CiAO-OS

17. April 2007

Dieses Dokument ist das Zentraldokument für Ziele, Analysen, Entscheidungen, Implementierungsanweisungen, offene Punkte, einfache Notizen, Ideen und sonstige Belange bezüglich der Entwicklung des CiAO-Betriebssystems. Es ist gedacht als Dokumentation für die Projektbeteiligten und als Diskussionsgrundlage im Rahmen des Entwicklungsprozesses. Als ein *lebendes Dokument* **darf und soll** es von allen Beteiligten in diesem Sinne bearbeitet werden. Auch „nicht druckreife“ Inhalte, Ideen, etc. sind ausdrücklich erwünscht.

Inhaltsverzeichnis

1 Grundlegende Ziele und Herangehensweise

Kapitelversion
Author: siwahofe
Rev: 387
Revision Date: 2007-03-30 09:34:51 +0200 (Fri, 30 Mar 2007)
Datum: 17. April 2007

Dieses Kapitel beschreibt die Designziele des CiAO-Betriebssystems und die daraus resultierende Herangehensweise. Es diskutiert die wissenschaftlichen Ziele, die im Rahmen des Projekts „CiAO-Betriebssystem“ erreicht werden sollen. Intention dieser Seiten ist, die dafür wesentlichen, weniger wesentlichen und unwesentlichen Merkmale herauszuarbeiten, um so zu einer weitestgehenden Eingrenzung des Entwurfraums und einer Fokussierung auf „das Eigentliche“ zu kommen.

1.1 Ziele - Worum es eigentlich geht

> *Mit dem CiAO-Betriebssystem soll gezeigt werden, dass durch die **gezielte Anwendung von Aspekttechniken** im Betriebssystembau **Konfigurierbarkeit** von **architekturellen Eigenschaften** ohne signifikante Nachteile im Bezug auf **Laufzeit- und Speicherplatzeffizenz** erreichbar ist.*

Dieser eine Satz umreißt die wissenschaftlichen Ziele des CiAO-Betriebssystems. Bezogen auf das Gesamtprojekt sollten sie als mittelfristige Ziele verstanden werden. Längerfristig mögen sich, auch je nach Interesse der Beteiligten, weitere Ziele ergeben. Im Folgenden sollen jedoch erst einmal nur diese mittelfristigen Ziele näher erörtert werden.

1.1.1 Primärziele

Z1) Konfigurierbarkeit architektureller Eigenschaften ist das angestrebte Hauptmerkmal des CiAO-*Betriebssystems*, dem sich letztlich alles unterzuordnen hat; wobei die Konfigurierbarkeit der Architektur insbesondere durch Z2 erreicht werden soll.

Z2) Anwendung (und Evaluation) von Aspekttechniken ist das andere angestrebete Hauptmerkmal. Im *allgemeinen* konnte die erfolgreiche Anwendung von Aspekttechniken im Systemkontext bereits durch die vorangegangenen Arbeiten (EUROSYS, TAOSD, SIGOPS-EW) gezeigt werden. Schwerpunkt soll daher deren *gezielte Anwendung* für Z1 sein.

Z3) Laufzeit- und Speicherplatzeffizienz. Hierbei handelt es sich einfach um das wichtigste nicht-funktionale Akzeptanzkriterium für Z1 und Z2. Effizienz gilt als *die* notwendige Bedingung in der Domäne (eingebettete) Betriebssysteme.

Die Primärziele sind gleichsam die *kritischen Ziele*, die erreicht werden müssen, um den Ansatz wissenschaftlich als erfolgreich zu zeigen. Der Ansatz ist gescheitert, wenn Z1 entweder gar nicht oder im Wesentlichen ohne Z2 erreicht wird (weil sich etwa herausstellt, dass man Architektur viel besser mit Makros konfigurierbar gestalten kann). Es wird zumindest schwierig, wenn sich herausstellt, dass Z1 in erheblichem Maße Z3 widerspricht.

1.1.2 Sekundärziele

Neben den genannten Zielen gibt es noch eine Reihe weiterer Ziele beim Betriebssystembau, die allgemein als wichtig oder wünschenswert betrachtet werden. Derartige Ziele sind zahlreich, ihre Auswahl ist beim Einzelnen stark beeinflusst von der Prägung durch eine bestimmte Schule oder Philosophie (wie z.B. Mikrokerne, UNIX, Kleinstrechner, Anwendungsanpassbarkeit). Auf diesem Hintergrund ist jede Aufzählung weiterer Ziele naturgemäß unvollständig, dieses Dokument konzentriert sich daher insbesondere auf diejenigen, die bei uns oft genannt werden.

Diese Ziele werden im Folgenden *Sekundärziele* genannt, weil sie für die Erreichung der *wissenschaftlichen* Alleinstellungsmerkmale nicht kritisch sind. Das bedeutet nicht unbedingt, dass sie grundsätzlich unwichtig sind, insbesondere wenn man CiAO als „Produkt“ betrachtet. Sie sind jedoch zweitrangig für das mittelfristig angestrebte Gesamtergebnis. Kurz gesagt: Primärziele *müssen gezeigt* werden, Sekundärziele *sollen dabei berücksichtigt* werden.

Z4) Portabilität beschreibt die Wiederverwendbarkeit für unterschiedliche Hardwareplattformen.

Z5) Funktionale Anpassbarkeit (Variabilität) beschreibt die angebotene Funktionsfülle der Betriebssystemfunktionalität bzw. ihrer Ausprägungen *jenseits* der konfigurierbaren Architekturmerkmale aus Z1 und der Portabilität aus Z4.

Z6) Funktionale Auswählbarkeit (Granularität) beschreibt die Skalierbarkeit der angebotenen Funktionalität aus Z5. D.h. in wie weit Funktionalitäten explizite minimale Erweiterungen anderer Funktionalitäten sind.

1.1.3 Superziele

Schließlich gibt es noch Ziele, die nicht das Betriebssystem oder wissenschaftlichen Fragestellungen selbst betreffen, sondern die Zielentwicklung selbst sowie den Entwicklungsprozess und alles was damit verbunden ist.

