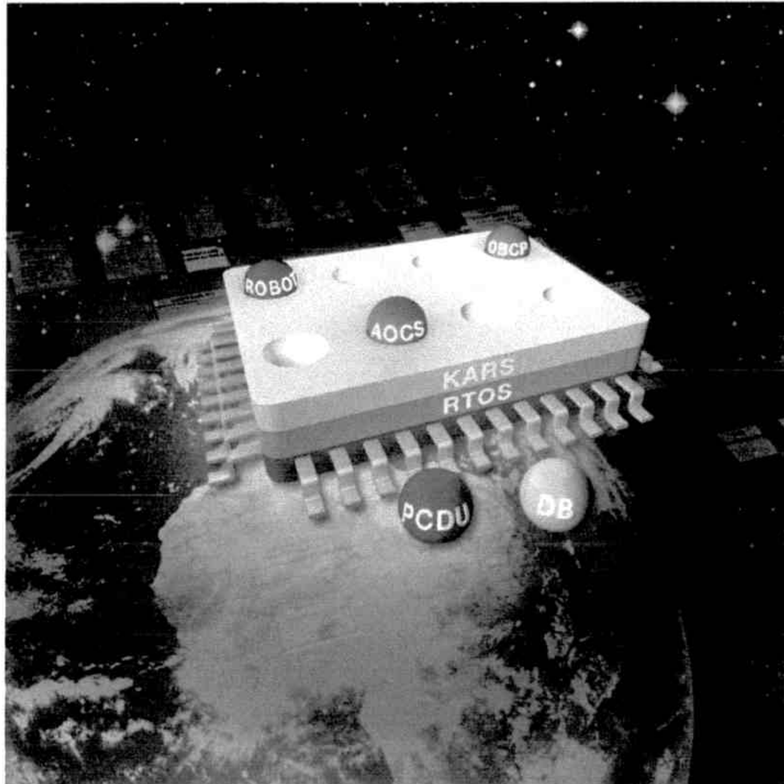
FKZ: 50 RA 1110

KARS: Entwicklung der Systemsteuerungssoftware: Kontroller fuer autonome Raumfahrtsysteme



Abschlussbericht

Doc.-No. **KAR_SYS_ASI_RP_21034**

Version: 1.0

Date: 29.08.2014

	Name	**Date**	**Signature**
Author	S. Jaekel, HJ. Herpel	29.08.2014	
Checked by		29.08.2014	
Quality assurance		29.08.2014	
Released by	Herpel	29.08.2014	

ChangeLog

Ver.	Date	Section/Pg.	Description	Ref.

Contents

1 Einfuehrung 1

1.1 Zweck des Dokumentes . . . 1

1.2 Struktur des Dokumentes . . . 1

2 Abkuerzungen und Definitionen 3

3 Aufgabenstellung und Ziele 5

3.1 Motivation und Aufgabenstellung . . . 5

3.2 Ziele . . . 6

3.3 Planung und Ablauf des Vorhabens . . . 7

3.4 Stand der Technik . . . 10

3.5 Zusammenarbeit mit anderen Stellen . . . 11

4 Erzielte Ergebnisse 13

4.1 Allgemeine Aspekte . . . 13

4.1.1 Einstufung der operationellen Komplexität des Raumfahrzeugs . . . 14

4.1.2 Funktionale Architektur eines autonomen Raumfahrzeugs . . . 15

4.1.3 Die funktionalen Charakteristik der Raumfahrzeugarchitektur . . . 17

4.1.4 Operationelle Aspekte . . . 20

4.1.5 Anwendung von Software-Standards und Compliance zu European Cooperation For Space Standardization (ECSS) . . . 22

4.1.6 Software-Architekturprinzipien . . . 23

4.1.7 Implementierungs Aspekte . . . 25

4.1.8 Aspekte der Zielhardware . . . 26

4.1.9 Entwicklungsumgebung . . . 27

4.2 Softwarearchitektur . . . 30

4.2.1 Überblick . . . 30

4.2.2 Middleware - Component Support Services . . . 30

4.2.3 Basis- und (missions)spezifische Module - System Support Services . . . 31

4.3 Software Konfiguration . . . 36

4.4 Demonstrator-Architektur . . . 39

4.4.1 Demonstrator-Module . . . 40

4.4.2 Test-/Demonstrationsumgebung . . . 43

4.5 Software Evaluation Kit . . . 45

5 Verwertbarkeit der Ergebnisse und Anschlussfaehigkeit **53**
5.1 Verwertbarkeit 53
5.2 Anschlussfähigkeit 53

6 Fortschritte bei anderen Stellen **55**

7 Erstellte Dokumente und Veroeffentlichungen **57**
7.1 Erstellte Dokumente 57
7.2 Veroeffentlichungen 59

References **61**

1 Einfuehrung

1.1 Zweck des Dokumentes

Der Abschlussbericht der Airbus DS beschreibt die Aufgabenstellung, die Planung, die Ergebnisse und den voraussichtlichen Nutzen des Projektes Entwicklung der Systemsteuerungssoftware: Kontroller fuer autonome Raumfahrtsysteme.

1.2 Struktur des Dokumentes

Das Dokument ist wie folgt aufgebaut:

Kap. 3 beleuchtet kurz die Aufgabenstellung und die Ziele, die im Projekt erreicht wurden.
Kap. 4 umfasst die eingehende Darstellung der erzielten Ergebnisses und
Kap. 5 beschreibt den voraussichtlichen Nutzen

2 Abkuerzungen und Definitionen

AOCS Attitude and Orbit Control

API Application Programming Interface

AUTOSAR AUTomotive Open System ARchitecture

CCSDS Consultative Committee for Space Data Systems

CDR Critical Design Review

ECSS European Cooperation For Space Standardization

EDAC Error Detection and Correction

FDIR Failure Detection, Isolation and Recovery

IMA Integrated Modular Avionics

KARS Kontroller fuer autonome Raumfahrtsysteme

OBCP On-board Control Procedure

OMAC Open Modular Architecture for Controls

PCDU Power Control And Distribution Unit

PDR Preliminary Design Review

PUS Packet Utilization Standard

QR Qualification Review

SOI Silicon On Insulator

SRR System Requirements Review

SSS Software System Specification

TRL Technology Readiness Level

UML Unified Modelling Language

DM Data Management

SIF Service Interface

IOH IO Handler

EVH Event Handler

LOH Logging Handler

OBCP On-Board Control Procedure

OBCPH On-Board Control Procedure Handler

MTH Mission Timeline Handler

SUV Supervisor

MPL Mission Planner

OSAL Operating System Abstraction Layer

SRR Software Requirements Review

SDP Software Development Process

SIF Service Interface

TCS Thermal Control System

OBC-SA On-Board Computer System Architecture

DLR Deutsches Zentrum fuer Luft- und Raumfahrt e.V.

SOA Service Oriented Architecture

CORBA Common Object Request Broker Architecture

TIPC Transparent Inter Process Communication

ZeroMQ Zero Message Queue

DDS Data Distribution Service

openDDS open Data Distribution Service

SAVOIR Space AVionics Open aRchitecture

NITRD Networking and Information Technology Research and Development

SSS System Support Services

CSS Component Support Services

FCP Flight Control Procedure

SDP System Datapool

MTL Mission Timeline

TOP Task Orienten Programming

3 Aufgabenstellung und Ziele

3.1 Motivation und Aufgabenstellung

Das vom Deutsches Zentrum fuer Luft- und Raumfahrt e.V. (DLR) initierte Vorhaben leitet sich aus dem Technologie-Programm des DLR ab, das folgende Leitelemente enthält (s. Abb. 3.1):

- Orbiter zur Kommunikation, Erdbeobachtung, In-orbit-Servicing
- Explorations-Missionen (Landen auf Planeten, Monden, Asteroiden)
- Rover (Mobilität auf unbekanntem Terrain) und Robotiksysteme
- Autonome in-situ Labore

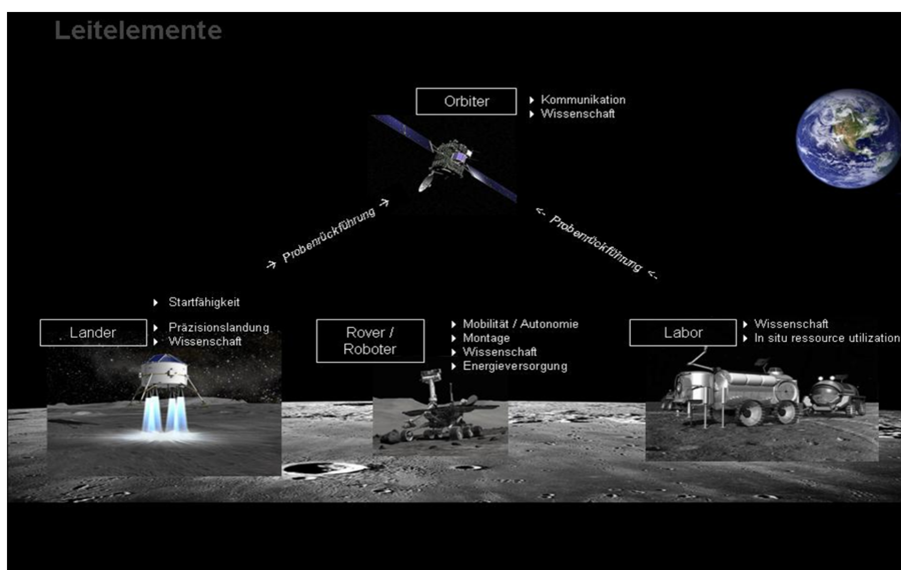


Figure 3.1: DLR Leitelemente der Technologie-Strategie (Quelle: DLR)

Die oben genannten Missionen werden Nutzlasten mit sehr hohen Datenraten aufweisen, die bei bildgebenden Sensoren wie Kameras, Radar-Systemen und Spektrometern bedingt durch hohe zeitliche, geometrische, radiometrische und/ oder spektrale Auflösungen Roh-Datenraten von mehreren Gb/s pro Instrument aufweisen können. Eine derartige Informationsflut lässt sich nur auf akzeptable telemetrierbare Raten reduzieren, wenn gegenüber bisher eingesetzten Kompressions-Verfahren schon eine intelligente, thematische Verarbeitung der Daten an Bord erfolgen kann. Dieses gilt sowohl für die Fernerkundung im erdnahen Orbit als auch in verschärftem Maße für Deep-Space-Missionen. Werden bildgebende Sensoren wie beim On-Orbit-Servicing oder interplanetaren Robotik-Missionen im geschlossenen Regelkreis eingesetzt, sind gegenüber reinen Nutzlast-Anwendungen

zudem noch deutlich erhöhte Anforderungen an die Echtzeitfähigkeit, an die Systemzuverlässigkeit und die Autonomie des Onboard-Rechnersystems und deren Software zu stellen. Weiterhin ist aus Kostengründen ein flexibler, weitestgehender autonomer operationeller Einsatz anzustreben. Dabei ist sicher zu stellen, dass die Onboard-S/W jederzeit z.B. nach Analyse von Rohdaten auf einfache Weise vom Nutzer angepasst und an Bord neu konfiguriert werden kann. Die Beherrschung der wachsenden Komplexität an Bord von Raumfahrzeugen erfordert den Einsatz neuer Technologien sowohl bei den On-board-Rechnern (s. Vorhaben On-Board Computer System Architecture (OBC-SA)) als auch bei der Software, die auf diesen Rechnern zum Einsatz kommt. Im Rahmen dieses Auftrages sollten zukunftsweisende Konzepte für die On-board-Software entwickelt und anhand einer Robotik- Anwendung demonstriert werden. Entsprechend dieser Aufgabenstellung setzte sich das Konsortium aus der Firma Airbus Defence and Space (früher Airbus DS GmbH) und dem DLR Institut für Robotik und Mechatronik zusammen. Airbus DS als typischer Hauptauftragnehmen für komplette Satellitensysteme hat umfassende Erfahrung in der Erstellung hoch-qualitativer Software nach entsprechenden ESA-Standards und das Institut für Robotik und Mechatronik ist führend im Bereich Weltraum-Robotik.

3.2 Ziele

Im Projekt Kontroller fuer autonome Raumfahrtsysteme (KARS) sollte Software entwickelt werden, die geeignet ist, die Anforderungen zukünftiger Missionen nach Autonomie, Flexibilität, Modularität, Performance, Wartbarkeit und Ausfallsicherheit zu gewährleisten. Im Einzelnen sollten folgende Aufgaben bearbeitet werden:

- Definition einer offenen, modularen Softwarearchitektur für zukünftige Missionen mit einem hohen Grad an Autonomie (insbesondere Robotik Missionen, Servicing Missionen und interplanetare Missionen stellen hier hohe Anforderungen) Die Architektur sollte gekennzeichnet sein durch:
  - Verteiltes System, d.h. Anwendungen sollen auf eine Ansammlung von gleichen oder unterschiedlichen Hardware-Modulen verteilt werden können
  - Dienstorientiert
  - Interprozess-Kommunikation über Rechnergrenzen hinweg
  - Verwendung des Packet Utilization Standard (PUS) für die interne und externe Kommunikation und Kommandostruktur
  - Plattform-unabhängigkeit bzw. portabel (Zielarchitektur: x86, Freescale, Leon, ARM)
  - Echtzeit-fähig
  - Zuverlässig, sicher und schnell rekonfigurierbar
- Implementierung von Basis-Komponenten der oben genannten Architektur unter Berücksichtigung des Technology Readiness Level (TRL) 5 (-> tailored ECSS40):
  - Implementierung/Adaptation der Basisdienste (event, logging, time, parameter monitoring, real-time communication, PUS, Failure Detection, Isolation and Recovery (FDIR), Database, ...)

  - Implementierung eines Rahmenprogramms zur standardisierten Einbindung von Komponenteen und Sicherstellung des Datenaustausches zwischen angeschlossenen Modulen
  - Implementierung eines Supervisors
  - Spezifikation der Schnittstellen für eine Missionsplanung
  - Implementierung von Komponenteen zur Demonstration der Funktionalität in einer relevanten Umgebung
  - Implementierung/Adaptation des Robotiks-Komponentes
  - Implementierung/Adaptation eines DataHandling-Komponentes inkl. FDIR-Komponente

- Lieferung eines Demonstrators mit einer repräsentativen Fallstudie implementiert auf einer heterogenen Ausführungsplatform, z.B.
  - Linux als Entwicklungsplatform
  - VxWorks für das Robotik-Komponente
  - PikeOS (oder andere) für andere Komponentee

3.3 Planung und Ablauf des Vorhabens

Das gesamte Vorhaben wurde in fuenf Phasen durchgefÃ¼hrt werden. Jede Phase wurde durch ein Review abgeschlossen. Die Phasen und die dazugehoerigen Reviews (s. Tbl. 3.1) entsprechen den Richtlinien des ECSS-E40.
Die waehrend der Projektdurchfuehrung erstellten Dokumente sind in Tbl. 7.1 aufgelistet.

Machbarkeitsphase (PHASE 0/A)

Ziel der Phase 0/A war es, eine Aussage über die grundsätzlich Machbarkeit der Software. Dazu wurden folgende Aktivitäten durchgeführt:

- Analyse von vorangegangenen Entwicklungen unter Zuhilfenahme der Dokumentation
- Identifikation von Anforderungen und Randbedingungen für die SW Entwicklung [ASI11b]
- Abstimmung und Analyse des Problem- und Anwendungsbereichs der Steuerungssoftware sowie der Zielvorgaben des AG
- Entwicklung eines Referenzszenarios [ASI13a],[ASI13g]
- Definition und Aufsetzen einer gemeinsamen Entwicklungsumgebung
- Erstellung eines Berichts zum Stand der Technik [ASI11c] und den Randbedingungen
- Erstellung eines Berichts KARS Anwendungsbereichs-Analyse und Machbarkeit [ASI13c]

Anforderungsphase (PHASE B1)

Die Phase B wurde in zwei Phasen aufgeteilt. In der Phase B1 werden die Benutzer- sowie die Softwareanforderungen erfasst. Diese Phase schloss mit dem Software Requirements Review (SRR) ab.
Im Einzelnen wurden folgende Aufgaben bearbeitet:

- Identifikation und Festlegung der funktionalen und nicht-funktionalen Benutzeranforderungen
- Bewertung und Priorisierung der funktionalen und nicht funktionalen Benutzeranforderung
- Definition der Verifikationsmethode für jede Benutzeranforderung
- Erstellung eines Dokumentes Software System Spezifikation [ASI11b] und eines dazugehörigen Testplans [ASI12g]
- Umsetzung der Benutzeranforderungen in eine technische Spezifikation (Software Requirements Specification, [ASI12c])
- Definition der Verifikationsmethode für jede Softwareanforderung und Erstellung eines dazugehörigen Testplans [ASI12f]
- Definition der Schnittstellenanforderungen zur Einbindung von Komponentenn

Definitionsphase (PHASE B2)

In der Phase B2 wurde die Softwarearchitektur basierend auf den Softwareanforderungen festgelegt. Diese Phase schloss mit dem Preliminary Design Review (PDR) ab. Die Beschreibung der Software-Architektur erfolgte in UML. Im Einzelnen wurden folgende Aufgaben bearbeitet:

- Abbildung der Anforderungen auf eine System-/Softwarearchitektur [ASI11a]
- Identifikation der Systemkomponenten von KARS und der Demonstratorkomponenten
- Identifikation der operationellen Abhängigketen zwischen den Komponenten [ASI12h]
- Definition der Schnittstellen zwischen den Komponenten [ASI11a]
- Erstellen einer Verhaltensbeschreibung für die einzelnen Komponenten [ASI12b]..[ASI14]

Implementierungsphase (PHASE C)

Während der Phase C wurde die Softwarearchitektur weiter verfeinert und in ausführbare Komponenten umgesetzt. Die Verfeinerung erfolgte in UML. Die einzelnen Komponenten werden in C codiert. Die Phase schloss mit dem Critical Design Review (CDR) ab. Im Einzelnen wurden folgende Aufgaben bearbeitet:

- Verfeinerung des Softwareentwurfs
- Erstellung von Unit-Testprozeduren [ASI12d]

- Erstellung von Testskripten
- Codierung der Module
- Test der Module entsprechende der Unit-Testprozeduren
- Dokumentation der Testergebnisse [ASI13e]
- Erstellung der Testberichte
- Integration der Komponenten mit dem Framework
- Definition und Durchführung von Integrationstests [ASI12d]

Uebergabephase (PHASE D)

Während der Phase D wurde die Demonstrationsumgebung realisiert und das Referenzszenario auf der Demonstrationsumgebung implementiert. Die Abnahme der Kontrollersoftware erfolgte mit Hilfe der Demonstratorumgebung. Die Phase schloss mit dem Software Qualification Review (QR) ab. Im Einzelnen wurden folgende Aufgaben bearbeitet:

- Aufsetzen der Demonstratorhardware
- Integration der Demonstratorhardware mit der Software
- Integration der Demonstratorhardware inkl. Software mit der Testumgebung
- Durchführung von System- und Use-Case-Tests
- Umsetzung der Referenz-Szenarien in Testsequenzen
- Ausführung der Testsequenzen
- Dokumentation der Testergebnisse [ASI12e]
- Beseitigung der aufgetretenen Fehler
- Nachweis, dass die erstellte Software konform mit den Software- und Benutzeranforderungen ist [ASI12i].

