardisierten Schnittstellen, die mit Hilfe einer Interface Definition Language beschrieben werden.

Komponenten können mehr als eine Schnittstelle haben, vorstellbar ist z. B. eine Einteilung in Schnittstellen, die zur Verfügung gestellt werden bzw. benötigt werden, damit eine Komponente mit anderen interagieren kann, und eine Schnittstelle zur Konfiguration der Komponente. Normalerweise geht man davon aus, dass diese Schnittstellen die einzige Möglichkeit zum Zugriff auf eine Komponente bieten. Der innere Zustand ist nach aussen nicht sichtbar, verhält sich also wie eine “black box”.

3. Design-Kriterien

An die Entwicklung neuartiger Designmethoden zur Erstellung komponentenbasierter Echtzeitsysteme werden folgende Anforderungen gestellt:

3.1. Geheimnisprinzip

Eine Komponente soll Interna verbergen. Dazu gehören unter anderem Design und Implementation, aber auch die Programmiersprache, in der sie implementiert ist.

Dies dient in erster Linie dazu, den Austausch von Komponenten zu ermöglichen.

3.2. Schnittstellen

Auf eine Komponente kann über mindestens eine definierte Schnittstelle zugegriffen werden. Darüberhinaus soll dies auch der einzige Weg sein, um auf die Komponente zuzugreifen. Wie schon der Punkt Geheimnisprinzip dient dies der Erleichterung beim Austauschen von Komponenten. Wenn sich Schnittstellen über mehrere Versionen einer Komponente nicht ändern, kann eine Komponente durch eine neuere Version ersetzt werden, ohne Änderungen am restlichen System vornehmen zu müssen.

3.3. Komponentensichten

Komponenten sollen sowohl strukturell als auch zeitlich betrachtet werden können. Das heißt, die strukturelle Sicht soll ein Echtzeit-System als Menge von Komponenten darstellen, während die zeitliche Sicht das System als Menge von Tasks darstellt.

3.4. Zeitliche Eigenschaften

Das Komponentenmodell soll Methoden zur Verfügung stellen, mit dem sich die zeitlichen Eigenschaften, wie z.B. worst-case-execution-time der Komponenten untersuchen lassen. Außerdem sollen sowohl statische als auch dynamische Untersuchungen des resultierenden Echtzeitsystems möglich sein.

3.5. Aufgabenzuweisung

Richtlinien, oder besser Werkzeuge sollen zur Verfügung stehen, mit denen sich Komponenten einzelnen Tasks zuweisen lassen.

3.6. Aspekt-Unterstützung

Unterstützung für die Trennung von “crosscutting concerns”, Aspekte des Softwaresystems sollen spezifiziert werden können.

3.7. Aspektweben

Werkzeuge, die Aspekte in die Software weben sollen zur Verfügung gestellt werden.

3.8. Mehrere Schnittstellen

Komponenten sollen mehrere Schnittstellen besitzen, durch die man auf sie zugreifen kann.

3.9. Mehrere Aspekttypen

Neben Aspekten, die Komponentencode verändern, sollen auch Aspekte

unterstützt werden, die das Verhalten von Komponenten in der Laufzeitumgebung beschreiben.

3.10. Unterstützung bei der Konfiguration

Das Zusammenfügen verschiedenster Komponenten zu einer Systemkonfiguration soll vereinfacht werden.

3.11. Zeitliche Analyse

Das aus Komponenten zusammengesetzte Echtzeit-System soll natürlich auch auf seine Echtzeitfähigkeit untersucht werden können.

4. ACCORD

ACCORD ist der Versuch, eine Designmethode für Echtzeitsysteme zu entwickeln, die möglichst viele der Anforderungen aus Kapitel 3 erfüllt.