Z7) Einfachheit ist ein Ziel des gesamten Entwicklungsprozesses. Es basiert auf der Grundannahme, dass jede zusätzliche Anforderung, jede zusätzliche Komponente und jedes zusätzliche Werkzeug zunächst einmal die Komplexität erhöht und damit die Erreichung der Primärziele potentiell gefährdet. Einfachheit als Ziel bedeutet, dass Kosten/Nutzen-Abwägungen im Hinblick auf diese Ziele durchzuführen sind.

Z8) Wissenschaftliche Vermarktbarkeit ist ein weiteres Superziel. Es beschreibt qualitativ, wie einfach oder gut die gewonnenen Erkenntnisse (in Veröffentlichungen) vermittelbar sind. Es beschreibt außerdem quantitativ, aus wie vielen verschiedenen Blickwinkeln („Märkte“) die Erkenntnisse vermittelbar sind. Voraussetzung ist natürlich, dass mit der Erfüllung der wissenschaftlichen Primärziele auch ein echter Erkenntnisgewinn verbunden ist.
Grundannahme ist nun, dass die Erkenntnisvermittlung um so leichter ist, je weniger Hintergrundwissen ein potentieller Leser über das sonstige Umfeld benötigt. Das spricht zunächst mal grundsätzlich für Einfachheit (Z7). Besonders wichtig für die Vermarktbarkeit ist die Problemmotivation: Idealerweise löst man bekannte Probleme in einem bekannten Umfeld. Der Nachweis eines „Benefit“ ist umso leichter, je direkter man sich mit existierenden Arbeiten vergleichen kann.

1.2 Gewichtung

Wenn man mit verschiedenen Beteiligten mit je eigenem Blickwinkel über Ziele spricht, landet man am Ende oft bei der Feststellung, dass sie alle „irgendwie wichtig“ sind. Das ist zwar richtig, aber eben nicht zielführend, weil so keine Konzentration möglich ist. Im Folgenden soll deshalb eine Gewichtung der genannten Ziele erfolgen.

1.2.1 Methode: Zielpunkte

Die Gewichtungsmethode basiert auf folgender (vereinfachender) Grundannahme: Je stärker ein Ziel berücksichtigt werden soll, desto höher ist der dadurch verursachte Aufwand. Verschiedene Ziele stehen jeweils in Konkurrenz zueinander. Für eine Gewichtung bietet es sich deshalb an, eine begrenzte Anzahl von *Zielpunkten* zu verteilen.

Gegeben seien hier 3 Zielpunkte pro Ziel, bei 8 Zielen wären das insgesamt 24 Punkte. Diese können nun beliebig verteilt und die Ziele dadurch relativ zueinander gewichtet werden, wobei Primärziele natürlich höhere Punktezuweisungen als Sekundärziele haben sollen. Für die verschiedenen Zielarten sollen die Punkte folgendermaßen interpretiert werden:

Primärziele: Primärziele sind *kritische* Ziele, sämtliche Primärziele müssen erreicht werden. Zielpunkte beschreiben darum keine absolute Priorisierung, sondern das relative Gewicht untereinander. Es kann als *Maßstab* für den investierten Aufwand sowie für Entscheidungen zugunsten einzelner Primärziele in Konfliktfällen verstanden werden. Falls z.B. Z1 zu Z3 mit 7 zu 3 gewichtet sind, so soll in 7 von 10 Konfliktfällen für Z1, in 3 von 10 Fällen für Z3 entschieden werden.

Sekundärziele: Sekundärziele sind *wünschenswerte* Ziele. Zielpunkte sollen hier als maximal akzeptabler Zusatzaufwand (Zeit für Einarbeitung, Entwurf, Entwicklung, Test, Wartung) verstanden werden, d.h. die Menge an zusätzlicher Zeit, die man bereit ist, in die Realisierung eines Sekundärziels zu investieren. Ein Zielpunkt entspreche 10% zusätzlichem Aufwand.

Superziele: Superziele sind übergeordnete und damit *immaterielle* Ziele. Sie werden verfolgt mit der Absicht der globalen Aufwands- und Risikominimierung und haben damit einen positiven Einfluss auf die Erreichbarkeit der anderen Ziele. Zielpunkte für Superziele stellen quasi einen „freiwilligen Verzicht“ bei anderen Zielen dar.

1.2.2 Zielpunkteverteilung für CiAO

Die folgende Tabelle zeigt die Zielpunkteverteilung, die ich (Daniel) für das CiAO-Betriebssystem gewählt habe:

Z1	Konfigurierbare Architektur	6	
Z2	Anwendung von AOP	4	
Z3	Effizienz	3	
Z4	Portabilität	1	
Z5	Variabilität	1	
Z6	Granularität	2	
Z7	Einfachheit	4	
Z8	Wissenschaftliche Vermarktbarkeit	3	
	Summe	**24**	

1.2.3 Interpretation und Begründung

Folgende grundsätzliche Überlegungen spiegeln sich in der Punkteverteilung wieder:

- Die Punktesummen von Primärzielen sind zu den von Sekundär- und Superzielen jeweils etwa mit 3:1 gewichtet.

- Z1 ist höher als Z2 gewichtet, da zu Z2 ja bereits einiges gezeigt wurde (insbesondere EuroSys, TAOSD). Im Rahmen des CiAO-*Betriebssystems* liegt der Fokus daher auf der Anwendung für Z1, die Realisierung der konfigurierbaren Architektur.
- Z3 (Effizienz) wurde bezüglich Z2 ebenfalls schon gezeigt, der Fokus im Rahmen des CiAO-*Betriebssystems* liegt daher auf den aus Z1 resultierenden Kosten bzw. darauf, das System „als Ganzes“ schlank zu halten.
- Architekturelle Konfigurierbarkeit ist wichtiger als Effizienz. Einfachheit ist wichtiger als Effizienz.
- Einfachheit ist wichtiger als jedes Sekundärziel.
- Vermarktbarkeit ist wichtiger als jedes Sekundärziel.