Table 3.1: Projektmeilensteine

ID	**Beschreibung**	**Durchgefuehrt**
099900	KO (Kick-Off)	04.Apr.11
399900	FSR (FSR)	02.May.11
499900	SRR (Abschluss Phase B1)	28.Sep.11
599900	PDR (Abschluss Phase B2)	03.May.12
699900	CDR (Abschluss Phase C)	27.Mar.13
799900	QR (Abschluss Phase D)	10.Dec.13

3.4 Stand der Technik

Der enorme Kosten- und Termindruck sowie der Zwang zur Verbesserung der Qualität haben in anderen industriellen Domänen zu Initiativen geführt, deren Ziele ganz ähnlich zu den Zielen sind, die in der Leistungsbeschreibung zu KARS genannt werden. Daher werden diese Entwicklungen in den Domänen Automotive, Luftfahrt und Industriesteuerungen hier aufgelistet. Details zu den einzelnen Lösungen sind in dem Bericht *Stand der Technik* [ASI11c] zu finden.

- Industrieautomatisierung: Open Modular Architecture for Controls (OMAC)
- Automobilindustrie: AUTomotive Open System ARchitecture (AUTOSAR)
- Luftfahrtindustrie: Integrated Modular Avionics (IMA)
- Konsumerbereich: Service Oriented Architecture (SOA)
- Raumfahrt: Space AVionics Open aRchitecture (SAVOIR)

Die oben aufgelisteten Ansätze verwenden i.d.R. eine domänenspezifische *Middleware*.
Die *Middleware* ist eine Zwischenschicht zwischen der eigentlichen Anwendung und der Infrastruktur zum Austausch von Informationen zwischen Anwendungen. Ihre Aufgabe es ist, die Zugriffsmechanismen auf unterhalb angeordnete Schichten zu vereinfachen und die Details deren Infrastruktur nach außen hin zu verbergen. Dazu stellt die Middleware Funktionen zur Verteilung sowie Dienste zur Unterstützung der Anwendung bereit. Die Middleware regelt dabei den Informationsaustausch innerhalb eines Rechners als auch über Rechnergrenzen hinweg. Auf diese Weise wird die Anwendungs-Software entlastet und durch die standardisierte Schnittstelle zu anderen Anwendungen ergibt sich eine höhere Produktivität im Entwicklungsprozess.
Es gibt zahlreiche allgemeine Middleware-Ansätze. Die am häufigsten verwendeten Produkte sind hier aufgelistet:

- Common Object Request Broker Architecture (CORBA)
- Transparent Inter Process Communication (TIPC)
- Zero Message Queue (ZeroMQ)
- Data Distribution Service (DDS)
- open Data Distribution Service (openDDS)

Die hier genannten Produkte sind z.T. sehr aufwendig (z.B. CORBA, DDS) und eignen sich daher nicht für eine Nutzung auf den vergleichsweise leistungsschwachen Raumfahrtrechnern. Zum anderen müssen die Quellen verfügbar sein, um die Software für Anwendungen in der Raumfahrt nutzen zu können. In der Raumfahrt erfolgt die Kommunikation zwischen der Bodenstation und dem Satelliten auf der Grundlage von PUS Diensten. Dieser Standard wird auch für die Kommunikation zwischen Bordrechnern und intelligenter Sensorik/Aktorik verwendet. Aus den oben genannten Gründen lag es daher nahe, eine raumfahrtspezifische Middleware auf der Grundlage der PUS Dienste zu implementieren.

3.5 Zusammenarbeit mit anderen Stellen

Entsprechend der Aufgabenstellung und der jeweils relevanten Erfahrung hatte Airbus DS die Gesamtkoordination des Vorhabens uebernommen. Darueber hinaus war Airbus DS zustaendig fuer die Definition der Software-Architektur und Einhaltung der Qualitaetsanforderungen an die zu erstellende Software entsprechend des ECSS. Das DLR-RM war als Unterauftragnehmer eingebunden, um die umfassende Erfahrung bei der Realisierung von robotischen Systemen einzubringen. Sowohl Airbus DS als auch DLR-RM waren in die Implementierung und den Test der Software eingebunden. Die Fa. KONZEPT GmbH war von Airbus DS beauftragt kritische Komponenten der Software als Prototypen zu realisieren. Fraunhofer FOKUS hat im Rahmen eines weiteren Unterauftrags die Softwarequalitaet mit Hilfe der statischen Codeanalyse beurteilt. Die Empfehlungen wurden von Airbus DS und DLR-RM umgesetzt.
Die Fa. 2Solutions hat Airbus DS beim Review von Dokumenten und des Quelltextes unterstuetzt. In Tabelle 3.2 sind alle beteiligten Institutionen aufgelistet.

Table 3.2: Konsortium

Ref.	**Name**	**Role**
ASI	Airbus DS GmbH Claude-Dornier-Str. 1 88090 Friedrichshafen Phone: (+49) 7545 8 0 http://www.airbusdefenceandspace.com	Prime
2S	2Solution GmbH Schonenfahrer Str. 7 18057 Rostock Phone: +49 (381) 36 76 83 - http://www.2solution.de	SubCo
DLR-RM	Deutsches Zentrum fuer Luft- und Raumfahrt e.V. Postfach 1116 82230 Wessling Phone: +49 8153 28-2400 http://www.robotic.dlr.de/	SubCo
FhG	Fraunhofer-Institut fuer Offene Kommunikationssysteme Kaiserin-Augusta-Allee 31 10589 Berlin Phone: +49 (0) 30 6392 1841 http://www.fokus-fraunhofer.de	SubCo

continued next page ...

Ref.	**Name**	**Role**
KONZEPT	Konzept Informationssysteme GmbH Am Weiher 13 88709 Meersburg Phone: +49 7532 4466 -0 http://www.konzept-is.de	SubCo

4 Erzielte Ergebnisse

4.1 Allgemeine Aspekte

Bis auf einige Ausnahmen sind selbst heutige Raumfahrtsysteme hochgradig manuell kommandiert bzw. werden durch deterministische, linear abgearbeitete Automatismen gesteuert. Je komplexer sich eine Raumfahrtmission allerdings gestaltet, desto weniger ist es dem Menschen möglich, durch die Eingabe von manuellen Kommandos Herr der Lage zu bleiben. Dies gilt insbesondere für ungewisse Umgebungen am Einsatzort, sowie mechanische (Kollision) und zeitliche (Kommunikationsfenster, Kommunikationsverzögerung) Randbedingen bzw. Einschränkungen. Robotische Komponenten lassen die Komplexität eines Raumfahrtsystems durch ihre Interaktion mit der Plattform sowie der Umgebung stark ansteigen und bedürfen daher eines höheren Grads an Autonomie als klassische Raumfahrtsysteme. Dies gilt sowohl für die Steuerung als auch für die Vermeidung von Fehlern und damit der Absicherung, jederzeit in einen sicheren Systemzustand übergehen zu können.
Für die nominale Durchführung der Mission können die folgenden Autonomie-Ebenen (s. Tbl. 4.1) identifiziert werden:

Table 4.1: Autonomie-Ebenen

Level	Beschreibung	Funktionen
E1	Mission execution under ground control; limited on board capability for safety	Real time control from ground for nominal operations Execution of time-tagged commands for safety issues
E2	Execution of pre planned, ground defined, mission operations on board	Capability to store time-based commands in an on board scheduler
E3	Execution of adaptive mission operations on board	Event based autonomous operations Execution of on-board operations control procedures
E4	Execution of goal orient-ed mission operations on board	Goal-oriented mission re-planning

Abb. 4.1 zeigt eine Kategorisierung der möglichen Autonomiegrade für Raumfahrtsysteme. Ein Inkrement in Richtung eines autonomen Systems, ausgehend von manueller Steuerung, ist die assistierte Kontrolle (Assisted Control). Hier wird die manuelle Steuerung eines menschlichen Operators durch autonome Teilsysteme unterstützt. Einen Schritt weiter geht die geteilte Autonomie (Shared Autonomy), bei der sich Operator und System Aufgaben teilen und in der Regel Eingaben auf hohem Level getätigt werden. Komplett eigenständige Systeme, die allerdings weiterhin von einem Operator überwacht werden können (Supervised Autonomy), arbeiten Benutzereingaben auf sehr hohen Level ab. Komplett autonome Systeme müssen ohne jegliche Kommunikation zwischen Operator und Raumfahrtsystem auskommen; ausschließlich anfangs festgelegte Missionsziele sowie

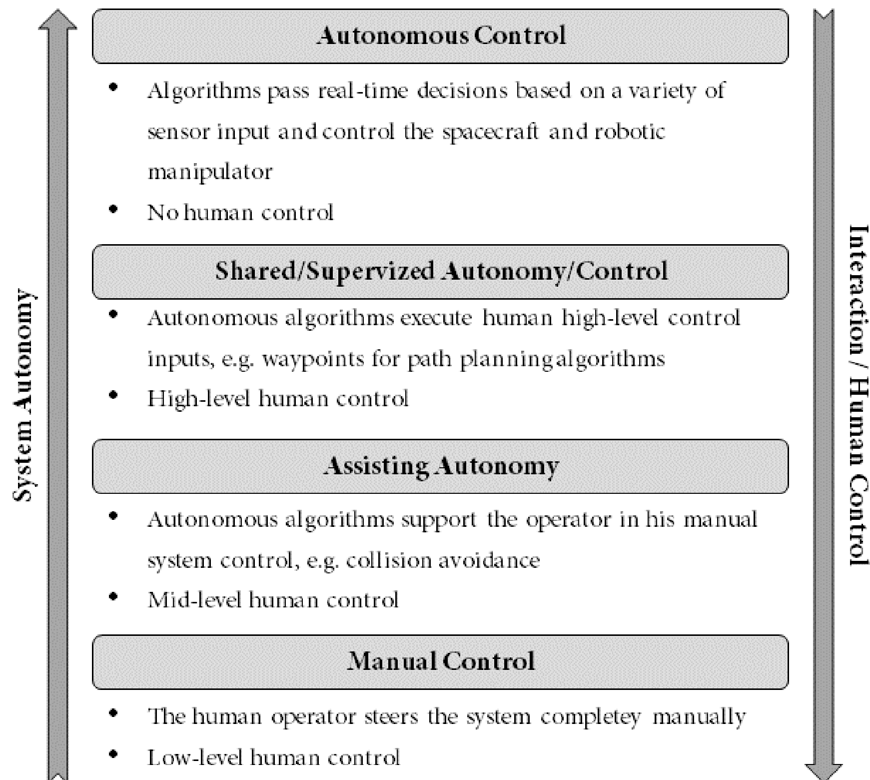


Figure 4.1: Autonomie-Ebenen

die Wahrnehmung und Interkation mit der Umgebung dienen hier als Grundlage für einen sicheren und effektiven, zielgerichteten Betrieb
KARS wurde mit dem Ziel entwickelt, die Anforderungen nach erhöhter Autonomie zukünftiger Satellitenmissionen zu unterstützen.

4.1.1 Einstufung der operationellen Komplexität des Raumfahrzeugs

Die funktionale Architektur bezüglich der an Bord des Raumfahrzeugs und innerhalb des Bodensegmentes zu implementierenden Applikationen und Dienste orientiert sich sehr stark an den im ECSS-E-70-11C S/C Operability Standard definierten Kriterien zur Einstufung der Autonomieanforderungen einer Mission. Gemäß der im Kapitel 5.7 des Standards definierten Autonomiekriterien rangiert die angestrebte Autonomie des Raumfahrzeug in der aktuellen Studie sowohl für die nominalen Betriebsfälle wie auch für die Aspekte der Fehleridentifikation und -behandlung auf den höchsten vorgesehenen Einstufungen.

A: Autonomie in Bezug auf Missionsdurchführung

Level	Beschreibung	Funktionen
E4	Durchführung eine zielorientierten Missionsbetriebs an Bord des Raumfahrzeugs	Zielorientierte Missionsneuplanung

Level	Beschreibung	Funktionen
D2	Die Verwaltung aller Missionsdaten erfolgt an Bord des Satelliten unabhängig von der Verfügbarkeit von Bodenstationen innerhalb eines definierten Autonomieintervals	Abspeicherung aller event, monitoring und control wie auch Missionsdaten an Bord des Raumfahrzeugs.

B: Autonomie in Bezug auf Missionsdatenverwaltung

C: Autonomie in bezug auf Missionsdurchführung

Level	Beschreibung	Funktionen
F2	Autonome Wiederherstellung des vollen operationellen Zustandes nach einem Fehler	Autonome Identifikation an Bord mit Berichterstattung an höhere Überwachungsebenen und an den Boden Autonome Isolation und Rekonfiguration fehlerbehafteter Geräte und Funktionen bis hin zu einer Wiederaufnahme des nominellen operationellen Betriebs Wiederaufnahme der Missionsdatenerzeugung.

Aus diesen Randbedingungen wurden die Anforderungen bzgl. der operationellen und FDIR-Fähigkeiten der KARS-Software abgeleitet.

4.1.2 Funktionale Architektur eines autonomen Raumfahrzeugs

Die Applikationsprozesshierarchie ist ein zentrales Element der funktionalen Architektur des autonomen Raumfahrzeugs. Die in Bild 4.2 gezeigten und bei aktuellen Satelliten angewandten modularen hierarchischen Applikationshierarchie ist bereits auf die Implementierung einer flexiblen verschiedenen Missionanforderungen einfach anzupassenden funktionalen Architektur ausgerichtet. Die Applikationen kommunizieren über die standardisierte Schnittstellen basierend auf Telekommandopaketen, Telemeriepaketen sowie Eventberichtspaketen, welche über die Schnittstellenfunktionen abstrahiert werden. Die Applikationen können zusammen auf einem zentralen Rechner innerhalb eines gemeinsamen Rahmensystems oder verteilt auf mehreren Rechnern, welche über verschiedene Kommunikationsschnittstellen verbunden sind realisiert werden. Die Applikationsebene abstrahiert die physikalische Implementierung von der Nutzersicht.
Diese bewährte modulare Architektur basierend auf standardisierten, abstrahierenden Schnittstellen beinhaltet konzeptionell bereits heute die Erweiterung von den im Raumfahrzeug integrierten Applikation und deren Hierarchie auf die im Bodensegment insbesondere den Mission Operation Segment angelagerten funktionellen Gruppen wie Missionskontrollsystems und dem Missionsplanungssystem gezeigt. Im klassischen Raumfahrzeugbetrieb ermittelt das Missionsplanungssystem basierend auf eingehenden Orders die Kommandosequenz des Missionsplan und stellt diesen zur Freigabe für die Übermittlung an das Raumfahrzeug dem Mission Controller im Mission Operation Systems des Bodensegments zur Verfügung. In oben gezeigtem Bild wir diese Funktion durch das Missionskontrollzentrum sowie die Betriebsmannschaft inklusive des Regelwerk der Betriebsproze-

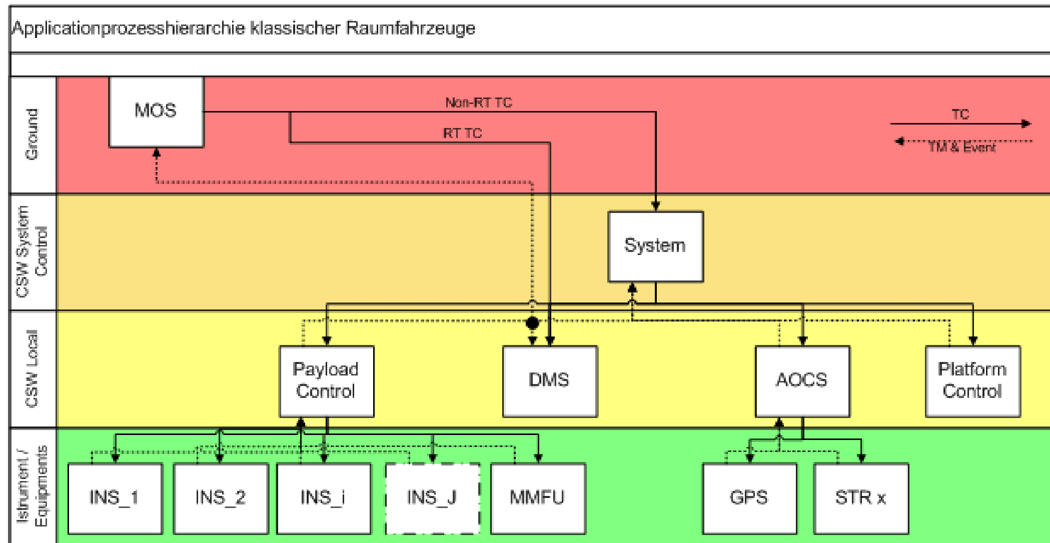


Figure 4.2: Klassische Applikationsprozesshierarchie

duren bereitgestellt. Im der oben gezeigten Applikationshierarchie werden diese Funktionsgruppen durch die MOS Applikation stellvertretend abgebildet. Bei einer vollständigen Implementierung des Space Link Extended (SLE) Protokolls zwischen Spacecraft und Missionsbetriebzentrums über die Empfangsstationen bis auf die Paketebene könnte dies bereits heute realisiert werden. Dies würde eine weitgehend autonome Steuerung klassischer Raumfahrzeuge vom Bodensegment aus ermöglichen.
Basierend auf dem abstrahierenden Charakter dieser funktionalen Architektur in Bezug auf die physikalischen Schnittstellen kann durch eine *einfache* Verlagerung dieser bisher im Bodensegment angelagerten Applikationen in das Raumfahrzeug eine einfache Erweiterung der Architektur für autonome Raumfahrzeuge erreicht werden. Bei zunehmender Autonomie des Raumfahrzeugs werden ein eine Planungsapplikation und eine dem Mission Controller entsprechende koordinierende und überwachende Supervisor Applikation ergänzt.
Diese erste durch einfachen Transfer von bisher im Bodensegment angelagerten Funktionen in das Raumsegment erzeugt eine zusätzlichen Hierarchieebene innerhalb der funktionelle Architektur des Raumfahrzeugs. Berücksichtigt man, dass innerhalb der Applikationsarchitektur für klassische Raumfahrzeuge die Systemapplikation die Überwachungs- und Koordinierungsfunktion für die übergreifende Systemsteuerung übernimmt und dass die Planungs-Applikation keine direkten Schnittstellen zu den anderen Applikationen des Raumfahrzeugs, zumindest keine die nicht über die Systemapplikation sichergestellt werden kann, hat, ist eine weiterer Optimierungs und Konsolidierungsmöglichkeit gegeben. Somit ergibt sich die in folgendem Bild dargestellte optimierte Applikationshierarchie des autonomen Raumfahrzeugs, welche die ausgezeichnete Bereitschaft der Ausgangsarchitektur für die Anwendung in hoch autonomen Raumfahrzeugen unterstreicht. Im Rahmen der Konsolidierung und Optimierung der funktionalen Architektur und deren Anforderungen in der ersten Phase des KAR Projektes sollte die Verschmelzung der Supervisor Applikation mit der bestehenden Systemapplikation noch einmal auf volle Konsistenz und Eindeutigkeit in Bezug auf Schnittstellen geprüft werden.
Aus diesen Betrachtungen wurden die Anforderungen an die Modularität, Grad der Wiederver-