4.1. Klassifikation von Aspekten in Echtzeitsystemen

ACCORD unterscheidet zwischen drei verschiedenen Typen von Aspekten:

- Anwendungsaspekte: Diese Aspekte verändern das Verhalten von Komponenten, indem sie ihren Code beeinflussen. Unter Anwendung versteht man in diesem Zusammenhang einen bestimmten querschneidenden Belang, z.B. Synchronisation.
- Laufzeitaspekte: Diese Aspekte ändern den Code der Komponente nicht, sondern geben vielmehr Auskunft über Laufzeitanforderungen der Komponente, wie z.B. temporale Aspekte, Speicheranforderungen oder unterstützte Echtzeitbetriebssysteme.
- Kompositionsaspekte: Diese Aspekte ändern den Code der Komponente ebenfalls nicht, sondern beschreiben Voraussetzungen bzw. Bedingungen für das Zusammenfügen von Komponenten. Beispiele wären Kompatibilitätsaspekte oder Versionsaspekte.

Hierzu ist allerdings zu erwähnen, dass Laufzeiaspekte und Kompositionsaspekte keine Aspekte im aspektorientierten Sinn darstellen, sondern wohl eher als Eigenschaften verstanden werden sollen.

4.2. Real-Time Component Model

RTCOM (real-time component model) ein Echtzeit-Komponentenmodell ermöglicht einfaches und vorhersagbares Aspektweben, also Integrieren von Aspekten in Komponenten. Es besteht aus einem funktionalen Teil, einem Laufzeitsystem-abhängigen Teil und einem Kompositionsteil, dessen genaue Funktionsweise allerdings noch zu untersuchen ist.

4.2.1 Funktionaler Teil von RTCOM

Einige Definitionen sind nötig, bevor man den funktionalen Teil definieren kann:

1. Eine **Komponente** c ist ein Tupel <M, O, I>, wobei M eine Menge von Mechanismen gekapselt in c, O eine Menge von Operationen von c und I eine Menge von Komponentenschnittstellen von c sind.
2. Eine Menge von **Mechanismen** M einer Komponente c ist eine nichtleere Menge von Funktionen, die in c gekapselt sind.
3. Eine Menge von **Operationen** O einer Komponente c ist eine Menge von Funktionen, die in c implementiert sind, wobei für jede Operation $o \in O$ eine nichtleere Teilmenge von Mechanismen $K \subseteq M$, sowie eine Teilmenge von Operationen L anderer Komponenten (C\{c}) und eine Abbildung existiert, so dass o = f(K, L).
4. Der **funktionale Teil** einer Komponente c wird repräsentiert durch das Tupel <M, O>, wobei M eine Menge von Mechanismen der Komponente sind und O eine Menge von Operationen ist, die von der Komponente implementiert wird.
5. Ein **Anwendungsaspekt** $a \in A$ ist eine Menge von Tupeln $<a^t, P>$ wobei $t \in \{before, after, around\}$, a^t ein Advice vom Typ t ist, definiert durch die Abbildung $a^t = f(K)$, $K \subseteq M$ und P eine Menge von Pointcuts ist, die eine Teilmenge von Operationen in Komponenten beschreiben.

Nach kurzem Nachdenken erkennt man sofort, dass das Weben von Anwendungsaspekten nicht mehr beliebigen Code einer Komponente ändern kann. Die Mechanismen einer Komponente sind vielmehr feste Bestandteile, deren Implementation nicht verändert wird und somit eine Elementaroperation darstellen (was aber nicht bedeutet, dass sie aus einer einzigen Anweisung bestehen). Lediglich die Implementation der Operationen, die letztendlich immer mit Hilfe von Mechanismen implementiert sind, kann durch Aspekte beeinflusst werden.