Die starke Gewichtung der Primärziele ist sicherlich unmittelbar einsehbar. Die verhältnismäßig hohe Gewichtung der Einfachheit auf Kosten der Sekundärziele soll den grundsätzlichen Projekterfolg (im Sinne der wissenschaftlichen Ziele) sicherstellen.

Der höchste Interpretationsbedarf besteht wohl bei den Sekundärzielen. Der grundsätzliche Gedanke ist dabei: Keine Verfolgung von Sekundärzielen *„auf Vorrat*“, d.h. ohne konkreten Bedarfsfall. Sekundärziele sollen damit jedoch nicht völlig ignoriert werden. Sie sollen vielmehr in einem Maße verfolgt werden, dass die notwendige Änderungen im *konkreten* Bedarfsfall begrenzt bleiben. Das wird vor allem durch Einfachheit erreicht.

Im Einzelnen bedeutet das:

Portabilität ist gering gewichtet, weil sie im Bezug auf die wissenschaftlichen Primärziele unerheblich ist.

- Der mit CiAO verfolgte *Ansatz* ist im Erfolgsfall inhärent portabel: Wenn es gelingt, durch Weben in einer Hochsprache Architekturmerkmale wie Kernsynchronisation oder Zeit-/Ereignissteuerung grundsätzlich konfigurierbar zu gestalten, so muss dieses nicht für verschiedene Hardwarearchitekturen bewiesen werden.
- Es geht also, wenn überhaupt, um Portabilität auf der Implementierungsebene. Dort ist Portabilität jedoch nur für die Teile von Belang, die tatsächlich unabhängig von der Hardware sind.
- Dieses sind insbesondere *nicht* die *konkreten Ausprägungen* der Architekturmerkmale. Wie genau bestimmte Varianten der Architektureigenschaften (wie z.B. Speicherschutz) implementiert werden, welche exakte Semantik jeweils geboten werden kann und sinnvoll ist, hängt stark von der gebotenen Hardwareunterstützung ab. Das gebietet schon das Primärziel der Effizienz (Z3).

- Jenseits der konfigurierbaren Architektur soll CiAO hingegen so einfach wie möglich gehalten werden (Z7). Daraus folgt, dass es nicht viele Bestandteile, die, wie Scheduler und Threadverwaltung, eventuell auch Timer und Debug-Konsole, offensichtlich und sinnvoll plattformunabhängig gestaltet werden können.
- Für diese verbleibenden Komponenten sind Inhibitoren einer grundsätzlichen Plattformunabhängigkeit mit den angesetzten 10% Zusatzaufwand realistisch auszuschließen. D.h. im konkreten Bedarfsfall ist die tatsächliche Plattformunabhängigkeit schnell zu erreichen.

Variabilität und Granularität sind *jenseits* von Z1 gering gewichtet, weil ebenfalls unerheblich.

- Dies ist zunächst einmal als Abgrenzung von PURE zu verstehen, wo Variabilität und Granularität in jeglicher Hinsicht Primärziele waren. In CiAO sind Variabilität und Granularität nur in Hinsicht der Architekturmerkmale von Belang. Der Aufwand dafür wird jedoch unter Z1 „verbucht“. Wir brauchen z.B. nicht diverseste Scheduler-Varianten vorzusehen, solange es keinen konkreten Bedarfsfall dafür gibt.
- Das heißt jedoch nicht, dass das System in der Summe kaum variabel sein wird. Schon alleine durch die Interaktionen der Architekturvarianten (z.B. zeit-/ereignisgesteuerter Betrieb) mit den funktionalen Systemkomponenten (z.B. Scheduler) werden sich verschiedene konkrete Variantenanforderungen ergeben.
- Bei CiAO geht es eher darum, zu zeigen, dass man eine *große Spanne* des Entwurfsraums (einer architekturellen Eigenschaft) abdecken. Also etwa zwei bis vier möglichst unterschiedliche Varianten (einer architekturellen Eigenschaft). Bei PURE ging es hingegen eher um eine möglichst feine Unterteilung der Entwurfsspanne.

1.3 Operationalisierung — Grundlegende Entscheidungen

1.3.1 Funktionaler Rahmen

Das zu entwickelnde CiAO-OS muss letztlich die diskutierten Primärziele erfüllen. Nun handelt es sich jedoch bei sämtlichen Primärzielen um Anforderungen mit *nicht-funktionalem* Charakter: Das resultierende System muss *konfigurierbare Architekturmerkmale* bieten (Z1), die Konfigurierbarkeit soll dabei durch *Aspekttechniken* erreicht werden (Z2) und das resultierende Gesamtsystem soll *effizient* sein (Z3). All dieses sagt nichts darüber aus, wie das System *funktional* aussehen soll, d.h. was für Abstraktionen es dem Anwender bieten soll. Letztlich ist das auch unerheblich, es reicht aus, Z1-Z3 anhand *eines* beliebigen *geeigneten* BS (einer beliebigen geeigneten Menge von BS-Abstraktionen) zu zeigen. Nun hilft derartige „Beliebigkeit“ aber zunächst nicht weiter:

1. Die bisherigen Erfahrungen zeigen, dass es zwar möglich ist, BS-Abstraktionen architekturtransparent zu entwerfen und implementieren, dieses jedoch explizit erfolgen muss und einen erheblichen Zusatzaufwand verursacht. Bestehende Implementierungen lassen sich kaum verwenden. Da Z5 (Variabilität) nur ein Sekundärziel von CiAO ist, soll die Menge der geforderten BS-Abstraktionen im Sinne von Z7 (Einfachheit) „beliebig aber *klein*“ sein.

2. Ohne konkrete Anforderungen bezüglich der Funktionalität ist das System hoffnungslos unterspezifiziert - alleine anhand nicht-funktionaler Anforderungen kann man es nun mal nicht sinnvoll entwerfen und bauen. Zum Zwecke der Operationalisierung wird deshalb im Sinne von Z7 (Einfachheit) weiter eingeschränkt: Die Menge der *geforderten* BS-Abstraktionen soll „beliebig aber klein und *fest*“ sein - fest im Sinne von wohlspezifiziert.