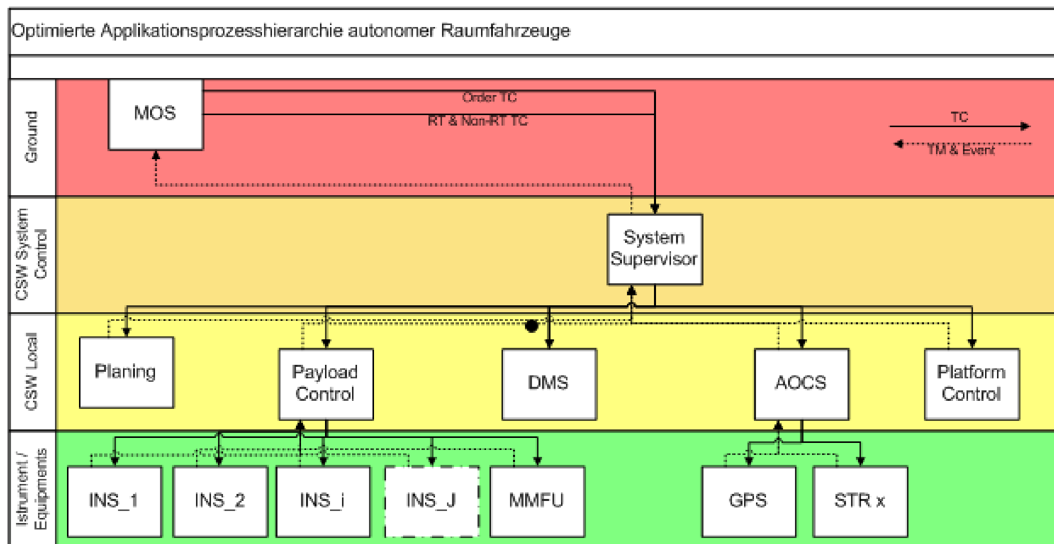


Figure 4.3: Applikationsprozesshierarchie autonomer Raumfahrzeuge

wendbarkeit und Erweiterbarkeit der KARS-Software abgeleitet.

4.1.3 Die funktionalen Charakteristik der Raumfahrzeugarchitektur

Das Gesamtsystembetriebskonzept des von Airbus DS entwickelten autonomen Raumfahrtsystems und dessen Kontrollers ist durch die folgenden Hauptmerkmale gekennzeichnet:

- Effektive und applikationsorientierte Kommandierbarkeit, welche die einfache und modulare Kontrolle des autonomen Raumfahrzeugs und seiner einzelnen Geräte, Einheiten und funktionalen Gruppen entweder vom Boden oder von den integrierten Bordkontrollsystemen erlaubt

- Konfigurierbare detaillierte und modular Beobachtbarkeit; welche eine genaue und effektive Erfassung der betrieblich relevanten Informationen des autonomen Raumfahrzeugs an Bord wie auch für einen Nutzer im Bodensegment ermöglicht

- Systematische und kontrollierte Verwaltung und Einstellung der Satellitenkonfigurations inklusive regelbasierte Umkonfiguration der Satellitenelemente

- Effektive Dienste zur Verwaltung und Bedienung von Ablaufkontrollfunktionen für zeit, orbit position und eventbestimmte Abläufe innerhalb des autonomen Systems.

- Autonome Planung der bordseitigen Aktivitätsabläufe und Sequenzen: Die in klassischen Raumfahrzeugen im Bodensegment erstellten Ablaufsequenzen werden zur Sicherstellung der Missionsziele des autonomen Raumfahrzeugs basierend auf Auftragsvorgaben (Orders) vom Bodensegment an Bord bestimmt und gegebenenfalls basierend auf Erkenntnissen der Bordsysteme unter Überwachung des System Supervisor Applikation angepasst.

- Effektive Datenpaket basierte Verteilung der Nutzdaten an Bord mittels konfigurierbarer Systemdienste zur Nutzung innerhalb des Nutzdatensegmentes sowie zu Speicherung, Übertragung zum und Auswertung im Bodensegment.

- Systematische, hierarchische und konfigurierbare Fehlerbehandlungsmethoden und -dienste zur Identfikation, Isolation und Wiederherstellung eines sicheren operationellen Zustandes mit dem Ziel der Wiederaufnahme des nominellen Betriebs für möglichst viele Fehlerzustände. Die SW basierten Fehlerbehandlungssyteme des autonomen Raumfahrzeugs werden durch unabhängige und robuste HW basierte Überwachungsfunktionen des zentralen Bordrechners geschützt.

Ein zentraler Aspekt und damit eine zentrale Anforderung an die zu erstellende Systemsoftware eines autonomen Raumfahrzeugs sind die Kompatibilität zu Multimissionsbetriebssysteme sowie die Sicherstellung eines effektiven Missionsbetriebs. Deshalb sind die Schnittstellen für den Satellitenbetrieb sowie die Betriebs-und Nutzungskonzepte optimal auf die operationellen und schnittsellentechnischen ECSS und Consultative Committee for Space Data Systems (CCSDS) Standards abgestimmt. Die funktionale Architektur ist in Übereinstimmung mit diesen ECSS Betriebsstandards auf einer modularen und hierarchischen Anwendungsprozeßarchitektur aufgebaut.
Neben der nominalen Nutzung des autonomen Raumfahrzeugs kommt der modularen hierarchischen, und damit Betriebsphasen und Systemkonfiguration angepassten Auslegung der Fehlerbehandlungssystems (FDIR) des Raumfahrzeugs eine zentrale Bedeutung zu. In der in folgenden Bild (s. Abb. 4.4) dargestellten FDIR Konzeption werden die höheren FDIR Ebenen 3, 4 und 5 von zentralen Applikationen bereitgestellt, während die Ebenen 2 und 1 dezentral in den Teilsystemen und Einzelfunktionen bzw. Geräten verteilt sind.

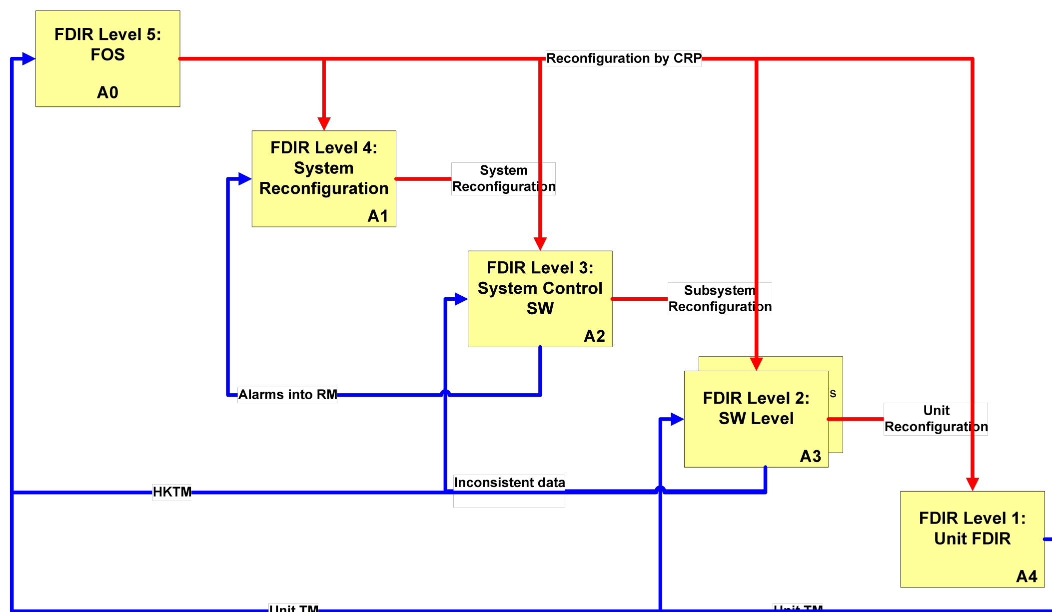


Figure 4.4: Hierarchisches FDIR Konzept

Die Unterteilung und Einstufung der FDIR-Level basiert auf der Bewertung der Fehlerauswirkungen sowie deren zeitlichen Behandlungsanforderungen. Die Einstufung wird gemäß folgender Richtlinie vorgenommen: Das FDIR Level einer Fehlerbehandlung ist niedriger, je kürzer die Zeit und je lokaler die Auswirkung einer Reaktion ist:
Das FDIR des klassischen Teils des Raumfahrzeugs insbesondere auf FDIR Level 2 stützt sich auf die Anwendung der konfigurierbaren Dienste des PUS, welche sowohl die im ECSS-E-70-41 vorgese-

Table 4.2: FDIR Level

Level	Fehlerkennzeichen	Autonomiereaktion und einfluss
1	Geräteintern (e.g. Error Detection and Correction (EDAC)) oder designgetrieben innerhalb einer dedizierten Funktion (e.g. majority voting)	nur beobachtbar
2	Fehlerlokalisierung und behandlung auf Payload oder Teilsystemebene, wobei die Fehlerbehandlung keinen Einfluss auf die Resourcenverteilung außerhalb des Teilsystems hat und keine Änderung des Hauptbetriebszustandes des Teilsystems notwendig ist.	Geräte oder Funktionsab- und -umschaltung innerhalb des Teilsystems
3	Fehleridentifikation innerhalb eines Teilsystems oder durch die Systemüberwachung, welche eine Fehlerbehandlung ohne Änderung der Resourcenverteilung- oder Gesamtresourcen im System nicht möglich ist.	Änderung von Teilsystembetriebszuständen, Umplanung der Aufgabenabfolgen mit dem Ziel der Wiederaufnahme der Missionsaufgabe oder Sicherung des Satelliten bei schwerwiegenden Fehlerfällen
4	Fehleridentifikation innerhalb des Sicherheitsüberwachungssystems die auf eine Gefährdung des Raumfahrzeugs ohne konkrete Fehlerlokalisierung, innerhalb kritischer Geräte oder Ausführungsfehler der SW hinweisen	Umkonfiguration des Zentralrechners und Sicherung der Raumfahrzeugs
5	Fehler, basierend auf Langzeitauswertungen oder nicht autonom wieder herstellbarer höherer Systemzustände, welche eine Missionswiederaufnahme durch bodensegmentgestützte Wiederherstellungsprozeduren erfordern.	keine

henen Überwachungs- und Fehlerbehandlungsdienste beinhaltet wie auch die Erweiterung auf einen FDIR Supervisor Service. Dieser Service erlaubt eine funktionale Gruppierung und Koordinierung verschiedener Einzelüberwachungskriterien unter Berücksichtigung betrieblicher Zustände und Konfigurationen innerhalb eines Teilsystems aber auch aus Sicht des zentralen Systemkontrollers. Die Fehlerbehandlungsdienste stellen konfigurierbare Dienste zur Fehlerisolation, Umkonfiguration wie auch zur Wiederherstellung höherer Systemzustände. So wie die nominalen Funktionen dezentral auf die Applikationen verteilt sind, sind auch die Fehlerbehandlungsmechanismen in jeder Applikation verfügbar. Die eingebauten FDIR Mechanismen sind stellen sicher, dass:

- Während der Ausführung einer primären Fehlerbehandlung innerhalb der Applikation keine weitere Fehlerbehandlungen durchgeführt werden, um vorschnelle Reaktionen auf Sekundäreffekte auszuschließen
- Die primäre Fehlerreaktion nach der ersten Ausführung gesperrt wird. Dies gründet auf der Grundregel, dass ein Wiederauftreten eines entsprechenden Ausfallsymptoms eine andere,

stärkere Reaktion als die erste Reaktion bedarf, um das Problem zu lösen. Die Wiedereinbindung erfolgt nur auf ausdrückliche Aufforderung durch ein Kommando der höheren Überwachungsinstanz, typischerweise des Bodensegmentes. Dies gilt für die Fehlerbehandlung in einer Applikation aber wird prinzipiell auch zwischen den FDIR Ebenen angewendet.

- Eine Fehlerbehandlung auf Level N führt dazu, dass die Fehlerbehandlung auf den niedrigeren FDIR Level gemäß der FDIR Hierarchie (d.h. N-1, N-2 usw.) autonom abgeschaltet bzw. der neuen Konfiguration angepasst werden.
- Die Fehlerbehandlung auf den höheren Leveln (d.h. N+1, N+2...) bleiben aktiviert und damit bereit weitere stärkere Reaktion auszulösen, insbesondere auch die Reaktion das Raumfahrzeug in seinen sicheren Betriebszustand zu überführen.

Diese FDIR Mechanismen in Verbindung mit der FDIR Hierarchie garantiert eine hohe bordeigene Autonomie des Raumfahrzeugs für alle möglichen klassischen Fehlerursachen und stellt einen effektiven Schutz gegen ungewollte hin- und herschaltende FDIR Reaktionen bereit.
Die Erweiterung der bordseitigen Autonomie kann nicht nur auf der Seite der nominalen Autonomie sondern auch zur Fehlerbehandlung zur Einführung neuer Methoden in der System Supervisor Applikation wie regelbasiert oder modelbasierte Fehleridentifikationsmethoden und erweiterte Fehlerbehandlungsreaktionen wie bordseitige Kontrollprozeduren mit komplexeren Fähigkeiten zur Ablaufsteuerung wie logische Entscheidungen, bedingte Warteschleifen, Algorithmik oder parametrisierbare Kommandos bereitstellt.
Aus diesen Überlegungen wurden die Anforderungen an die Funktionalitäte und (Re-)Konfigurierbarkeit der KARS-Software abgeleitet.

4.1.4 Operationelle Aspekte

Operationelle Szenarien

In einem Szenario mit einer Robotik-Nutzlast auf einem Satelliten gibt es Abhängigkeiten von Nutzlast/Roboterarm und der Satellitenplattform. Der Roboterarm beeinflusst durch seine Bewegungen die Lage und den Schwerpunkt der Plattform. Umgekehrt beeinflusst natürlich auch die Plattform durch ihre Möglichkeiten der Veränderung von Lage und Position durch das Attitude and Orbit Control (AOCS) die Roboterarmbewegungen. Um bestimmte Absolutbewegungen des Roboterarms auszuführen, beziehungsweise Relativbewegungen des Roboterarms auf dem Servicingsatelliten zum Target zu erzielen, muss die Robotersteuerung die Lageregelung übernehmen. Dazu benötigt die Robotersteuerung Informationen über die Lage des Satelliten. Die Aktoren des AOCS werden dabei abgeschaltet. Es sollte dazu aber keine kompletten dedizierten AOCS Algorithmen innerhalb der Nutzlast geben. Funktionen des Plattform-AOCS sollen nicht in der Nutzlast kopiert oder implementiert werden, vielmehr muss eine Befehls und Datenschnittstelle zwischen beiden erstellt werden und in der Datenbank bzw. im Datenpool berücksichtigt werden. Über dieses Interface sollen Informationen über Kräfte, Drehmomente, Schwerpunktänderungen etc. zwischen Plattform-AOCS und Roboter ausgetauscht werden.
Abhängig von dem Autonomie-Modus, der zwischen Supervisor auf der Plattform und der Robotik-Software auf der Nutzlast vereinbart wird, muss auf dem Nutzlast-Rechner eine High-Level Rendezvous-

Software erstellt werden, die den Supervisor auf der Plattform über notwendige Plattformlage- bzw. Positionsänderungen informiert und das AOCS entsprechend ansteuert (s. Abb. 4.5). Mögliche Autonomie-Modi während des Rendezvous müssen definiert und implementiert werden In KARS sollten vor allem die Konzepte für die Autonomie-Verfahren entwickelt werden. Wenn hohe Stufen der Autonomie gefordert sind, kann dies auch in der Entwicklung von komplexen Algorithmen resultieren. Es war nicht Ziel von KARS , autonome Szenarien und Planungen selbst zu entwickeln, sondern nur die zugehörigen Mechanismen für eventuelle Pilotanwendungen zur Verfügung zu stellen.