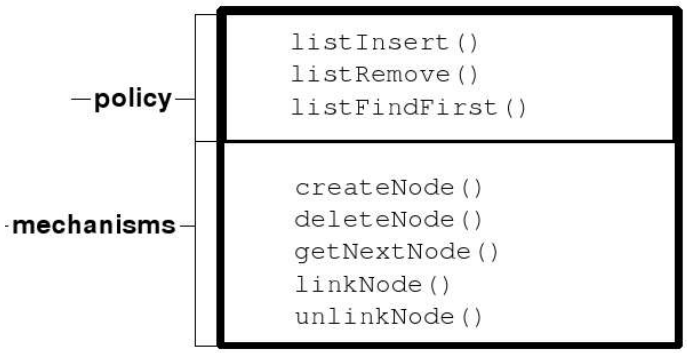

Abb. 2

Dies macht man sich bei der Entwicklung des funktionalen Teils einer

Komponente zunutze. Zuerst werden Mechanismen entworfen, die die Basis für eine Komponente bilden. Mit Hilfe dieser Mechanismen werden nun Operationen implementiert. Diese Operationen und ihre Implementation bilden ein erstes Policy Framework, welches sich durch Aspektweben beeinflussen läßt.

Als einfaches Beispiel sei eine verkettete Liste, implementiert als Komponente genannt. Geeignete Mechanismen zur Manipulation der Liste wären dann z.B. `createNode`, `deleteNode`, `getNextNode`, `linkNode` und `unlinkNode` (Abb. 2). Die Operationen `listInsert`, `listRemove` und `listFindFirst` beschreiben das Verhalten der Liste indem sie die Mechanismen der Komponente verwenden.

Angenommen `listInsert` benutzt `createNode` und `linkNode` um ein neues Listenelement einzufügen und `listRemove` entfernt immer das erste Element aus der Liste, so ist die Policy der Komponente offensichtlich FIFO. Will man nun die Komponente auf eine prioritätenbasierte Liste umstellen, bietet es sich an, `listInsert` als Joinpoint für einen `before`-Advice zu verwenden, der vor dem Einfügen den richtigen Platz in der Liste für das neue Element sucht (vgl. Abb. 3).

```
aspect listPriority{
1:
2: pointcut listInsertCall(List_Operands * op)=
3:    call("void listInsert(List_Operands*)")&&args(op);
4:
5: advice listInsertCall(op):
6:    void before(List_Operands * op){
7:      while
8:         the node position is not determined
9:       do
10:        node = getNextNode(node);
11:        /* determine position of op->node based
12:        on its priority and the priority of the
13:        node in the list*/
14:    }
15: }
```

Abb. 3

4.2.2. Laufzeitsystemabhängiger Teil

Dieser Teil von RTCOM beschäftigt sich mit Laufzeitaspekten von Komponenten, hauptsächlich mit deren zeitlichem Verhalten und zwar sowohl in ihrer Standardform, als auch mit eingewobenen Aspekten. Einer der wichtigsten Laufzeitaspekte ist die worst-case-execution-time (WCET).

Zwei Dinge fallen bei der Betrachtung von Komponenten, die durch Aspektweben modifiziert wurden auf:

- die WCET von Mechanismen ändert sich nicht, da sie unveränderliche Bestandteile des Echtzeit-Systems sind

- Aspektweben verändert Operationen, indem die Anzahl der Mechanismen, die sie aufrufen geändert wird, es ändert sich also das zeitliche Verhalten einer Operation

Dies erleichtert die Berechnung der WCET, auch wenn Aspekte in eine Komponente gewoben werden, Voraussetzungen sind lediglich:

- die WCET jedes Mechanismus ist bekannt und spezifiziert
- die WCET jeder Operation errechnet sich aus den WCETs der Mechanismen, die sie verwendet und der internen WCET des Funktionsrumpfes, der die Operation implementiert, das heißt der Funktionsrumpf, der die Mechanismen aufruft
- die WCET eines Advices, der die Implementation der Operation verändert, basiert auf den WCETs der Mechanismen, die verwendet werden, um den Advice zu implementieren und der internen WCET des Advice-Rumpfs