Gesucht ist also ein *funktionaler Rahmen*, eine möglichst kleine und wohlspezifizierte Menge von BS-Abstraktionen, die in ihrer Gestalt und Gesamtheit dennoch geeignet sind, genügend „Angriffsfläche“ für konfigurierbare Architektureigenschaften zu bieten: Ein inhärent einfädiges System bietet z.B. keine Verwendung für konfigurierbare Synchronisationsmechanismen. Für ein System, in dem die BS-Objekte grundsätzlich von der Anwendung instantiiert werden, ist die Idee von konfigurierbarem (hardware-basiertem) Speicherschutz widersprüchlich, zumindest aber sehr schwierig durchzusetzen.

Da, abgesehen von der „Angriffsfläche“, der funktionale Rahmen nebensächlich ist, bietet es sich im Sinne von Z7 und Z8 an, möglichst nahe an einer *existierenden* Spezifikation zu arbeiten. Als Vorbild könnte man eCos oder AUTOSAR-OS nehmen. AUTOSAR-OS bietet sich dabei in besonderer Weise an:

- Es handelt sich um einen ausgearbeiteten Anforderungskatalog[SRS] sowie eine fertige Spezifikation[SWS] von relativ gut überschaubarem Umfang. (**Z7**)
  - Das erleichtert insbesondere auch die Einbeziehung von Studenten. (**Z7**)
- AUTOSAR-OS sieht bereits einige eingeschränkt konfigurierbare Architektureigenschaften vor (Speicherschutz[SRS 14] , Zeitschutz[SRS 15] , Privilegebenen[SRS 16]).
  - Zumindest für diese Eigenschaften ist damit schon mal sicher gestellt, dass sie nicht im Widerspruch zur funktionalen Spezifikation stehen. (**Z1**, **Z2**, **Z7**)
  - Die Idee konfigurierbarer Architektureigenschaften ist in dieser Community grundsätzlich schon mal motiviert. (**Z1**, **Z8**)
  - Auch wenn die Spezifikation keinerlei Angaben zur Implementierung macht, merkt man ihr an, dass einige Konzepte (z.B. *Trusted Functions*[SWS 42]) aus implementierungstechnischen Gründen stark eingeschränkt wurden. Hier bietet sich eine reichhaltige Angriffsfläche für den AOP-Ansatz, mit dem derartige Einschränkungen nicht erforderlich wären. (**Z2**)

- AUTOSAR (und damit auch das OS) genießt hohe Aufmerksamkeit. Das erhöht die Attraktivität der eigenen Arbeiten sowohl in der Community als auch bei Studenten. (**Z7**, **Z8**)
- Ein auf Aspekten basierendes „AUTOSAR-OS-ähnliches System“ ist laut DFG-Antrag sowieso wesentliches Ziel des CiAO-Projekts.[1]
- Ein eigenes AUTOSAR-like-OS passt gut in das Profil des Lehrstuhls und ist somit längerfristig als ein strategisches wichtiges Thema zu betrachten. Außerdem wäre es eine gute Basis für weitere Studien.[2]

Feststellung 1: Die Spezifikation von AUTOSAR-OS bildet den *funktionalen Rahmen* für CiAO.

Wichtig ist dabei jedoch Folgendes: Die AUTOSAR-OS Spezifikation soll **zielgerichtet** (nicht „buchstabengetreu“) verwendet werden. Anteile, die weder Voraussetzung noch Beitrag für die Erreichung der Primärziele sind, sollen als optional verstanden werden. Gibt es sogar einen Widerspruch zu Primär- oder Superzielen, so soll gezielt von der Spezifikation abgewichen werden.

Feststellung 2: Die Spezifikation soll und muss nicht buchstabengetreu umgesetzt werden. Es findet eine Abwägung in Hinblick auf Primärziele und Superziele statt.

1.3.2 Konfigurierbare Architektureigenschaften

Nachdem ein funktionaler Rahmen feststeht, soll es nun um die Operationalisierung von Z1 gehen: Für welche *konkreten Architektureigenschaften* soll Konfigurierbarkeit angestrebt werden und welche *konkreten Ausprägungen* dieser Eigenschaften (im Sinne von Variabilität) sind möglich und sinnvoll.

1.3.2.1 AUTOSAR-OS Architektureigenschaften

Anbieten tun sich dabei zunächst die in AUTOSAR-OS sowieso vorgesehenen Architektureigenschaften: Speicher- und Zeitschutz. Wenn man zeigen kann, dass diese sich mit AOP einbringen lassen, ohne den Kern selber modifizieren zu müssen, ist das schon ein gutes Ergebnis. Falls man zudem darauf aufbauend die (teilweise wohl wegen implementierungstechnischer Bedenken) als optional klassifizierten Anteile wie z.B. Schutz des Stapels und der privaten Daten eines TASKs / einer ISR „genauso einfach“ durch Aspekte einweben kann, sollte das sehr überzeugend sein.

[1] Wir können ja ausnahmsweise auch mal das machen, was wir im Antrag versprochen haben :-)

[2] Man könnte z.B. zusammen mit Michael KESO darauf protieren und somit erstmalig eine direkte (=faire) Vergleichbarkeit von hardware- und softwarebasiertem Speicherschutz in verschiedensten Ausprägungen und „Mischformen“ auf *demselben* System erreichen. Ergebnisse ließen sich sicherlich auf hohem Niveau publizieren.

1.3.2.2 Weitergehende Architektureigenschaften

Für alle weitergehenden Architekturmerkmale muss bewusst der Rahmen der AUTOSAR-OS Spezifikation verlassen werden. Ein Beispiel ist die Synchronisierung mit Ressourcen, die immer explizit erfolgt und immer das PCP (Priority Ceiling Protocol) verwendet. Hier ließen sich schöne Erweiterungen modellieren, die alternativ PIP (Priority Inheritance Protocol) statt PCP anbieten, ISRs mit einbeziehen und Synchronisationsdomänen mit impliziter Synchronisation beim Betreten und Verlassen vorsehen.