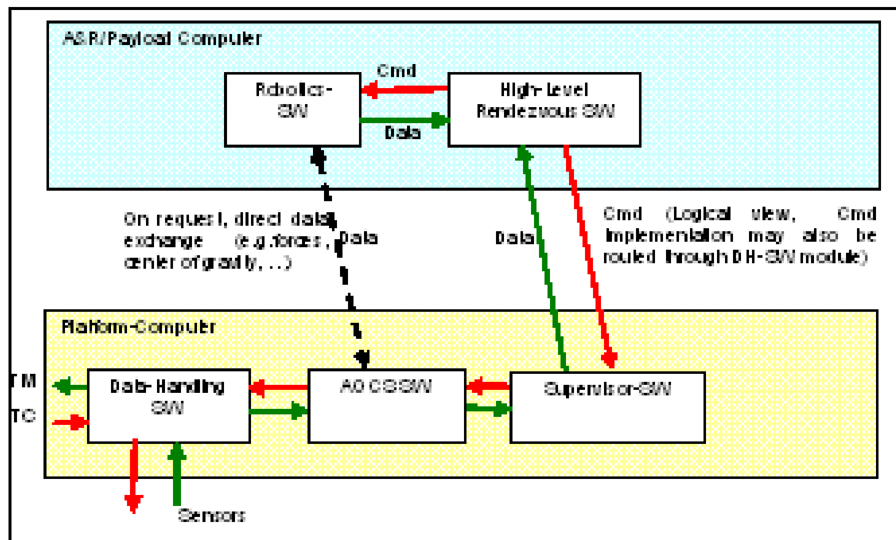


Figure 4.5: Beispiel einer komplexen Nutzlast, bestehend aus Robotics und Rendezvous-Sensor sowie -Software

Basis PUS-Dienste

Generell lassen sich Software-Dienste wie folgt charakterisieren:

- Ein Dienst ist eine Software-Repräsentation von fachlicher Funktionalität.
- Ein Dienst ist in sich abgeschlossen (autark) und kann eigenständig genutzt werden.
- Ein Dienst ist in einem Netzwerk verfügbar.
- Ein Dienst hat eine wohldefinierte veröffentlichte Schnittstelle (Vertrag). Für die Nutzung reicht es, die Schnittstelle zu kennen. Kenntnisse über die Details der Implementierung sind hingegen nicht erforderlich.
- Ein Dienst ist plattformunabhängig, d. h. Anbieter und Nutzer eines Dienstes können in unterschiedlichen Programmiersprachen auf verschiedenen Plattformen realisiert sein.
- Ein Dienst ist in einem Verzeichnis registriert.
- Ein Dienst ist dynamisch gebunden, d.h. bei der Erstellung einer Anwendung, die einen Dienst nutzt, braucht der Dienst nicht vorhanden zu sein. Er wird erst bei der Ausführung lokalisiert und eingebunden.

- Ein Dienst sollte grobgranular sein, um die Abhängigkeit zwischen verteilten Systemen zu senken.

Im ECSS-E-70 ist folgende Definition zu finden:
set of on-board functions offered to a service user that can be controlled and monitored (e.g. by a ground system) through a well-defined set of service requests and reports, ECSS-E-70-41A, (PUS)
Das KARS-Software bietet einen Satz an Basis-PUS-Diensten (System Support Services) an, die von Anwendungskomponenten bzw. von der Bodenstation genutzt werden können. Anwendungsspezifische PUS-Dienste werden in den jeweiligen Komponenten selbst implementiert.
Der PUS definiert folgende Core-Services

- Service 1: Telecommand Verification Service
- Service 2: Device Command Distribution Service
- Service 3: Housekeeping and Diagnostic Data Reporting Service
- Service 4: Parameter Statistics Reporting Service
- Service 5: Event Reporting Service
- Service 6: Memory Management Service
- Service 8: Function Management Service
- Service 9: Time Management Service
- Service 11: On Board Operations Scheduling
- Service 12: On Board Parameter Monitoring
- Service 13: Large Data Transfer
- Service 14: Packet Forwarding Control Service
- Service 15: On Board Storage and Retrieval
- Service 17: Test Service
- Service 18: On Board Operations Procedures
- Service 19: Event/Action Service

4.1.5 Anwendung von Software-Standards und Compliance zu ECSS

In Phase 0/A wurde eine Analyse der vorgeschriebenen Standards durchgeführt und ein Projekt-Tailoring der ECSS Standards durchgeführt. Das Resultat war ein KARS Software Development Plan [ASI13f], der die Basis für alle KARS Softwareentwicklungen und Software-Modifikationen war.

4.1.6 Software-Architekturprinzipien

In der Luftfahrt, der Automobilindustrie und der Automatisierungstechnik wurden in den letzten Jahren enorme Anstrengungen unternommen, um die wachsenden Softwarekomplexität beherrschbar zu machen. IMA, AUTOSAR und OMAC sind die entsprechenden Initiativen. All diesen Initiativen verfolgen ähnliche Ziele:

- Systemfunktionen und Schnittstellen sollen standardisiert werden
- Software soll portabel sein und beliebig auf eine Anzahl gleichartiger Ausführungssysteme verteilt werden können
- Es sollen die zukünftigen Anforderungen bezüglich Verfügbarkeit, Sicherheit und Softwareaktualisierung erfüllt werden
- Komponenten unterschiedlicher Kritikalität sollen nebeneinader existieren können
- Die Systeme soll trotz der gestiegenen Produkt- und Prozesskomplexität beherrschbar bleiben
- Die Systeme sollen kostengünstig skaliert werden können
- Die Wartbarkeit soll über den gesamten Produktlebenszyklus gewährleistet sein.

Zur Erreichung dieser Ziele bieten sich komponenten-basierte Ansätze an. In der angewandten Informatik ist die Komponentenbasierte Entwicklung (engl.: Component Based Development - CBD oder auch Component Based Software Engineering - CBSE) ein aus früheren Ansätzen entwickeltes Paradigma. Der Grundgedanke komponentenbasierter Entwicklung ist die Unterteilung von Anwendungen in wiederverwendbare Komponenten, um möglichst wenig Code neu programmieren zu müssen. Mit der Zeit kann so ein *Komponentenmarktplatz* entstehen, aus dem heraus Anwendungen nach dem Baukastenprinzip zusammengestellt werden. Zusätzliche Komponenten müssen nur für Funktionalität entwickelt werden, für die es bisher keine Implementierung gibt. Vorteile sind neben einer Zeitersparnis bei der Entwicklung auch eine erhöhte Qualität der Komponenten durch eine große Nutzeranzahl und verschiedene Anwendungsszenarien, die automatisch als Testszenarien dienen. In einem Softwaresystem werden in der Regel Annahmen über einen Kontext impliziert, in dem das System funktioniert. Die CBSE verlangt, dass alle diese Annahmen explizit definiert werden, damit das System in verschiedenen Kontexten (von Dritten) wiederverwendet werden kann. Damit die Komponenten miteinander agieren können, nutzen sie in eine Middleware, die Basisfunktionen zum Austausch von Nachrichten, zur Überwachung der Kommunikation, zur Einbindung/Registrierung von neuen Komponenten etc. bereitstellt.
Die konsequente Umsetzung des Komponentengedankens ermöglicht es, Komponenten auszutauschen, auf eine andere Ausführungseinheit zu verlagern oder eine weitere Komponente hinzuzufügen (s. Abb. 4.6).
Ähnlich wie in der Automobil- und der Luftfahrtindustrie gibt es auch in der Raumfahrt Überlegungen zu einer Softwarearchitektur, die zum einen die gestiegenen Anforderungen nach Funktionalität, Modularität, Skalierbarkeit, etc. erfüllt und gleichzeitig aber auch Aspekte wie funktionale und operationelle Sicherheit (Safety and Security) berücksichtigt.

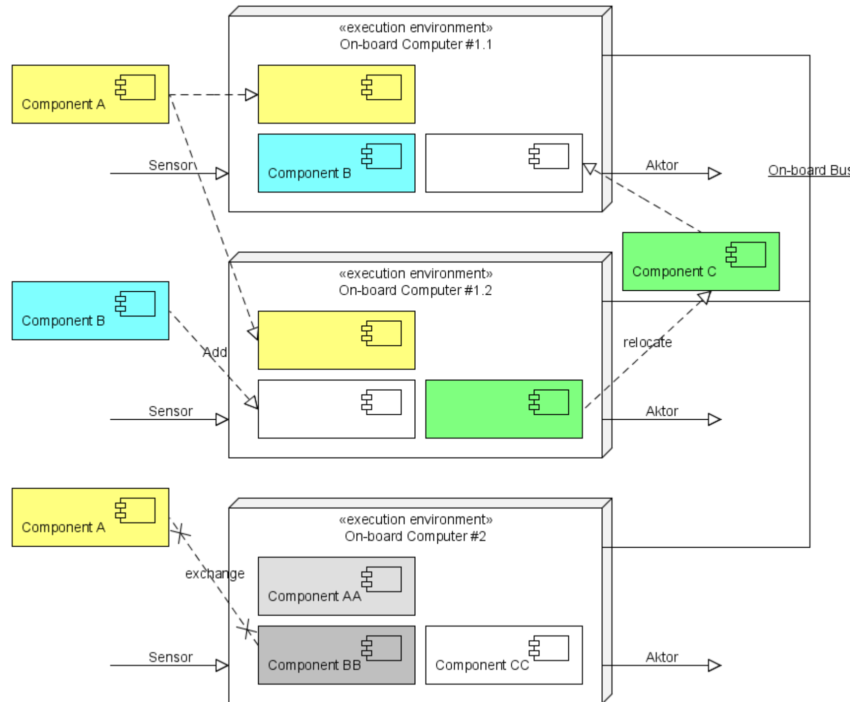


Figure 4.6: Komponenten-basierte Entwicklung

So wird z.B. in einem White Paper des Networking and Information Technology Research and Development (NITRD) eine Architektur vorgeschlagen wie sie in Abb. 4.7 dargestellt ist. Kernelemente dieser Architektur sind eine domänen-spezifische *Middleware* und ein Echtzeit-Betriebssystem mit einem *Separation Kernel*.
Die ESA schlägt in dem IMA-SP Programm eine ähnliche Architektur vor (s. 4.8).
Basierend auf den Überlegungen in den vorangegangenen Kapiteln und den Entwicklungen in anderen industriellen Domänen wurde eine Software-Architektur entwickelt, die weitgehend identisch ist, mit den Architekturkonzepten, die in den Abb. 4.7 und 4.8 dargestellt ist (s. Abb. 4.9).
Die KARS-Software setzt auf einem Betriebssystem auf, das Time and Space Partitioning unterstützt und somit die Koexistenz von Komponenten unterschriedlicher Kritikalität auf einer Hardwareplattform unterstützt. Die Software unterscheidet zwischen zwei Ebenen: der Anwendungsebene und der Middleware.
Die Middleware untergliedert sich wiederum in zwei Ebenen, den Component Support Services (CSS) und dem Operating System Abstraction Layer (OSAL). Die CSS stellt alle Funktionen bereit, damit Komponenten untereinander und mit Betriebssystem- und Harware-Elementen kommunizieren können. Der OSAL ermöglicht es, das Betriebssystem auszutauschen, ohne, dass damit die Komponenten in der Anwendungsebene oder die CSS geändert werden müssen.
Die Anwenungsebene untergliedert sich ebenfalls in zwei Teile: Komponenten, die missionsübergreifende Dienste bereitstellen (System Support Services (SSS)) und die eigentlichen Anwendungskomponenten wie z.B. AOCS, Thermal Control System (TCS) oder Komponenten zur Steuerung eines Roboterarms.

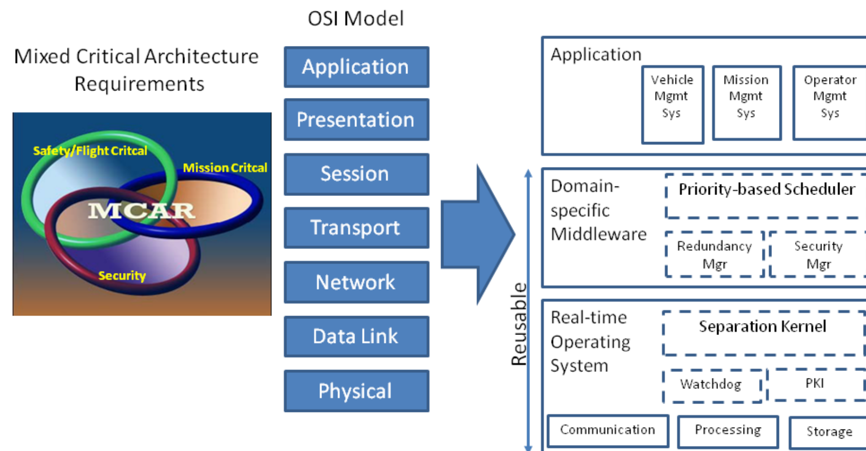


Figure 4.7: NITRD Architektur (Quelle: NITDR)

4.1.7 Implementierungs Aspekte

Die Softwareentwicklung der letzten Jahre ist gekennzeichnet durch einen Übergang von prozeduralen Ansätzen hin zu objektorientierten Ansätzen. Die Vorteile der Objektorientierung sind verbesserte Wartbarkeit und Wiederverwendbarkeit des statischen Quellcodes. In der Welt der Embedded Systeme im Allgemeinen und in der Raumfahrt im Besonderen bestand lange Zeit ein Vorbehalt gegenüber dem Einsatz von objektorientierten Sprachen wie C++, da die harten Echtzeitforderungen aufgrund des *Overheads* auf den vergleichsweise schwachen Prozessoren oft nicht erreicht werden konnten. Durch den Einsatz neuerer Prozessoren und besserer Compiler ist man heute in der Lage, diese harten Echtzeitanforderungen auch mit objektorientierten Sprachen zu erfüllen.
Im Rahmen des Projektes wurde untersucht, inwieweit C++ für KARS verwendet werden könnte. Die Analyse ergab, dass C verwendet werden soll, da der Vorteil von C++ durch die vielen Einschränkungen nicht gegeben ist [ASI12a].
Durch den Einsatz von Unified Modelling Language (UML) wird der objektorientierte Ansatz beim Entwurf der Software verfolgt. Durch die Möglichkeit Code-Rahmen aus der grafischen Beschreibung des Systems zu generieren wird sichergestellt, dass Information, die auf Systemebene eingegeben wird, über Tags in den Quelltext übertragen wird. Entsprechend der in Airbus DS geltenden Codierungsrichtlinien [ASI13b] sind die Programmierer angehalten den Quelltext mit aussagekräftigen Kommentaren zu versehen.
Betriebsystem spezifische Softwareanteile wurden in einer leicht austauschbaren Schicht zusammengefasst, dem sog. OSAL. Dadurch wurde sichergestellt, dass die Software portable bleibt. Die Portabilität wurde während des nächtlichen Build-Prozesses für ausgewählte Zielplattformen laufend überprüft.

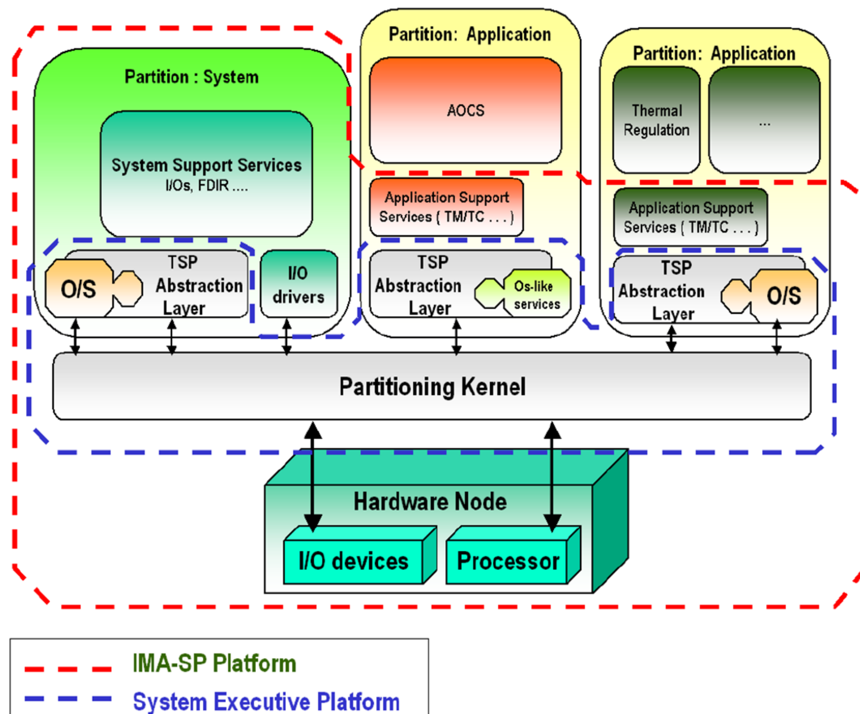


Figure 4.8: IMA-SP Architektur (Quelle: ESA)

4.1.8 Aspekte der Zielhardware

Die Entwicklung der Rechnerarchitektur wird seit Jahren durch die fortschreitende Integrationstechnologie getrieben. Nachdem lange Zeit eine kontinuierliche Leistungssteigerung durch höhere Integrationsdichten und vor allem durch die Erhöhung der Taktfrequenz ermöglicht wurde, scheinen diese Parameter weitgehend ausgereizt. Heute sind praktisch alle Funktionsblöcke einer von Neumann Architektur in einem Chip integriert und eine weitere Steigerung der Taktfrequenz ist aufgrund der daraus resultierenden Verlustleistung nicht mehr wirtschaftlich. In Verbindung mit der fortschreitenden Reduzierung der Strukturbreiten ermöglicht jetzt die Multi-Core Technologie eine weitere Leistungssteigerung bei moderaten Taktfrequenzen durch die Nutzung der in vielen Anwendungen vorhandenen Parallelität. Hierdurch können wichtige Erfahrungen aufgrund der rasanten Entwicklung der Parallelrechnertechnologie nun innerhalb eines Bausteins genutzt werden. Im Bereich der eingebetteten Systeme ist diese Tendenz inzwischen ebenfalls angekommen und erste Anbieter drängen mit Systemen auf den Markt. Für die zukünftige Entwicklung von hochleistungsfähigen Rechnersystemen ist deshalb die Beherrschung der Multi-Core Technologie eine unverzichtbare Vorraussetzung.
Diese Entwicklung trifft - zwar zeitverzögert - auch die Raumfahrt bzw. ist dort scheinbar schon angekommen, wie an den Anstrengungen des amerikanischen OPERA Programms zu sehen ist, in dessen Rahmen im MAESTRO Projekt mit dem Tilera Chip gearbeitet wird. Auch in der Raumfahrttechnik ist die Tendenz hin zu einem immer höheren Bedarf an Rechenleistung eindeutig vorgegeben. Beispiele sind die Vorverarbeitung (on-the fly) hochvolumiger Datenströme aus anspruchsvollen Experimenten und Nutzlasten oder eine leistungsfähige On-Board Verarbeitung, um z.B. die begrenzte Downlink Kapazität optimal zu Nutzen.
Für zukünftige Raumfahrtanwendungen bietet die aktuelle technologische Entwicklung im Bereich