Für das Beispiel der verketteten Liste bedeutet das folgendes (vgl. Abb. 4): die Mechanismen deleteNode, createNode usw. werden zusammen mit ihrer WCET deklariert. In der Policy-Deklaration werden die jeweiligen Operationen definiert, und zwar indem sowohl die Mechanismen, die sie verwenden angegeben werden, als auch die interne WCET der Operation. Zum Beispiel verwendet die Operation listInsert jeweils einmal die Mechanismen createNode und linkNode und hat darüberhinaus eine interne WCET von 1 ms.

```
policy(noOfElements){
  operation{
    name listInsert;
    uses{
      createNode 1;
      linkNode   1;
    }
    intWcet 1ms;
  }
  operation{
    name listRemove;
    uses{
      getNextNode noOfElements;
      unlinkNode  1;
      deleteNode   1;
    }
    intWcet 4ms;
  }
....
}
```

```
mechanisms{
  mechanism{
    name createNode;
    wcet  5ms;
  }
  mechanism{
    name deleteNode;
    wcet  4ms;
  }
  mechanism{
    name getNextNode;
    wcet  2ms;
  }

....
}
```

Abb. 4

4.3. RTCOM Schnittstellen

Drei verschiedene Schnittstellentypen werden von RTCOM unterstützt, die im

weiteren näher erläutert werden.

4.3.1. Funktionale Schnittstellen

Unter einer funktionalen Schnittstelle I_f einer Komponente c versteht man ein Tupel $<I^p_f, I^r_f>$. I^p_f bezeichnet hierbei eine Schnittstelle, die die Komponente zur Verfügung stellt ("provided") und I^r_f bezeichnet eine Schnittstelle, die von c benötigt ("required") wird, also eine Menge von Operationen anderer Komponenten. Die Unterscheidung zwischen diesen beiden Arten von funktionalen Schnittstellen soll den Austausch von Komponenten erleichtern. Abbildung 5 zeigt beispielhaft die Spezifikation solcher Schnittstellen.

```
provided interface {
    name [nameOfOperation];
    name [nameOfOperation];
    name [nameOfOperation];

....
}
```

```
required interface {
  interface{
    name [nameOfOperation]
    requires{
       [operationRequired]
    }
 ....
  }

....
}
```

Abb. 5

4.3.2. Konfigurationsschnittstellen und Kompositionsschnittstellen

Eine Konfigurationsschnittstelle I_g einer Komponente c ist ein Tupel <O, X>, wobei O eine Menge von Signaturen der Operationen, die von der Komponente implementiert werden repräsentiert und X eine Menge von Parametern ist, die das Verhalten der Operationen beschreiben.

Diese Schnittstelle soll, wenn ihre volle Funktionalität feststeht, der leichteren Integration von Komponenten in Echtzeit-Systeme dienen.

Eine Kompositionsschnittstelle I_c einer Komponente c hingegen ist eine Menge von Signaturen von Mechanismen der Komponente.

Kompositionsschnittstellen werden zum Aspektweben benutzt, wobei die Schnittstellen auch nur dann aktiv werden, wenn tatsächlich Aspekte in das System gewoben werden sollen.

5. Zusammenfassung und Ausblick

ACCORD erfüllt die meisten der Kriterien, die unter 3. von einer Designmethode gefordert wurden. RTCOM ermöglicht die Zerlegung eines Softwaresystems in einzelne Komponenten und Aspekte. ACCORD und RTCOM bilden somit eine Kombination aus aspektorientierter komponentenbasierter Softwareentwicklung unter Berücksichtigung von WCET-Verfahren. Defizite existieren noch beim Zuweisen von Komponenten an Tasks und bei der Konfigurationsunterstützung von Echtzeit-Systemen. Für letzteres stehen zwar Designrichtlinien zur Verfügung, jedoch noch keine Werkzeuge. Eine genauere Spezifikation der Komponentenschnittstellen ist ebenfalls noch nötig.

6. Bibliographie

[1] A. Tešanović, Towards Aspectual Component-Based Real-Time System Development, Licentiate Thesis, Department of Computer Science, Linköping University, 2003

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley / ACM Press, 1998