1.3.3 Angestrebte Veröffentlichungen

Nachdem die Festlegung von AUTOSAR-OS als funktionaler Rahmen zunächst vor allem der Einfachheit (Z7) zuträglich ist, so bieten sich in diesem Zusammenhang auch Chancen für hochkarätige Veröffentlichungen. Die Veröffentlichungsabsicht, also die Operationalisierung von Z8 (wissenschaftliche Vermarktbarkeit), sollte erfahrungsgemäß frühzeitig berücksichtigt werden. Angedacht sind dabei zunächst zwei Papiere:

- Ein eher softwaretechnisch orientiertes Papier über die Methodik bzw. Erfahrungen beim Einsatz von AOP, um die spezifizierten Anforderungen umzusetzen. Dieses könnte man gut z.B. auf der AOSD veröffentlichen und wird daher im Folgenden **AOSD-Papier** genannt.
- Ein eher technikorientiertes Papier über den Einsatz von AOP, um BS-spezifische Eigenschaften konfigurierbar zu gestalten und wie sich dieses auf die quantitativen Eigenschaften (Speicher/CPU) auswirkt. Dieses könnte man gut z.B. auf der EuroSys veröffentlichen und wird deshalb im Folgenden als **EuroSys-Papier** bezeichnet.

Frühzeitiges Berücksichtigen bedeutet in diesem Zusammenhang, schon während der Entwicklung die geeigneten Daten zu erheben bzw. die Prozesse zu dokumentieren. Die Frage ist dabei auch, wie die jeweilige „Community tickt“, d.h. was dort als „interessant“ und „wichtig“ erachtet wird.

AOSD-Papier: Die AOSD-Community interessiert sich nur sehr bedingt für Betriebssysteme, Systemsoftware und Dinge wie Laufzeit- oder Speichereffizienz. Nicht dass all dieses als völlig unwichtig gilt, aber die meisten Beteiligten (und damit auch die Reviewer) sind Sprachdesigner und Softwaretechniker (Wosch-Jargon: „Müslifresser“) mit oftmals Java-lastigem Hintergrund. Mit raffinierten Implementierungsdetails und Byte-Zählerei kann man hier erfahrungsgemäß nicht landen. Es geht eher um das „große Ganze“.
Ein heißes Thema ist jedoch „aspect traceability“, also die Frage, wie sich Aspekte und querschneidende Anforderungen durch den Entwicklungsprozess hindurchziehen – von den Anforderungen über die Spezifikation bis zu Design und Implementierung. Da wir einen „industriellen“ Anwendungskatalog inklusive Spec haben, in dem viele querschneidende Belange auftreten, ohne dass diese explizit gekennzeichnet sind, bieten sich hier

gute Möglichkeiten. Ausgehend von einer „Querschnittsanalyse“ zu Beginn (welche als querscheidend klassifizierten Anforderungen wirken sich vermutlich auf welche Systemdienste und Datentypen aus) könnte man das schön über den Entwurf (Abbildung auf AspectC++ Aspekte, aber auch z.B. „linker script slices“ und andere Artefakte) und die Implementierung verfolgen. Mit ein paar Softwaremetriken und etwas Erfahrung gewürzt, dürfte das bereits genug Material für ein gutes AOSD-Papier ergeben.

EuroSys-Papier: Für die OS-Community dürfte das Ergebnis „konfigurierbare BS-Architektur“ und die jeweiligen Auswirkungen von großem Interesse sein. AUTOSAR-OS wäre hier eher der motivierende Rahmen und nicht das Ergebnis. Viel wichtiger wäre wohl die Analyse und Diskussion der *Auswirkungen* von konfigurierbaren Architekturmerkmalen, sowohl qualitativ als auch quantitativ. Besonders gut wäre natürlich, wenn man zeigen könnte, dass dadurch tatsächlich erheblicher Einfluss auf emergente nicht-funktionale Eigenschaften genommen werden kann.

Für das EuroSys-Papier braucht man, alleine schon, um die Messungen durchführen zu können, mit Sicherheit eine sehr viel ausgereiftere Implementierung als für das AOSD-Papier. Dadurch ergibt sich eine „logische Abfolge“ zwischen den Papieren. Wir sollten mit dem AOSD-Papier anfangen.

2 Analyse

Kapitelversion
Author: sijnstre
Rev: 390
Revision Date: 2007-04-03 19:15:41 +0200 (Tue, 03 Apr 2007)
Datum: 17. April 2007

In diesem Kapitel werden die identifizierten Primärziele nebst den Entscheidungen bzgl. ihrer Operationalisierung weitergehend analysiert.

2.1 Architekturelle Eigenschaften

AUTOSAR bietet grundsätzlich zwei Arten von Schutzmechanismen an: *Timing-Schutz* und eine Reihe von Schutzkonzepten, die nur gleichzeitig auftreten und hier unter dem Begriff *Zugriffsschutz* zusammengefasst werden. Aufgabe dieser Schutzmechanismen ist es, dass bestimmte Applikationen die Funktionalität anderer Systemkomponenten oder Applikationen nicht gefährden können.

Während der Timing-Schutz die Einhaltung von zeitlichen Constraints wie Deadlines durch forcieren bestimmter Randbedingungen wie WCETs und Ankunftshäufigkeiten garantiert, verhindert Zugriffsschutz im wesentlichen die Ausführung unzulässiger Instruktionen bzw. Operation auf den falschen Daten.

Zusätzlich zu den von OSEK bekannten *conformance classes* kann man dazu zwischen vier verschiedenen *scalability classes* wählen, von denen einige diese Features teilweise anbieten *müssen* (siehe Tab. 2.1).