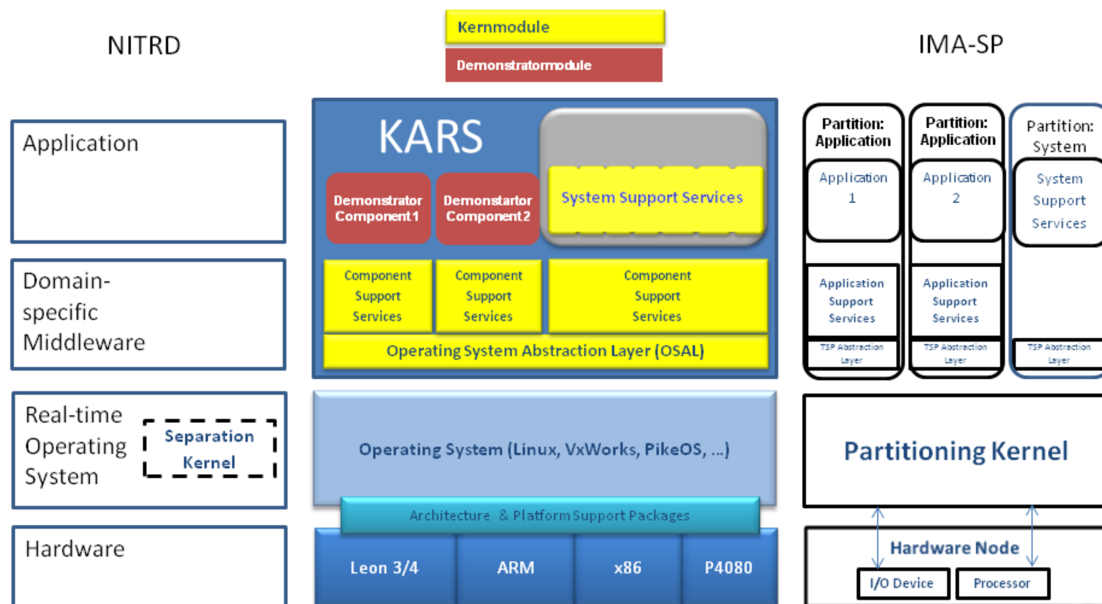


Figure 4.9: KARS Software-Architektur

der Multi-Core Prozessoren neben der höheren Rechenleistung bei gleichzeitiger Reduzierung der Verlustleistung eine Reihe weiterer wichtiger Vorteile. So ist die in einigen Prozessoren primär zur Reduzierung der Verlustleistung eingesetzte Silicon On Insulator (SOI) Technologie praktisch immun gegenüber dem Latch-Up Effekt und auch wesentlich unempfindlicher gegenüber anderen Strahlungseffekten. Zudem können bei geeigneter Auslegung der Hardwarestruktur die redundanten Strukturen eines Multi-Core Prozessors auch für die effektive Beherrschung der immer noch vorhandenen SEU Effekte genutzt werden.
Die Verwendung von Multi-Core-Prozessoren hat erhebliche Auswirkungen auf die Software-Architektur: die Software muss ebenfalls parallelisiert werden, um die darunterliegende Hardware ausnutzen zu können. Nur so ist es möglich auf der Anwendungsebene von der höheren Leistungsfähigkeit der Hardware zu profitieren.
Bei der Entwicklung von KARS wurde diesem Trend Rechnung getragen und die Software auch auf dem Freescale P4080 (8-Kern CPU) getestet.

4.1.9 Entwicklungsumgebung

Die Entwicklung der KARS-Software erfolgte mit modernen, state-of-the-art Entwicklungswerkzeugen. Für die Entwicklung der KARS-Software wurde das klassische V-Modell zugrunde gelegt. Die einzelnen Phasen des V-Modells werden durch entsprechende Software-Werkzeuge unterstützt.

- Anforderungsanalyse: TEAMsuite - RM / Enterprise Architect (Sparx Systems)
- Grob- und Feinentwurf in UML: Enterprise Architect
- Implementierung: Eclipse IDE
- Erzeugen von ausführbarem Code: GNU comiler und Make Utilities

- Automatisierte Erzeugung von Binärdateien: Jenkins
- Erzeugen von Dokumenten: Latex / doxygen
- Integrations-/Systemtests: TEAMsuite - TM (inhouse Test Manager)

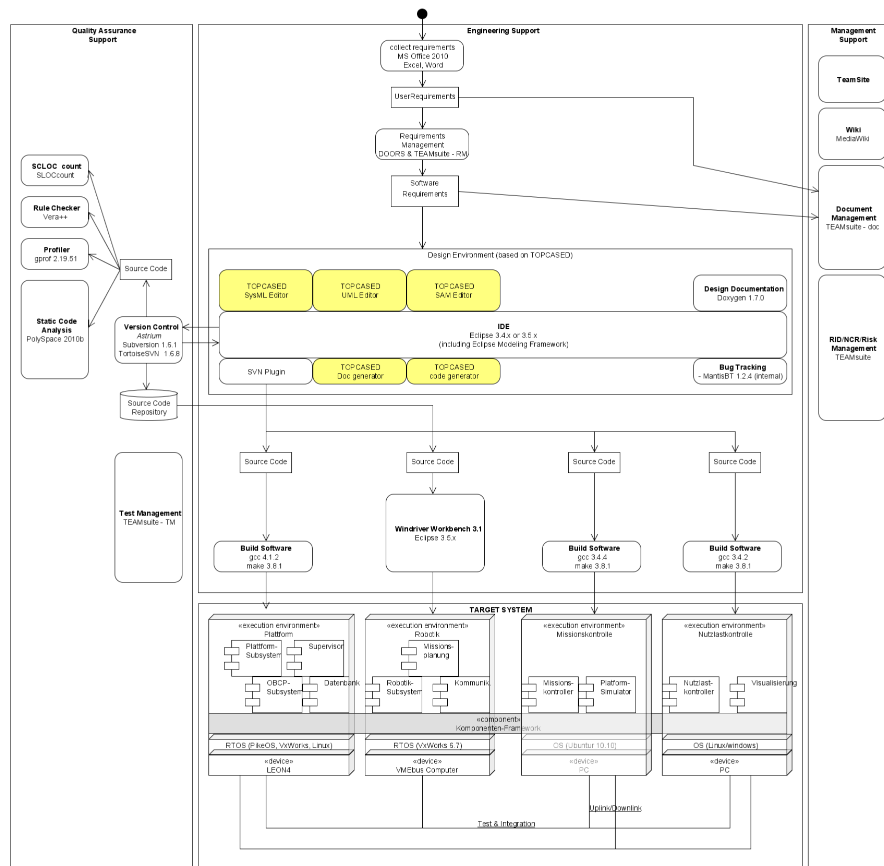


Figure 4.10: Entwicklungsumgebung

Die kontinuierliche Überprüfung der Software-Qualität wurde durch entsprechende Software-Werkzeuge unterstützt:

- Software-Komplexität: SLOCCount/CLOC
- Einhaltung der Codierungs-Richtlinien: Understand for C/sonar
- Statische Code-Analyse: Polyspace/FramaC
- Abschätzung des Laufzeitverhaltens: gprof
- Unit-Testing: VectorCAST
- Fehlerverfolgung: Redmine

Für die team-interne Kommunikation und zur Unterstützung des Managements standen entsprechende Werkzeuge zur Verfügung:

- Dokumenten-/Informationsaustausch zwischen den Projektpartnern und dem Auftraggeber: Teamsite
- Dokumentenverwaltung: TEAMsuite
- Verwaltung von AI, RIDs, NCRs, Risikotabellen: TEAMsuite
- Interne Kommunikation: Wiki und Chat-Tool

Entsprechend Abb. 4.10 kann die KARS-Software auf einem heterogenen Zielsysteme bestehend aus

- x86-Plattform mit Linux-Betriebsstem
- x86-Plattform mit VxWorks-Betriebsstem
- ARM-Plattform mit Linux-Betriebsstem
- LEON4 Plattform mit Linux bzw. PikeOS

ausgeführt werden.
Die Kernmodule wurden für die folgenden vier Zielsysteme: x86/Linux, ARM/Linux, x86/VxWorks, LEON4/PikeOS kompiliert. Alle anderen Module wurden lediglich für das Zielsystem kompiliert auf dem sie ausgeführt werden.

Table 4.3: Software Produktbaum

Item	**Ref.**	**Version**
1: KARS:	SYS	KAR_QR_20140131
..100000: Core Modules:	CM	1.5
......112000: Middleware:	MW	1.5
........112100: OSAL:	OSAL	1.5
........112200: MessageLib:	MessageLib	1.5
........112300: CommLib:	CommLib	1.5
........112400: SystemConfiguration:	CFG	1.5
......113000: Basic Components:	BCO	1.5
........113100: IOHandler OBC:	IOH	1.5
........113200: DataManagement:	DM	1.5
........113300: EventHandler:	EVH	1.5
........113400: OBCPHandler:	OBCPH	1.5
........113500: MissionTimeLineHandler:	MTH	1.5
........113600: LoggingHandler:	LOH	1.5
........113700: MACRO Procedures:	MACRO	1.5
....120000: Specific Components:	SCO	1.5
......121000: Supervisor:	SUV	1.5
......122000: MissionPlanner:	MPL	1.5

4.2 Softwarearchitektur

4.2.1 Überblick

Teil der Gesamt-Software eines Raumfahrzeugs sind unter anderem mehrere Softwareelemente, mit unterschiedlichem *criticality level*. Die Platform-Kontrollsoftware ist *Safety/Flight Critical* während die Steuerung der unterschiedlichen Instrumente zur Gewinnung von wissenschaftlichen Daten lediglich *Mission Critical* ist. In jüngster Vergangenheit rückt auch immer mehr das Thema Sicherheit/Security in den Vordergrund.
All diese Anwendungen/Komponenten müssen miteinander interagieren und nebeneinander ggf. auf der gleichen Hardwareplatform existieren ohne sich gegenseitig zu beeinflussen.
In Anlehnung an die in Kap. 4.1.6 präsentierten Architekturen wurde in KARS folgende Architektur realisiert:
Die Anwendungsebene wird durch die sog. Demonstrator-Module dargestellt. Die Demonstratormodule bilden typische Anwendungen wie z.B. AOCS nach und sind Teil der repräsentativen Umgebung in der die sog. Kernmodule getestet wurden.
Die Kernmodule beinhalten drei Elemente:

1. Middleware (Component Support Services) inkl. OSAL

2. Basis- und (missions)spezifische Komponenten (System Support Services)

Die SSS implementieren den - in der europäischen Raumfahrt verwendeten - PUS während die CSS Dienste bereitstellen, mit deren Hilfe Komponenten untereinander Daten austauschen und in standardisierter Art und Weise auf Ressourcen zugreifen, die das Betriebssystem und die Hardware zur Verfügung stellen. Eine OSAL Ebene erlaubt die Verwendung von verschiedenen Betriebssystemen. Gegenwärtig werden sowohl prioritäts-gesteuerte Betriebssystem wie Linux und VxWorks als auch Betriebssteme unterstützt, die Virtualisierung und Microkernel vereinen (z.B. PikeOS).
Diese Kernmodule stellen den eigentlichen Entwicklungsgegenstand dar, der nach den Richtlinen des ECSS-E-40 entwickelt wurde.
Die Testumgebung stellt die Bedienschnittstelle für die Kern- und Demonstratormodule dar. Sie basiert auf bereits vorhandener Software, die für die Verwendung in KARS angepasst wurde.
Tabelle 4.3 zeigt den KARS-Produktbaum mit allen Elementen, die nachfolgend kurz beschrieben werden. Diese Module (s. Abb. 4.11) bilden den Kern der entwickelte Software auf den der ECSS-40 Standard angewendet wurde.

4.2.2 Middleware - Component Support Services

Ein wesentliches Merkmal des Vorhabens war die Implementierung einer Middleware, damit Komponenten weitgehend unabhängig voneinander entwickelt werden können. Diese Middleware stellt jeder Komponente eine einheitliche Schnittstelle zur Verfügung mit der diese auf Dienste des Betriebssystems oder auf Hardware-Ressourcen zugreifen können. Als Beispiel sind hier Funktionen des Betriebssystems wie Semaphore oder Threads, aber auch Funktionen mit höherem Abstraktionsgrad wie z.B. Erzeugen eine Ereignisses und Zugriff auf den Datenpool zu nennen. Diese Middleware wird im Folgenden auch als CSS bezeichnet.

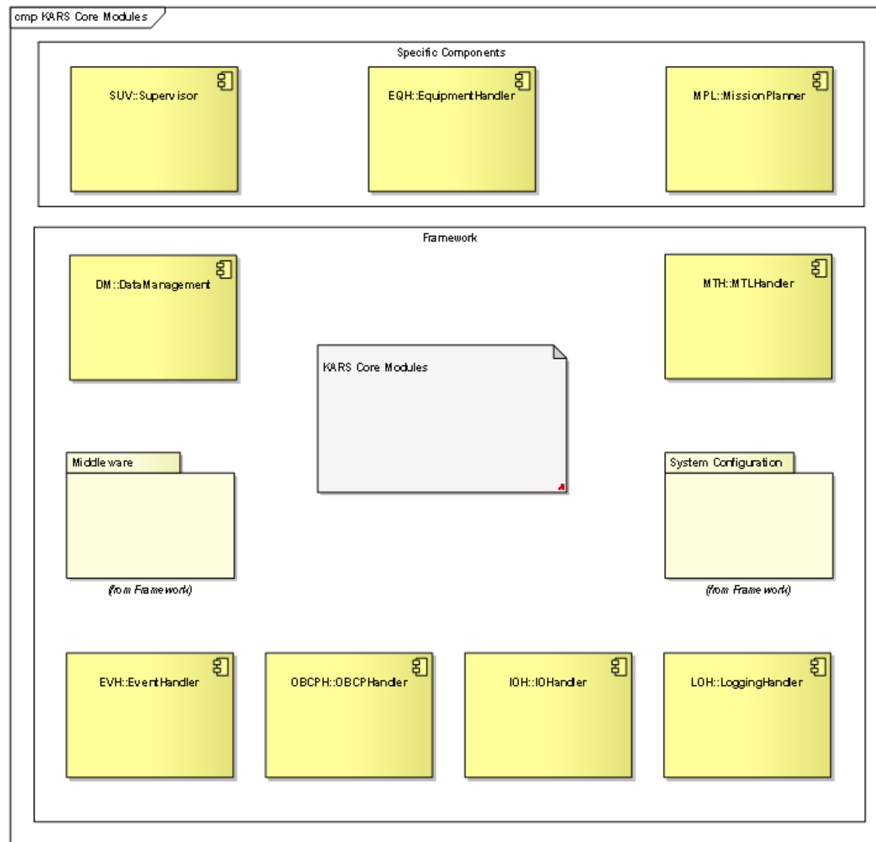


Figure 4.11: Kernmodule

Ein weiterer wichtiger Bereich innerhalb der CSS ist die Kommunikation. Die meisten Komponenten müssen in irgendeiner Form mit einem anderen Komponente oder einem Gerätetreiber kommunizieren. Dabei ist es von Vorteil, wenn man die häufigsten Arten der Kommunikation (Message Queues, Shared Memory, etc.) bereits als CSS implementiert. Auf diese Weise erhält man eine Abstraktionsschicht, das sogenannte *Communication Abstraction Layer* (CAL), zwischen der eigentlichen Komponente und der verwendeten Kommunikationsart.
Die abstrakten Kommunikations-, Syncronisation- und Taskingmechanismen werden von Betriebssystem leicht unterschiedlich implementiert. Aus diesem Grunde wurde eine Zwischenschicht zwischen der Middleware und dem Betriebsystem implementiert in dem die Abbildung der abstrakten Mechanismen auf konkrete Betriebsystemaufrufe erfolgt. Diese Zwischenbene bezeichnet man als OSAL.

4.2.3 Basis- und (missions)spezifische Module - System Support Services

IO Handler (IOH)

Die Komponente IOH dient dazu, die vom Boden empfangenen Kommandos an die jeweils adressierten Komponenten weiterzuleiten. Nur korrekt empfangene Kommandos dürfen weitergeleitet werden, daher überprüft das KommunikationKomponentes die von der Kontrollstation empfangenen Kommandos (Checksumme, Länge, ...). Die Komponente antwortet mit dem Service (1,1) oder (1,2), je nachdem, ob das Kommando korrekt empfangen wurde oder Fehler entdeckt wurden.

Daten, die der IOH von anderen Komponenten bekommt, werden entweder als PUS-Telemetrie an die Bodenstation geschickt oder an andere KARS-Komponenten auf anderen Prozessorknoten weitergeleitet.

Event Handler (EVH)

Der EVH implementiert den Service 5 (Event Reporting) und Service 19 (Event Action) des PUS. Berichte mit den im System aufgetretenen Ereignissen werden zyklisch an die Bodenstation geschickt. Andere Komponenten können über die Middleware Ereignisse auslösen, die im EVH entsprechend behandelt werden. Wenn mit einem Ereignis eine Aktion verknüft ist, dann führt der EVH die Aktion aus. Die Aktionen können einzelne Telekommandos oder komplette On-Board Control Procedure (OBCP)s sein.

Data Management (DM)

Zur Abgrenzung der verschiedenen Varianten von Datenhaltung kennt der Raumfahrt-Standard ECSS-E-ST-70-11C zwei Stufen, die im Folgenden dargestellt werden.