Hierfür gibt es zwei Klassen von *Applikationen*: *Trusted* und *non-trusted* applications (Tab. 2.2). Auf *non-trusted applications* werden die Schutzkonzepte, sofern sie durch die gewählte *scala-*

scalability class	1	2	3	4
Timing-Schutz erfordert		x		x
Zugriffsschutz erfordert			x	x

Tabelle 2.1: scalability classes in AUTOSAR

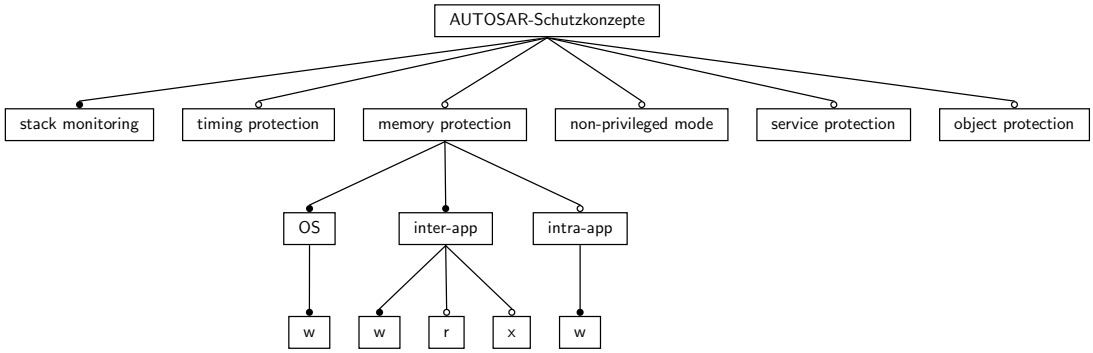

Abbildung 2.1: Schutzkonzepte in AUTOSAR

bility class gefordert sind, derart angewandt, dass diese zwar eventuell sich selbst, jedoch keine anderen Komponenten gefährden können. N*on-trusted applications* unterliegen diesen Beschränkungen nur teilweise.

Applikationen werden Tasks und ISRs zugeordnet, wobei Schutzmechanismen nur auf ISRs der Kategorie 2 angewandt werden.

Schutzmechanismen dienen vorrangig der *Fehlereindämmung*, zum Teil aber auch der *Fehlererkennung*.

2.1.1 Fehlerpropagation

Def.: *A* ist *funktional abhängig* von *B*, wenn die Korrektheit von A bei fehlerhaftem B nicht garantiert werden kann.

Hängt eine Komponente A funktional von einer Komponente B ab, so kann die Ausbreitung von Fehlern in Komponente B nach Komponente A logischerweise nicht ausgeschlossen werden. Das Betriebssystem kann funktionale Fehlerpropagation also nicht direkt verhindern. Die Wahrscheinlichkeit funktionaler Fehlerpropagation kann jedoch eingeschränkt werden, wenn eine Fehlerinkarnation in Komponente B frühzeitig erkannt wird und Maßnahmen zur Wiederherstellung von Komponente B, d.h. deren Überführung in einen fehlerlosen Zustand (beispielsweise durch Neustart) möglich sind.

Eine Applikation, die eine andere Applikation (z.B. einen Treiber) verwendet, funktioniert evtl. nicht mehr korrekt, wenn der Treiber fehlerhaft ist. Umgekehrt kann jedoch garantiert werden, dass sich Fehler der Applikation nicht auf den Treiber auswirken.

Nichtfunktionale Fehlerpropagation kann jedoch ausgeschlossen werden. In diesem Fall ist vollständige Fehlereindämmung bzw. Safety erfüllt.

Def. vollständige Fehlereindämmung: Für alle A,B: A ist nicht funktional abhängig von B: Fehler in B führen nicht zu Fehlern in A.

2.1.2 Timing-Schutz

Timing-Schutz wird *immer* auf die Tasks und ISRs der Kategorie 2 von *non-trusted applications* angewandt [7.6.2.2/OS028]. Für alle anderen Tasks und ISRs der Kategorie 2, soll das OS die Möglichkeit des Timing-Schutzes anbieten [7.6.2.2/OS089].

2.1.3 Zugriffsschutz

Zugriffsschutz sorgt im wesentlichen dafür, dass eine Applikation - absichtlich oder aufgrund eines Fehlers - keine Instruktionen ausführen kann, die die Korrektheit anderer Komponenten gefährden könnte. Damit sind sowohl andere Applikationen als auch das Betriebssystem selbst gemeint.

Hierbei gibt es vier Arten unerlaubter Operationen die jeweils auf unterschiedliche Weise abgefangen werden:

ungültige Systemaufrufe Ein Systemaufruf aus dem falschen Kontext oder mit ungültigen Parametern führt zu einer Ausnahmebehandlung, auch bei *trusted applications*.

fremde Systemobjekte Eine versuchte Manipulation mittels Systemaufruf an einem Objekt (Task, Alarm etc...) einer anderen Applikation führt zu einer Ausnahmebehandlung, falls dies nicht bei der Konfiguration explizit anders gewünscht wurde.

Während der erste Punkt dem Schutz des Betriebssystems dient, schützt der zweite Applikationen vor unbefugter Manipulation ihrer OS-Objekte durch *non-trusted applications*. Obwohl diese Objekte vom Betriebssystem verwaltet werden, kann deren Manipulation mithilfe der vom OS angebotenen Dienste nicht zur Korruption des OS selbst führen. Beide Schutzmechanismen wirken nur an der Betriebssystemschnittstelle und können daher ohne Hardwareunterstützung angeboten werden.

fremde Speicheradressen Eine Anwendung kann die Korrektheit anderer Komponenten gefährden, wenn diese schreibend auf deren Daten zugreift. Manche Prozessoren besitzen eine *MPU (memory protection unit)*, die beim Zugriff auf bestimmte Speicherbereiche eine Ausnahme auslöst. Für *trusted applications* sowie für das Betriebssystem ist der Speicherschutz deaktiviert. Ein Spezialfall des Zugriffs auf fremde Speicheradressen sind Stacküberläufe. Die Entdeckung von Stacküberläufen wird in allen *scalability classes* gefordert [7.4.2/OS087]. Existiert kein hardwareseitiges Speicherschutzkonzept, soll dies per *stack monitoring* bewerkstelligt werden.

spezielle Opcodes Bestimmte Maschinenbefehle haben Einfluss auf alle Komponenten, bzw. umgehen oder deaktivieren die vorhandenen Schutzmechanismen. Prozessoren bieten hierfür oft einen eingeschränkten Modus an, in dem diese Instruktionen zu einer Ausnahme führen. *Non-trusted applications* müssen stets in diesem eingeschränkten (nicht privilegierten) Modus ausgeführt werden, *trusted applications* werden im privilegierten Modus ausgeführt.

Diese beiden Schutzmechanismen sind hardwareabhängig. Während der Speicherschutz sowohl dem Schutz des Betriebssystems, als auch dem Schutz der einzelnen Applikationen dient, dient der nicht privilegierte Modus vor allem dem Schutz OS-verwalteter Register, die ebenso zum Zustand des Betriebssystems gehören.

2.1.3.1 Speicherschutz

Speicherschutz dient in erster Linie der *Fehlereindämmung*: Fehler in non-trusted applications sollen sich nicht auf andere, unbeteiligte Komponenten auswirken. Hierfür ist reiner Schreibschutz auf Applikationsebene ausreichend. Prinzipiell kann die Fehlereindämmung auch engmaschiger, durch Schreibschutz auf Task- bzw ISR-Ebene bewerkstelligt werden. Da es aufgrund funktionaler Abhängigkeiten jedoch wahrscheinlich ist, dass ein Fehler in einer Task die ganze Applikation unbrauchbar macht, ist dieses Feature eher der *Fehlererkennung* zuzurechnen.

Speicherschutz wird jedoch nicht nur zur *Fehlereindämmung* sondern auch zur *Fehlererkennung* eingesetzt. Während man bei der Fehlereindämmung wenigstens ungefähre Vorstellungen von „notwendig“ und „hinreichend“ bezüglich hat, *kann* die Fehlererkennung mittels Speicherschutz gar nicht hinreichend sein.

Durch Lese- und Ausführungsschutz kann die Fehlererkennung verbessert werden. AUTOSAR bietet Lese- und Ausführungsschutz jedoch nur für Applikationen an, was sinnvoll ist, da Anwendungen niemals direkt interagieren. Siehe hier zu auch die Diskussion in Abschnitt 2.3.

2.1.4 Feststellungen

Feststellung 3: *Shared code* wird mit den Rechten und auf dem Zustand der aufrufenden Applikation ausgeführt, während *trusted functions* mit den Rechten und auf dem Zustand der aufgerufenen Applikation ausgeführt werden. Dabei ist letztere Beschreibung eigentlich eine Verallgemeinertes Konzept, das in AUTOSAR jedoch nur als Spezialfall für *trusted applications* angeboten wird. Ein ähnliches Konzept für solche Aufrufe an non-trusted applications (*non-trusted functions*) ist in AUTOSAR nicht vorgesehen.

Feststellung 4: Treiber müssten *trusted* sein, da:

- *non-trusted functions* nicht vorgesehen sind (siehe 1.)
- *non-trusted applications* nicht auf die Peripherie zugreifen dürfen

Feststellung 5: Leseschutz zwischen *non-trusted applications* ist unproblematisch, da aufgrund 1. keine Aufrufe stattfinden, die den Zugriff auf Aufrufparameter notwendig machen, die in anderen Schutzdomänen liegen.

Applikation	non-trusted	trusted
Speicherschutz	x	
Timing-Schutz	x	(x)
privilegierter Modus		x
Peripheriezugriff		x
anbieten v. shared code	x	x (?)
anbieten v. trusted functions		x

Tabelle 2.2: *trusted* und *non-trusted applications*

Feststellung 6: Die Anwendung einiger der Schutzmechanismen soll für verschiedene Betriebssystemobjekte scheinbar einzeln wählbar sein (?):

Feststellung 7: [7.6.1.2/OS027] Das Betriebssystem *kann* Ausführungsschutz für die privaten Codesegmente einer Applikation *anbieten* (daher im MM mit Stern markiert).

- [7.6.2.2/OS089] Das Betriebssystem *muss* Timing-Schutz für Tasks/ISR2s *anbieten*, die nicht Teil einer *non-trusted* application sind.
- Die AUTOSAR-Spezifikation gibt Leseschutz für die Daten einer Applikation als optionales Merkmal an [7.6.1.2/OS026]. Von Leseschutz für die Daten einer Task oder ISR2 ist jedoch keine Rede.

Feststellung 8: Schreibschutz für das OS impliziert Schreibschutz für alle *trusted applications*, da diese im Falle einer Korruption durch andere Applikationen auch das OS korrumpieren können.

2.2 Crosscut-Analyse

	W	R	M	T	O	Wa	
DisableInterruptSource	x		x				
ActivateTask	x			x	x		
TerminateTask	x			x			
ChainTask	x			x	x		
Schedule	x						
GetTaskID		x					
GetTaskState		x			x		
DisableAllInterrupts			x	x			
EnableAllInterrupts			x	x			

	W	R	M	T	O	Wa	
DisableInterruptSource	x		x				
SuspendAllInterrupts	x		x	x			
ResumeAllInterrupts	x		x	x			
SuspendOSInterrupts	x		x	x			
ResumeOSInterrupts	x		x	x			
GetResource	x			x	x		
ReleaseResource	x			x	x		
SetEvent	x				x		
ClearEvent	x				x		
GetEvent		x			x		
WaitEvent	x			?			
GetAlarmBase		x			x		
GetAlarm		x			x		
SetRelAlarm	x				x		
SetAbsAlarm	x				x		
CancelAlarm	x				x		
GetActiveApplicationMode		x					
StartOS							
ShutdownOS							
GetApplicationID		x					
GetISRID		x					
CallTrustedFunction	x						
CheckISRMemoryAccess	*	x*			x		
CheckTaskMemoryAccess	*	x*			x		
CheckObjectAccess		x			*		
CheckObjectOwnership		x			*		
StartScheduleTableRel	x				x		
StartScheduleTableAbs	x				x		
StopScheduleTable	x				x		
NextScheduleTable	x						
SyncScheduleTable	x						
GetScheduleTableStatus		x					
SetScheduleTableAsync	x						
IncrementCounter	x						
TerminateApplication	x						
EnableInterruptSource	x		x				
APPLICATION	x						
SCHEDULETABLE					x		
TASK				x		x	
ALARM					x		
RESOURCE					x		