- Die wichtigsten Missionsdaten werden on-board gespeichert
- Alle Missionsdaten werden on-board gespeichert (wissenschaftliche Daten, Experimentdaten und Housekeepingdaten)

Die folgende Tabelle gibt die verschiedenen Autonomiestufen in Bezug auf Datenhaltung aus der ECSS-E-ST-70-11C wieder:

Level	Beschreibung	Funktion
D1	Storage on-board of essential mission data following a ground outage or a failure situation	Storage and retrieval of event reports, Storage management
D2	Storage on-board of all mission data, i.e. the space segment is independent from the availability of the ground segment	As D1 plus storage

Table 4.4: Autonomiestufen

Die Datenhaltungs-Autonomiestufe bestimmt indirekt auch die Operationsautonomiestufe. Volle Autonomie ohne Bodenkontakt kann nur bei Speicherung aller Daten on-board erreicht werden. Insbesondere für den Supervisor, der viele gespeicherte on-board-Daten verknüpfen muss, um sinnvolle Entscheidungen für autonome Aktionen zu treffen, sind die Fähigkeiten der DM-Komponente nach D2 von Bedeutung. Folgende Daten werden von dem DM gespeichert:

- Logging Information
- Sensor-/Aktordaten
- Housekeeping-Daten von Komponenten
- Interner Systemzustand

- System Configuration Vector
- System Status Vector

Die Daten werden in Tabellenform abgelegt, die Anzahl der Zeilen und Spalten ist konfigurierbar. Jedes Element in einer Tabelle erhält einen eindeutigen Identifier über den der Zugriff gesteuert wird.
Die zum Monitoring angemeldeten Sensor-/Aktor- und Housekeeping-Daten diese auf Bereichsüberschreitung geprüft und entsprechende Fehler-Zähler inkrementiert.
Die Daten im sog. System Datapool (SDP) können über den Service 140 direkt beschrieben oder gelesen werden. Elemente aus dem SDP können zu Gruppen zusammengefasst werden und über den Service 3 zyklisch an die Bodenstation geschickt werden.

Logging Handler (LOH)

Der LOH implementiert den PUS Dienst 15 (Storage and Retrieval) und speichert die Telemetriepakete in entsprechenden Packet-Stores wenn die Verbindung zur Bodenstation abreißt. Sobald die Verbindung wieder zur Verfügung steht, werden die gespeicherten Pakete an die Bodenstation geschickt.

Mission Timeline Handler (MTH)

Die Mission Timeline (MTL) ist eine Liste mit PUS-Kommandos, die an Board des Satelliten abgespeichert wird. Jedes **PUS!** (**PUS!**)-Kommando ist dabei mit einem Zeitstempel versehen. Sind Zeitstempel und aktuelle Uhrzeit identisch, wird das Kommando abgeschickt. Die kann einmalig oder zyklisch in einem vordefinierten Zeitintervall erfolgen. Der MTH verwaltet diese Liste und stellt Funktionen zur Verfügung, diese Liste zu verändern. Die umfasst: eingügen, löschen von einzelnen Kommandos und verschieben von ganzen Gruppen von Kommandos.
Nachdem eine Liste mit Aktionen definiert ist und freigegeben wurde, sorgt der MTH autonmon dafür, dass die Kommandos zu der angegebenen Zeit verschickt werden. Die Erstellung und Modifikation der MTL kann sowohl vom Boden aus als auch von der Supervisor- oder Missionsplanungs-Komponente aus erfolgen. Die Ausführung der MTL kann jederzeit vom Boden aus gestoppt werden.
Grundlage für die Implementierung der Komponenten MTH ist der PUS Dienst 11, On-Board Operations Scheduling.

On-Board Control Procedure Handler (OBCPH)

Seit einigen Jahren sind OBCPs Bestandteil der On-board Software. Sie ermöglichen eine hohes Mass an Flexibilität beim Betrieb eines Satelliten ohne die eigentliche On-board Software ändern zu müssen. OBCPs bestehen im Wesentlichen aus Sequenzen von Telekommandos. Die Möglichkeit Variablen zu definieren und einfache Kontrollstrukturen erlauben die Ausführung der Sequenzen in Abhängigkeit von externen Ereignissen. OBCPs werden am Boden erstellt, getestet und in ein Binärformat gewandelt bevor sie an den Satelliten geschickt werden. Der PUS hat dazu einen entsprechenden Dienst (Service 18) definiert. In KARS wurde dagegen der private Dienst 148

implementiert. Dieser findet z.B. in den Sentinel-Satelliten Anwendung. Neben den eigentlichen OBCP-Diensten unterstützt die Komponente noch den Service 1,17 und 128. Eine OBCP ist eine Flight Control Procedure (FCP), die an Bord des Satelliten ohne direkte Interaktion mit einem Operateur ausgeführt wird (siehe auch Abb. 4.12. Im Gegensatz zu MTL Kommandos, wird die Reihenfolge der Bearbeitung auch von dem Inhalt der empfangenen Telemetriepackete beeinflusst. Eine OBCP kann auf die Inhalte des On-board Datenpools zugreifen bzw. Elemente des Datenpools verändern.

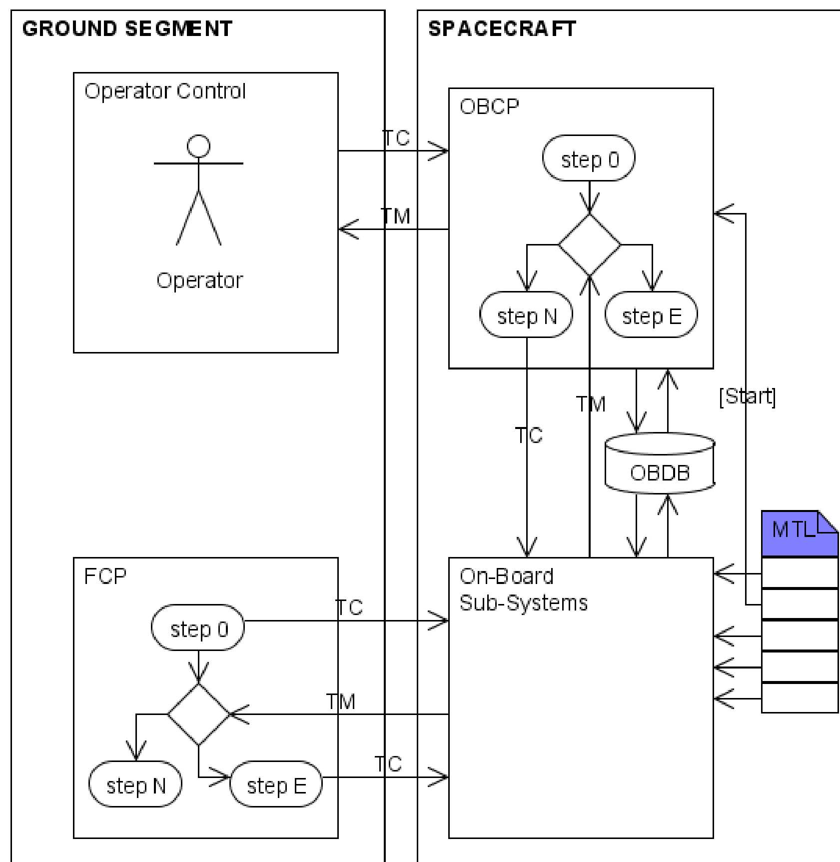


Figure 4.12: OBCP Konzept

Aus der MTL heraus kann eine OBCP gestartet werden. Die Ausführung der OBCP wird vom OBCP- Handler überwacht. Mehrere OBCPs können quasi gleichzeitig abgearbeitet werden.
Der PUS-Dienst 148 ermöglicht es, Prozeduren zu laden, TCs zu den Prozeduren hinzuzufügen bzw. zu löschen, einzelne Prozeduren zu starten und zu stoppen und den Status abzufragen.

Prozess zur Erzeugung einer Control Procedure OBCPs können mit einem ein einfache Editor erstellt werden. Die syntaktische Korrektheit wird mit Hilfe eines Syntaxcheckers überprüft. Die syntaktisch korrekte OBCP wird zunächt in ein Zwischen- format umgesetzt, das in eine Datenbank abgespeichert werden kann. Alternativ zu der Erstellung mittels Texteditor kann eine OBCP auch direkt in dem Zwischenformat erstellt werden. Hierzu werden entsprechende Hilfsmittel bereitgestellt. Parallel zu der Erstellung der OBCP wird ein Test definiert, mit dessen Hilfe die korrekte

Ausführung der OBCP geprüft werden kann. Im Anschluss an diesen Schritt wird die OBCP in ein Binärformat umgesetzt und auf einem Simulator (SVF) ausgeführt.

Mission Planner (MPL)

Der Missionsplaner stellt dem KARS-System spezifische autonome Funktionen zur Verfügung. Seine Hauptaufgabe ist die Bereitstellung eines Planes, also einer Anzahl an Telekommandos, die in einer vorgegebenen Reihenfolge ausgeführt werden. Typischerweise beinhaltet ein KARS-System einen Missionsplaner, es können aber auch mehrere Planer auf verschiedenen Abstraktionsebenen angewandt werden, z.B. auf einer hohen Missions-Aktionsebene, sowie auf einer tieferen Bahnplanungsebene für die autonome Bewegungsgenerierung des Roboters auf dem Satelliten. Ein im KARS-System verfügbare Missionsplaner kann von der Bodenkontrolle durch PUS-Kommandos gesteuert und observiert werden. Die Funktion des Missionsplaners ist es einen optimalen Satz an Aktionen zu generieren, die das zu steuernde System von einem Initialzustand in einen Zielzustand bringen. Die Transition zwischen diesen Zuständen wird Task genannt, z.B. dock satellite. Die Optimalität hängt dabei sehr mit der verwendeten Metrik zusammen. Ein Plan kann z.B. zeitoptimal, verbrauchsoptimal (Treibstoff oder Strom) sein. Es existieren viele verschiedene Planungsverfahren mit ihren jeweiligen Stärken und Schwächen, wobei die Anwendbarkeit sehr von der Problemdomäne abhängig ist. Daher muss jede Mission den für sie passenden Planer auswählen und die entsprechenden Algorithmen in das KARS-System portieren. Der im Rahmen von KARS entwickelte Planer stellt eine standardisierte Schnittstelle sowie eine vereinfachte Implementierung einer Planungsfunktionalität für robotische Demonstrationszwecke bereit.

Supervisor (SUV)

Der SUV in KARS besitzt alle Funktionen, die notwendig sind, um ein komplexes Raumfahrtsystem zu kontrollieren und zu steuern:

- Flexible autonome Systemsteuerung, d.h. Überwachung der zulässigen Mode-Übergänge
- Starten/Stoppen von Komponenten entsprechend des aktuellen Satelliten-Modes
- Direkter Zugriff auf alle Komponenten im System
- Zeit- und Ereignisbasierte automatisierte Kommandierung mit automatischem Einplanen der Kommandoausführung (Zugriff auf die MTH und OBCP-Komponente)
- Komponenten-Überwachung basierend auf Telemetriedaten (Zugriff auf den Software Development Process (SDP))
- Erstellen von Ereignisprotokollen bzw. Log-Daten (Zugriff auf den SDP)
- Zielorientierte autonome Kommandierung des Gesamtsystems (mit *Goals*)
- flexible autonome Ausführung der Goals (mit möglicher Re-Konfiguration)

Durch Implementierung von on-Board Autonomie sollen Möglichkeiten geschaffen werden, die es erlauben für das Gesamtsystem Vorgaben zu machen, die als Goals bezeichnet werden und autonom abgearbeitet werden.
Bei der Ausführung von Goals steuert der Supervisor verschiedene Komponenten, die auf der Platform oder im Instrument liegen können koordiniert an, um ein operationelle Gesamtergebnis zu erzielen. In MARCO V1 waren insbesondere überwachungen auf Komponentenbene und Ausführung von Goals mit aus heutiger Sicht eingeschränkter Leistungsfähigkeit und Benutzerfreundlichkeit implementiert. Zur Implementierung dieser Funktionen wurden in KARS neue und verbesserte Mechanismen genutzt, die eine Steigerung der Nutzbarkeit erlauben. Dies wurde durch eine effizientere Art der Programmierung von Goals.
Zur Planung der Einzelaktionen die für ein Goal notwendig sind, bedient sich der Supervisor des MPL.
Es war nicht Ziel von KARS autonome Szenarien und Planungen an sich zu entwickeln, sondern nur die zugehörigen Mechanismen für eventuelle Pilotanwendungen zur Verfügung zu stellen. Im Rahmen einer Demonstration wurden diese Mechanismen in einer einfachen Anwendung beispielhaft genutzt.

4.3 Software Konfiguration

Der Speicherbedarf und die Anforderungen bzgl. Rechenleistung für eine spezielle Mission werden wesentlich durch die Systemkonfiguration (Anzahl Komponenten, Anzahl der Element im System Datapool, etc.) bestimmt.
Die Schritte zur Konfiguration des Systems sind wie folgt:

1. Identifikation der Komponenten, die auf dem Zielsystem ausgeführt werden sollen
2. Definition der Kummunikations-Infrastruktur zwischen den ausgewählten Komponenten (Anzahl und Puffergrösse)
3. Festlegung des *major timeframe*
4. Unterteilung des major timeframe in Zeitscheiben (genannt Partitionen)
5. Zuteilung von Ressourcen zu Partitionen:
   - Zuweisung von Prozessen zu Partitionen
   - Definition der Speichers, der von einer Partition benutzt werden darf (Tbl: 4.6)
   - Definition der Zeitdauer für die jeweiligen Zeitscheiben (Tbl: 4.7)
6. Definition der Ausfuehrungsreihenfolgen fuer Partitionen (schedule) (Tbl: 4.7). Die untenstehende Tabelle zeigt ein Beispiel für eine typische Platformkontrollanwendung.
7. Auswahl der Zielplattform (Ausführungsplattform, Betriebssystem)
8. Erzeugen des Software-Image fuer das ausgewählte Zielsystem

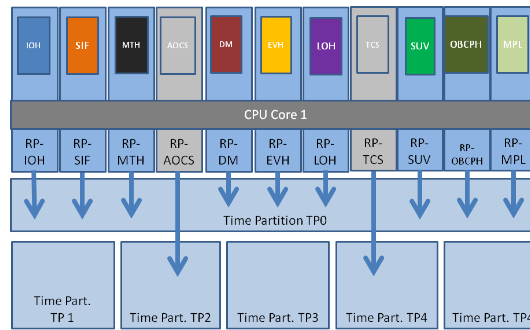


Figure 4.13: Ressource Partitions

Abb. 4.14 zeigt das Scheduling welches fuer die Systemtests verwendet wurde. Der Major Time Frame betraegt 50ms. Die Partitionen 1 bis 5 werden sequentiell ausgefuehrt, waehrend die Partition 0 die Rechenzeit bekommt, die von den anderen Partitionen nicht benötigt wurde. Komponenten, die in genauen Zeitabständen ausgefuehrt werden muessen (z.B. AOCS, und TCS) werden den Partitionen 2 und 4 zugewiesen.
Die KARS-Komponenten der System Support Services sind im Wesentlichen ereignisgesteuert. Daher ist es sinnvoll diese Komponenten der Partition 0 zuzuweisen. Die Komponenten in dieser Partition werden entsprechend ihrer Prioritaet ausgeführt.

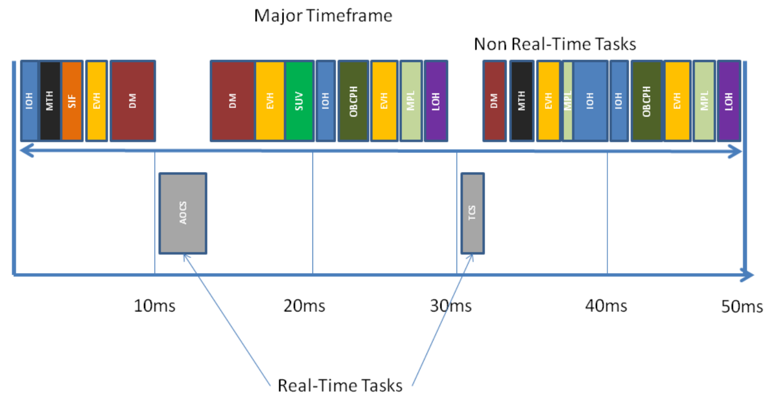


Figure 4.14: Scheduling

Table 4.5: OBC Software Konfiguration

ID	Name	PRID	Prio.
93111330: EVH	EventHandler	3	32
93111310: IOH	IOHandler OBC	1	82
93111360: LOH	LoggingHandler	4	62
93111320: DM	DataManagement	2	62
93111350: MACRO	MACRO Procedures	5	22
93111340: MTH	MissionTimeLineHandler	6	72

continued next page ...

ID	Name	PRID	Prio.
93111370: SUV	Supervisor	9	62
931113B0: MPL	MissionPlanner	10	22
931113E0: OS Kernel	PikeOS	0	100
931113D0: SIF	Service I/F	11	62
931113A0: AOCS	AOCS	7	100
931113C0: TCS	TCS	8	100
93111390: UDP	EQH - UDP	11	62
931113F0: ORBIT	EQH - Orbiter2010	12	62
93111380: MANIP	Manipulator	9	100
93111390: UDP-GND	EQH - UDP	11	62

Table 4.6: Resource Partitions

Resource Part.	Comp.
93111120: RP1_IOH	[IOH]
93111140: RP3_EVH	[EVH]
93111150: RP4_MTH	[MTH]
93111170: RP6_LOH	[LOH]
93111180: RP7_SUV	[SUV]
931111C0: RP11_MPL	[MPL]
931111E0: RP13_SIF	[SIF]
93111130: RP2_DM	[DM]
93111160: RP5_OBCPH	[MACRO]

Table 4.7: Scheduling Schema

Time Slot	Assigned time	Assigned Partition
TP-0	50 ms	RP-SUV, RP-MTH, RP-LOH, RP-IOH, RP-OBCPH, RP-SIF, RP-EVH, RP-MPL, RP-DM
TP-1	10 ms	-
TP-2	10 ms	RP-AOCS
TP-3	10 ms	-
TP-4	5 ms	RP-TCS
TP-5	15 ms	-

4.4 Demonstrator-Architektur

Entsprechend des vorgegebenen TRL von 5 sollte KARS in einer repräsentativen Umgebung demonstriert werden. Dazu war es notwendig eine Demonstrator- und Testinfrastruktur aufzubauen, die wesentliche Elemente eines Satellitensystems enthält:

- Raumsegment
  - Komponente Plattformrechner (repräsentative CPU plus Betriebssystem, KARS inkl. plattformspezifische Demonstratorkomponenten)
  - Komponente Nutzlastrechner (repräsentative CPU plus Betriebssystem, KARS inkl. robotikspezifische Demonstratorkomponenten)
- Bodensegment
  - Nachbildung einer Plattformkontrolle zur Kommandierung des Komponentes Plattformrechner
  - Nachbildung einer Nutzlastkontrolle zur Steuerung des Robotik-Komponentes

Darüber hinaus wurden Simulatoren benötigt, die das Umfeld des Satelliten in geeigneter Weise nachbilden und Visualisierungs-Tools, mit denen die Reaktion des Satelliten/des Komponentes dargestellt werden kann, z.B. in 3D. Abb. 4.15 soll dies veranschaulichen.

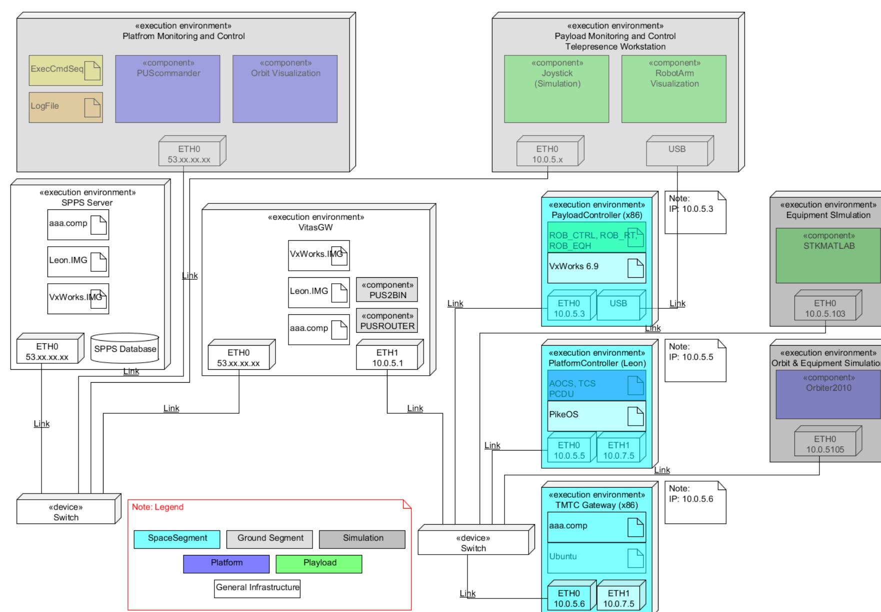


Figure 4.15: Demonstrator Hardware und Software

Die Steuerung und Überwachung des Plattform-Komponentes erfolgte mit Hilfe einer vereinfachten Nachbildung einer Plattformkontroll-Software. Diese erlaubt es vordefinierte Kommandosequenzen zeitgesteuert oder interaktiv auszuführen und die empfangene Telemetrie in geeigneter Form darzustellen. Darüber hinaus stellt diese Komponente Funktionen zur Verfügung, um Fehler in das On-board-System zu injizieren und damit die implementierten FDIR- und Autonomie-Funktionalität testen zu können.

Der Plattform-Simulator simuliert den Umlauf des Satelliten um die Erde, die Lage des Satelliten sowie weitere Umgebungsparameter, sofern sie für den Demonstrator benötigt werden.
Zur Steuerung und Regelung eines realen oder simulierten Manipulators wird ein Robotik-Komponente implementiert. Die Interaktion mit der Lageregelung erfolgt über eine Datenverbindung zum Plattform-Komponente, die bei Bedarf aufgebaut werden kann. Einfaches autonomes Verhalten wird durch die Integration einer Missionsplanung ermöglicht und anhand eines Szenarios beispielhaft demonstriert.
Die Nutzlastkontrolle dient dazu, die Robotik-Komponente des Demonstrators zu konfigurieren und zu bedienen. Eine Visualisierung eines 3D-Modells zeigt zusätzlich die aktuelle Position des Roboters. Wie bei der Missionskontrolle existiert keine direkte physikalische Kommunikationsverbindung zur Robotik-Komponente. Stattdessen werden die Daten wie in einem realistischen Szenario über einen einzigen Link versendet und empfangen.
Der komponenten-basierte Ansatz für die onboard-Module wurde auch für die Module der Testumgebung verwendet. Das Komponenten-Framework stellt den Rahmen für die gesamte Software bereit und sorgt unter anderem dafür, dass die einzelnen Komponenten miteinander interagieren können.
Der Demonstrator wird als Multi-Prozessor-System ausgeführt werden. Plattform-relevante Komponenten wurden auf einem System integriert, das aus einem raumfahrt-qualifizierbaren Prozessor (LEON4) und IMA-fähigen Betriebssystem bestand. Während das Robotik-Komponente auf einem PC mit VxWorks Betriebssystems ausgeführt wurde (s. Abb. 4.16).

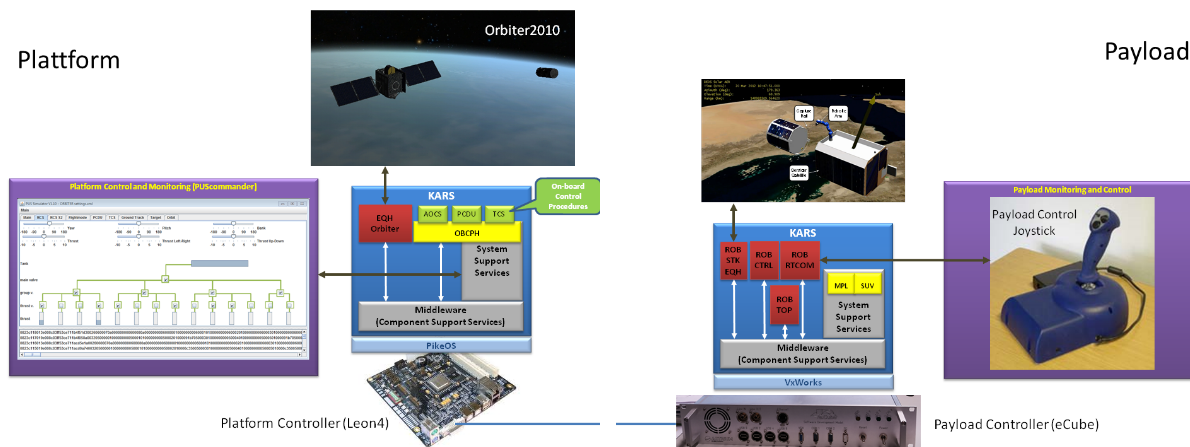


Figure 4.16: Plattform und Nutzlast

4.4.1 Demonstrator-Module

Die Demonstrator-Module dienen zum einen dazu, nachzuweisen, dass sich Anwendungen einfach in das Framework integrieren lassen, die angebotenen Dienste nutzen bzw. ihre Dienste für andere Anwendungen bereitstellen. Zum anderen soll mit diesen Modulen die Systemfunktionalität erweitert werden, um die in der Phase 0+A entwickelten Autonomie-Szenarien testen zu können.

Plattform-Komponente

Das Platform-Komponente bildet die Basis-Plattform-Funktionen nach:

- Power Control and Distribution Unit (Power Control And Distribution Unit (PCDU))
- Thermal Control
- Attitude and Orbit Control (AOCS)

Das PCDU-Komponente enthält ein einfaches Batterie- und Solar-Panel-Model. Über dieses Komponente kann die Spannungsversorgung für die anderen Komponenten und die Heizer eingeschaltet werden. Die Leistungsaufnahme der angeschlossenen Komponenten und Heizer wird über eine mode-abhängige Lookup-Table nachgebildet.
Das TCS umfasst eine konfigurierbare Anzahl von Thermistoren und Heizer. Die Heizleistung der Heizer und die Parameter der Thermistoren sind ebenfalls einstellbar. Die Wechselwirkung zwischen Heizern und Thermistoren kann über eine 2D-Tabelle eingestellt werden. Es können mehrere Regelkreise bestehend aus einem Heizer und mehreren Thermistoren definiert werden. Der Einfluss der Sonne wird über einen orbitabhängigen Temperaturoffset nachgebildet. Je nach ausgewähltem Referenzszenario (on-orbit servicing, lander, rover) müssen unterschiedliche Lageregelungs- bzw. Navigationsmodule entwickelt werden.
Das AOCS inkl. der Satellitenmodi und der dazugehörigen Sensoren und Aktoren wurde an die Implementierung von Sentinel-2 angelehnt.
Die detailierte Beschreibung der einzelnen Modi findet sich in [ASI13a].
Folgende AOCS Modi wurden implementiert:

- Standby (SBY)
- Acquisition Safe Mode (ASM) / Rate Damping (RD)
- Acquisition Safe Mode (ASM) / Deployment (DEP)
- Acquisition Safe Mode (ASM) / Earth Acquisition (EA)
- Acquisition Safe Mode (ASM) / Yaw Acquisition (YA)
- Acquisition Safe Mode (ASM) / Steady State (SS)
- Normal Mode (NOM) / Attitude Hold (AH)
- Normal Mode (NOM) / Fine Pointing (FP)
- Normal Mode (NOM) / Slew (SL)
- Normal Mode (NOM) / Extended Fine Pointing (EFP)
- Normal Mode (NOM) / Back Slew (BSL)
- Orbit Control Mode (OCM) / Slew (SL)
- Orbit Control Mode (OCM) / Stabilization (STAB)
- Orbit Control Mode (OCM) / DV Manoeuvre (DV)

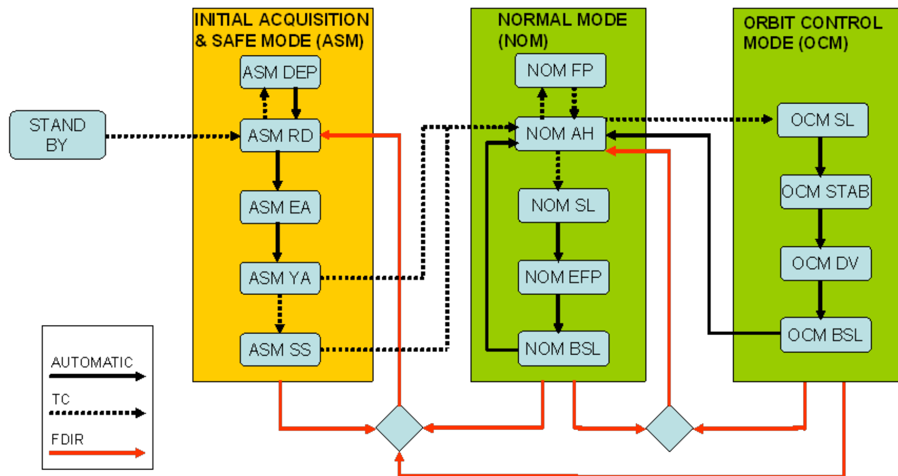


Figure 4.17: AOCS Modeübergänge

In Abb. 4.17 sind die zulässigen Modeübergänge dargestellt:
Folgende AOCS-Sensoren/Aktoren wurden nachgebildet:

- Erd-/Sonnensensor (CESS)
- Startracker (STR)
- Positionsbestimmung (GPS)
- Inertial Measurement Unit (IMU)
- Magnetfeldsensor (MAG)
- Magnettorquer (MTQ)
- Reaction Wheels (RW)
- Reaction Control System (RCS)

Die Detailtiefe mit der die einzelnen Sensoren/Actoren nachgebildet wurden, orientiert sich an dem Modelzweck und variiert zwischen Ein-/Ausschalten und voller Funktionalität.

Automation- und Robotiksoftware

Die Automation- und Robotiksoftware besteht aus drei Hauptkomponenten die eng miteinander zusammenarbeiten. Die Task Orienten Programming (TOP)-Komponente stellt eine lineare Aufteilung von Aufgaben bereit, teilt also z.B. die Aufgabe Finde Wasser in kleinere Operationen auf, die wiederrum in Elementaroperationen (Wechsel der Regelung, Geschwindigkeit, kurze Bewegung von einem Raumpunkt zum anderen) aufgegliedert werden. Elementaroperationen können auch als feingegliederte robotische Aktionen bezeichnet werden. Im Zusammenhang mit dem Planer lässt sich so eine Three-Tier (3T)-Architektur zur Erlangung autonomes autonomen Systems implementieren. Abb. 4.18 stellt die dabei verwendenten Steuerungs- und Regelungsschichten dar.

Die robotische Steuerungskomponente (ROB_CTRL) besteht aus Algorithmen und Interpolatoren die benötigt werden um den robotischen, kraftgeregelten Manipulator zu bewegen. Auf der tiefsten Ebene implementiert die Equipment Handler-Komponente (ROB_EQH) das direkte Interface zum Roboter, stellt also dem KARS-System den notwendigen Treiber zur Verfügung. Für Demonstrationszwecke wurde die Komponente mit einem virtuellen Roboter verbunden, sie stellt außerdem robotische FDIR-Level 1-Funktionen zur Verfügung. Durch die Aufteilung der robitschen Funktionalitäten in ROB_EQH und ROB_CTRL, wird die Steuerungskomponente Hardware-unabhängig. Die implementierten Regelungsalgorithmen können also auch für andere Roboter verwendet werden. Zusätzlich zu den vorgestellten Steuerungs- und Regelungskomponenten, die im Zusammenspiel mit dem Missionsplaner eine autonome Operation des Manipulators erlauben, stellt die Echtzeit-Komponente (ROB_RTCOM) eine Hochgeschwindigkeitsverbindung (1kHz) für den Telepräsenzbetrieb bereit. Die Komponente ist mit einem virtuellen, haptischen Joystick verbunden und leitet ihre Daten ich Echtzeit an die Equipment Handler-Komponente weiter.

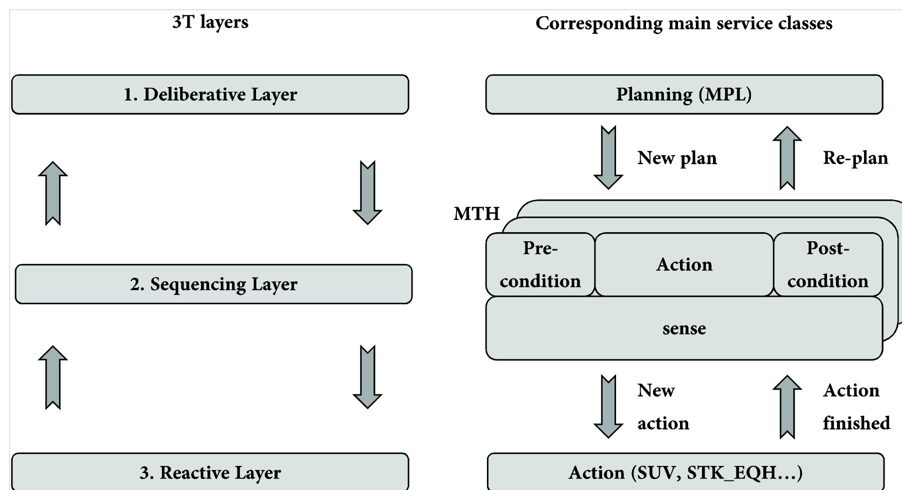


Figure 4.18: OBCP Konzept

4.4.2 Test-/Demonstrationsumgebung

Die Test-/Demonstrationsumgebung stellt Hilfsmittel zur Verfügung, die die Umgebung des autonomen Raumfahrzeugs darstellen und die Reaktion des Gesamtsystems online visualisieren zu können. Zu dieser Umgebung gehören Plattform- und Nutzlast-Simulatoren und eine für den Demonstrator geeignete Nachbildung der Kontrollstationen. Diese Umgebung stellt die Benutzerschnittstelle zu dem *Embedded System* dar. Die Benutzerschnittstelle verwendet State-of-the-Art Möglichkeiten zur Eingabe und Darstellung von Information.

Plattformkontrolle

Zur Interaktion eines Operators mit der KARS-Software wurde eine bereits vorhandene Software (PUScommander, s. Abb. 4.19) zur interaktiven Eingabe von PUS-Kommandos und zur Visualisierung der empfangenen Telemetriedaten verwendet.
Der PUScommander ist in Java implementiert und daher unter den gängigsten Betriebssystemen lauffähig. Das Aussehen der einzelnen Fenster und das Verhalten der Steuerelemente auf den

Fenstern ist weitgehend über eine XML-Datei konfigurierbar.