	W	R	M	T	O	Wa	
DisableInterruptSource	x		x				
COUNTER					x		
MESSAGE					x		
ISR				x		x	

W Schreibschutz für OS-Speicher und Stack (anderweitiger Schreibschutz hat keine Auswirkung)

Wa Schreibschutz zwischen Tasks/ISR2s innerhalb einer Applikation

R Leseschutz für OS-Daten (nicht Teil der AUTOSAR-Spezifikation!) Es wird hier von verwendetem W ausgegangen, da

1. ein reiner Leseschutz der OS-Daten kaum sinnvoll wäre.
2. aus der Spezifikation nicht immer ersichtlich ist, inwiefern bei schreibenden Zugriffen auch lesender Zugriff notwendig ist. (Hängt beispielsweise davon ab, ob Objekt-IDs direkt Zeiger repräsentieren, oder erst umgewandelt werden müssen)

M Verwendung des nicht-privilegierten Modus für non-trusted applications

O Schutz der Betriebssystemobjekte vor Fremdmanipulation

Weitere Erläuterungen:

- Da alle Operationen nur auf dem Zustand des Betriebssystems und nicht direkt auf dem Zustand der Applikationen, sondern nur auf deren OS-Objekten arbeiten, wirkt sich weder Speicherschutz zwischen Applikationen, noch Speicherschutz zwischen Tasks/ISR2s innerhalb einer Applikation auf diese aus.
- Der Schutz vor allgemein ungültigen OS-Aufrufen betrifft jede Operation und ist daher nicht in der Matrix enthalten.
- Gegenseitige Beeinflussung von Schutzmaßnahmen: Timing-Schutz fügt den betreffenden Operationen schreibenden Zugriff auf das OS hinzu (beispielsweise protokollieren des Ausführungszeitpunkts). Insbesondere interessant bei Operationen, die normalerweise keinen schreibenden Zugriff auf OS-Daten haben (DisableAllInterrupts, EnableAllInterrupts).

2.3 Diskussion

Dieser Abschnitt dient unter anderem der Anpassung der AUTOSAR-Spezifikation an CiAO, insbesondere durch Erweiterung des Merkmalmodells.

Zu diskutieren wäre:

- Zusammenfassung OS-Speicher-Schreibschutz & non-privileged mode zu einem Merkmal (z.B. Schutz der OS-Daten). Dagegen spricht z.B. Safety II mit statischer Überprüfung auf privilegierte Befehle jedoch ohne Ausführungsschutz.
- Wie sinnvoll ist Leseschutz für OS-Daten?
  - auch nur lesende Systemcalls erfordern teure Traps bzw MPU-Modifikationen
  - nur minimale Verbesserung der Fehlererkennung bei „verirrten“ Pointern
  - interessant könnte evtl. eine weitere Segmentierung sein, d.h. ein lesegeschütztes Segment, das alle OS-Daten enthält, auf die sowieso nur aus der *trusted*-Umgebung heraus zugegriffen wird <- Gedankenspiel, aber wohl nur mit ungerechtfertigt hohem Aufwand bis garnicht durchzusetzen.
- Wie sinnvoll ist Ausführungsschutz?
  - bemerkbare Verbesserung der Fehlererkennung bei „verirrten“ Pointern
  - zusätzliche Programmierung der CPRs (die DPRs müssen wg. Schreibschutz ja eh gesetzt werden, daher ist dies kein Negativkriterium für Leseschutz)
- Wie sinnvoll ist Ausführungsschutz für OS-Code?
  - (NB: TriCore: Falls Ausführungsschutz für Apps, aber nicht für OS, muss das OS-Code-Segment benachbart zum shared-code-Segment liegen, da wir nur 2 Ranges haben, und eine für das Codesegment der momentan laufenden Applikation gebraucht wird)
  - Nur sinnvoll für Code, der sowieso Trap bzw MPU-Rekonfiguration benötigt. Wenn kein Leseschutz für OS-Daten gefordert wird, bedeutet Ausführungsschutz für einen nur lesender Service nur unnötigen Overhead.
  - denkbar wäre auch hier eine diesbezügliche weitere Segmentierung des OS-Codes
- Fehlerszenarien, anhand derer man zeigen kann, dass die Zuverlässigkeit des Gesamtsystems bei vollständiger Fehlereindämmung (nur Schreibschutz für OS und Applikationen) durch Merkmale, die nur der Fehlererkennung dienen (Lese-. Ausführungsschutz) tatsächlich erhöht werden kann:
  - dazu muss ein frühzeitig erkannter Fehler auch irgendwie „repariert“ werden können (Neustart des Tasks bzw. der Applikation)
  - wenn der Fehler nicht erkannt wird (weil das entsprechende Merkmal nicht gewählt wurde), muss er sich irgendwie funktional ausbreiten können.
- Sollten *non-trusted applications* auch Services anbieten können (quasi *non-trusted functions*)?

  - Würde den Benefit von Lese- und Ausführungsschutz zwischen Applikationen genauso fragwürdig machen, wie Lese- und Ausführungsschutz für das OS
  - Man bräuchte evtl. genauso den Umweg über Traps, um die Schutzdomäne zu wechseln
  - Leseschutz macht die Parameterübergabe wieder zum Problem; auf wessen Stack läuft der Aufruf dann?
  - Welche Services könnte eine *non-trusted application* wohl anbieten, wenn sie nicht auf Peripherie zugreifen darf (vllt eine gemeinsam genutzte Datenstruktur, die somit nicht korrumpiert werden kann?)
  - Können *non-trusted applications* Dienste irgendwie anderweitig anbieten (z.B. über Messages)?
  - TriCore: *semi-trusted* Treiber?

- Wäre ein *semi-trusted*-Modus sinnvoll?

  - Treiber, die eigentlich *non-trusted* sind, aber trotzdem auf die Peripherie zugreifen können (TriCore: User-1-Modus)
  - bieten dann natürlich *non-trusted functions* an]

- *shared memory* zwischen Applikationen?

  - Noch mehr Schwierigkeiten
  - evtl. gehen uns die Ranges der MPU aus
  - -> Zusammenfassung zu einer Applikation, Messages verwenden