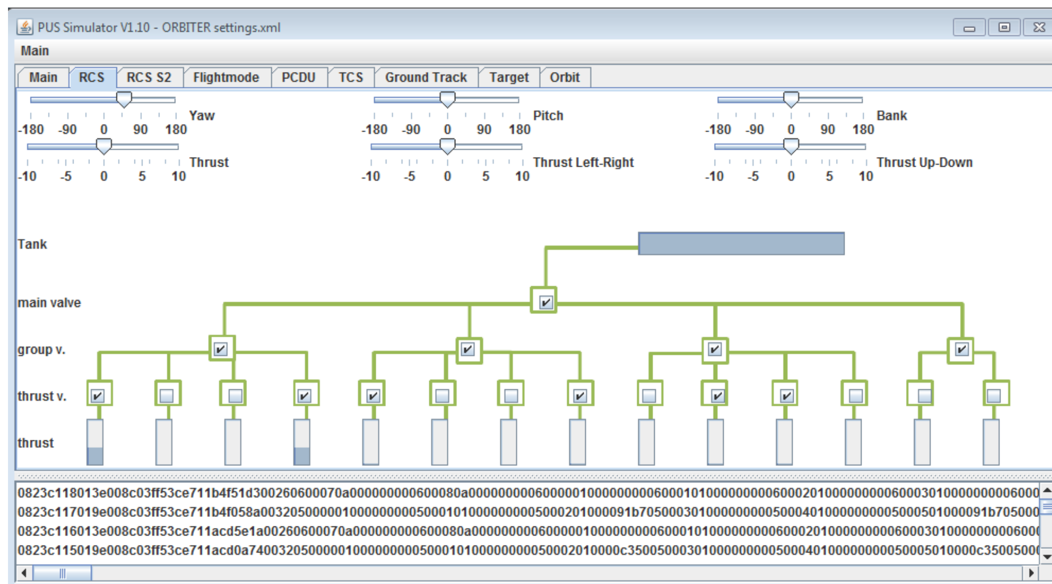


Figure 4.19: PUScommander

Plattformsimulator

Als Plattformsimulator wurde Orbiter2010 (s. http://orbit.medphys.ucl.ac.uk/) ausgewählt, da er eine hochgenaue Orbtberechnung beinhaltet und durch Plug-Ins flexibel erweiterbar ist. Mit Hilfe dieses Plattformsimulator ist es möglich Referenzszenarien aus den verschiedenen Domänen darstellen zu können. Für on-orbit Servicing-Missionen (default) stellt der Plattformsimulator folgende Daten pro simuliertem Satelliten bereit:

- Satellitenposition, -geschwindigkeit und -beschleunigung im ECEF-Koordinatensystem (andere Koordinatensysteme bei Bedarf)
- Satellitenlage
- Zeit
- Sonnenvektor und Flag für die Sichtbarkeit der Sonne
- Erdvektor
- Magnetfeldvektor

Der Plattformsimulator kann mehrere Satellitenorbits gleichzeitig berechnen. Die Funktionalität des Platformsimulator kann über Lua-Skripte erweitert werden. Für KARS wurden z.B. ein einfaches Thermalmodell, verschiedene Equipment-Modelle und eine Socket-Verbindung integriert.
Mit Hilfe der Socket-Verbindung ist es möglich auf die Orbiter-internen Daten zuzugreifen. Über diese Schnittstelle werden Sonsorinformationen in KARSeingelesen bzw. Aktoren (z.B. Thruster, Heizer) aus KARSheraus gesetzt. Die Daten werden auch im Orbiter2010-Fenster dargestellt (s. Abb. 4.20)

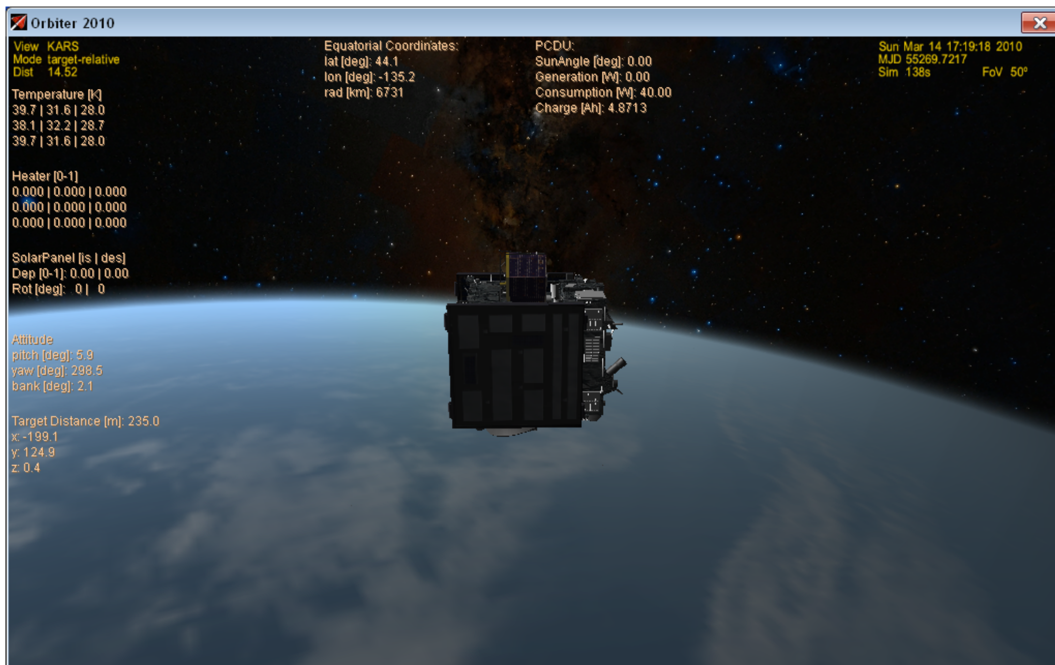


Figure 4.20: Orbiter2010 Screenshot

Nutzlastkontrolle

Zur Interaktion eines Operators mit dem Robotik-Komponente dient eine Nutzlastkontrollstation. Ähnlich wie beim Komponente selbst wird hier kein Wert auf vollen Funktionsumfang gelegt, sondern nur die zur Demonstration nötigen Elemente bereitgestellt. Dazu zählt eine Visualisierung des Roboters wie sie in Abb.

NutzlastSimulator

Der Nutzlastsimulator basiert auf dem Satellitensimulationsprogramm Systems Tool Kit (STK). Das STK Simulationsszenario entspricht dabei der DEOS-Mission und wird für Visualisierungszwecke sowie zur Bereitstellung realisitischer Orbit-, Kommunikations- und Beleuchtungsdaten benutzt.

4.5 Software Evaluation Kit

Im Rahmen von KARS kann ein Evaluation Kit auf der Grundlage eines Raspberry Pi Rechners bereitgestellt werden, um die Basisfunktionalität der Software zu evaluieren.
Der Raspberry Pi Rechner enthält die komplette KARS Software und einige Software-Tools, mit der KARS -Software kommunizieren zu können. Zu diesem Zweck muss der Raspberry Pi mit einem Windows-Rechner verbunden werden: Es werden zwei Verbindungen benötigt:

1. Direkte Netzwerkverbindung
2. USB-Verbindung, um den Raspberry Pi mit Strom zu versorgen

Sobald der Windowsrechner eingeschaltet wird und am USB-Port die Versorgungsspannung anliegt, bootet auch der Raspberry Pi.

Figure 4.21: STK orbit simulation of OOS scenario

Nach dem Abschluss des Boot-Vorgangs hat der Respberry Pi eine feste IP-Adresse: 10.0.5.10, d.h. der Windows-Rechner sollte ebenfalls eine IP-Adresse in diesen Adressraum bekommen, z.B. 10.0.5.100. Nachdem sie den Windows-Rechner entsprechend konfiguriert haben, sollten sie in der Lage sein, den Raspberry Pi im Netzwerk erreichen können.

```
$ ping 10.0.5.10
```

Antwortet der Raspberry Pi Rechner auf das ping-Kommando, können sie sich anschliessend mit dem freigegebenen Laufwerk auf dem Raspberry Pi verbinden. Dort finden sie alle Dokumente und einige Tools, um mit der KARS-Software kommunizieren zu können.
Öffnen sie dazu auf dem Windows-PC den Explorer und klicken sie mit der rechten Maustaste auf das Netzwerksymbol. Es öffnet sich ein Fenster mit verschiedenen Menüpunkten. Wählen sie *Netzlaufwerk verbinden* aus.
Tragen sie in das Fenster die Daten ein, wie sie in Abb. 4.23 dargestellt sind. Bestätigen sie die Eingaben mit [OK]. Anschliessend wird das freigegebene Laufwerk geöffnet.
Auf dem freigegebenen Laufwerk finden sie drei Unterordner

```
[+]-tools\
[+]-docs\
[+]-code\
[+]-tests\
```

Öffnen sie unter Windows eine Kommandozeilen-Shell und wechseln sie auf das freigebene Laufwerk:

```
$ cd U:
```

Wechseln sie in den Ordner *tools/*.

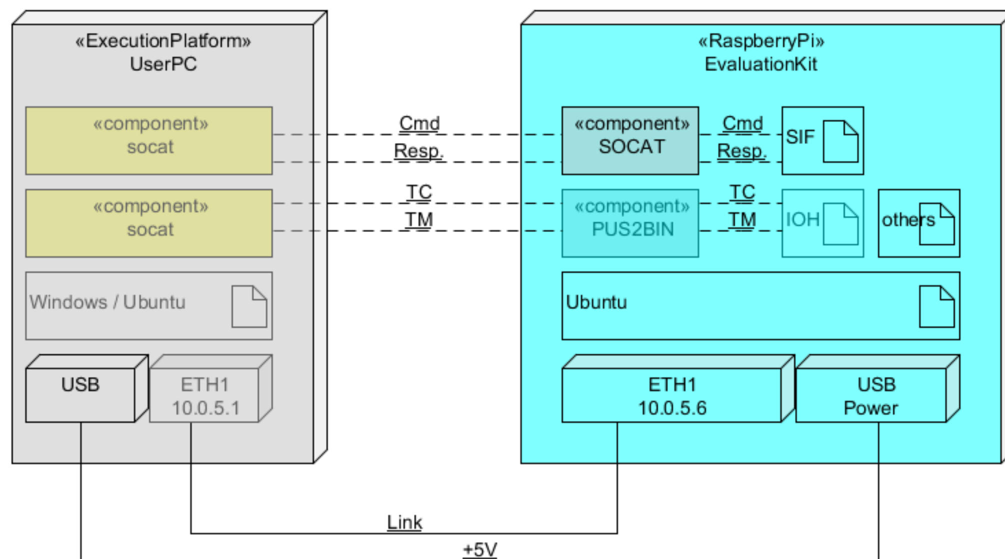


Figure 4.22: KARS Evaluation Kit

```
$ cd tools
```

Öffnen sie eine socat-Session mit folgendem Kommando:

```
$ socat stdin tcp-connect:10.0.5.10:9000
```

Socat wartet auf Konsoleneingaben, um diese dann auf die TCP-Verbindung zu dem angebenen Rechner weiterzuleiten. Auf dem Raspberry Pi nimmt der TCP-Server des Programms PUS2BIN die gesendeten ASCII-Kommandos entgehen, konvertiert diese in binäre UDP-Pakete und schickt sie an den IO-Handler von der KARS-Software weiter.
Die Telemetrie wird vom IO-Handler auf einem anderen Port ausgegeben, daher ist es notwendig, eine zweite socat-Session zu öffnen:

```
$ socat tcp-listen:9001 stdout
```

Wenn sie jetzt das folgende Kommando (PUS service 17) in die erste socat-Session eingeben:

```
TC,17,1,1,12,b0001,0,127
```

antwortet der KARS IO-Handler mit PUS (17,2) auf dem Telemetriekanal:

```
TM,1,1,1,1,b0001,0,127
TM,17,2,1,1,b0001,0,127
```

Die Struktur der Kommandos ist in der KARS PUS-Spezifikation [ASI13d] beschrieben.
Das Service Interface (SIF) der KARS-Software bietet weitere Möglichkeiten der Kommandierung, um z.B. während des Betriebs zusätzliche Information von KARS abzufragen bzw. Werte gezielt zu setzen.
Öffnen sie hierzu eine weitere socat-Session mit folgendem Kommando:

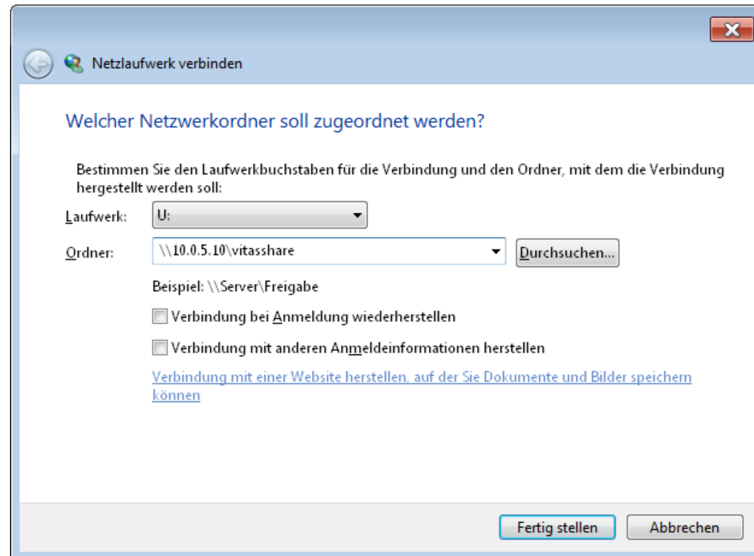


Figure 4.23: Netzlaufwerk auswählen

```
$ socat stdin udp-connect:10.0.5.1:8888
```

Auf diese Weise öffnen sie einen Kanal zur SIF-Komponente. Die Telemetrie wird auf einem anderen Kanal gesendet. Daher müssen sie eine weitere socat-Session öffnen, der die Telemetrie-Daten auf der Konsole ausgibt:

```
$ socat udp-listen:8889 stdout
```

Gegen sie jetzt das folgende Kommando in die erste socat-Session ein:

```
ping;
```

KARS antwortet auf dem Telemetriekanal mit:

```
Alive.
```

Die Liste der verfügbaren ServiceInterface-Kommandos wird nach Eingabe von

```
h   or
help
```

ausgegeben (siehe auch Tabelle 4.8).
Alternativ zu der Kommandierung von KARS aus der Kommandozeile heraus, können sie auch Tools die verwenden, die sich auf dem freigebenen Laufwerk befinden. Diese Toole verbergen die Details der PUS-Kommandos vor dem Anwender. Abb. 4.24 zeigt die Konfiguration des Evaluation Kits mit dem Orbitsimulator und dem PUScommander.
Starten sie den Orbiter2010 und PUScommander durch klicken auf die Batch-Datei

```
startUtilities.bat
```

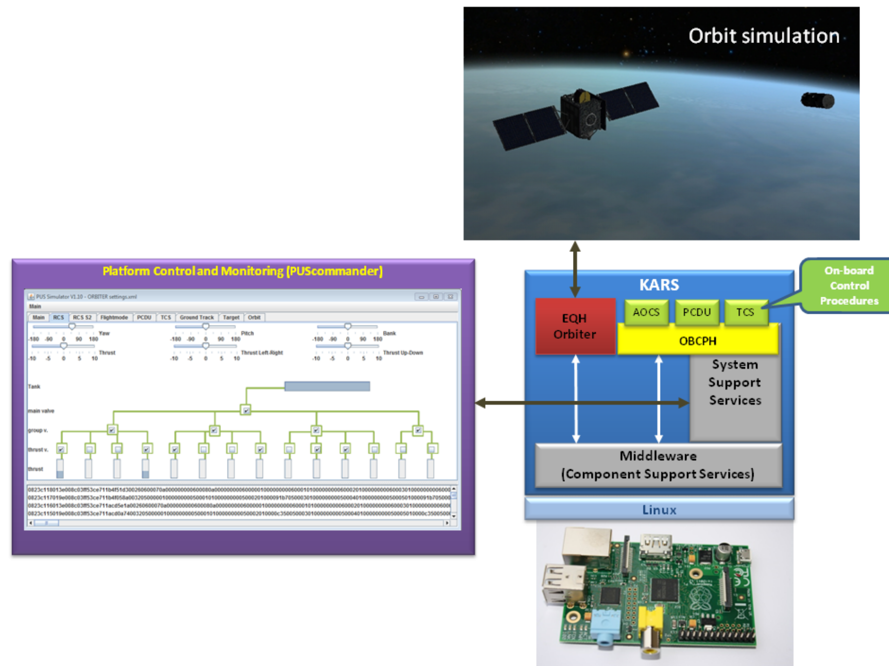


Figure 4.24: KARS Evaluation Kit inkl. Simulatoren

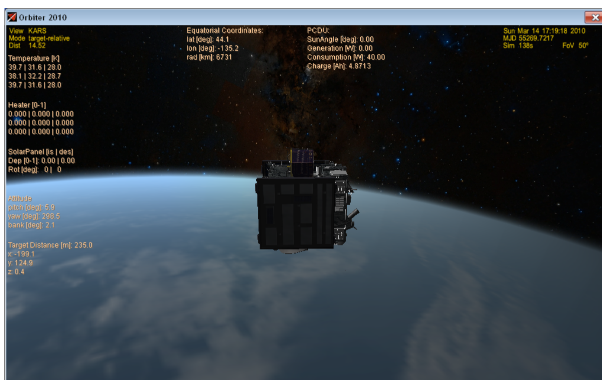


Figure 4.25: Orbiter2010

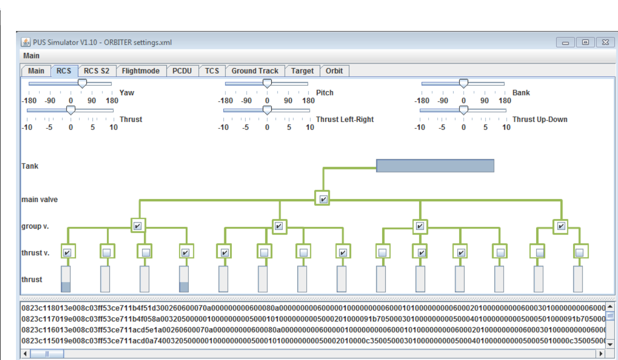


Figure 4.26: PUScommander

Anschliessend sollten die beiden folgenden Fenster (Abb. 4.25, 4.26) auf dem Bildschirm zu sehen sein.
Abb. 4.25 zeigt die Oberfläche der Orbitsimulation mit dem ausgewählten Szenario. Abb. 4.26 zeigt die Oberfläche der simulierten Bodenstation zum Senden von PUS-Kommandos und Empfangen von PUS-Telemetriedaten.
Wechseln sie in der Oberfläche der simulierten Bodenstation in das Tab-Sheet *abc* und setzen sie den Mode auf IAM-SP. Der Satellite sollte sich auf das Ziel ausrichten.

Table 4.8: Service Interface Commands

Service	**Description**
CM(1027,1)	*Service Interface ...*

continued next page ...

Service	Description
CC(1027,4)	*EXIT* exit the connection
CC(1027,5)	*TERMINATE* Terminate all connections and terminate the SIF component
CC(1027,6)	*PING* Check connection
CC(1027,12)	*SET* Set argument of any KARS component
CC(1027,15)	*SETTM* Define group of parameters to be requested by only one group parameter.
CC(1027,16)	*SUBSCRIBE* Subscribe for a periodic delivery of a set of parameters previously defined by a SETTM command.
CC(1027,17)	*UNSUBSCRIBE*
CC(1027,19)	*REQEXEC* Execute a KARS ressource id
CC(1027,20)	*REQREAD* Read a KARS ressource id
CC(1027,21)	*REQWRITE* Write a KARS ressource id
CC(1027,22)	*SETCOMPSTATE* Set the state of any KARS component
CC(1027,23)	*GETTM* Request a set of parameters once
CC(1027,26)	*GET* Get argument of any KARS component
CC(1027,27)	*H* Get argument of any KARS component
CC(1027,28)	*HELP* Get argument of any KARS component
CM(1027,100)	*NAK.* Terminate all connections and terminate the SIF component
CM(1027,101)	*Service Interface. Enter help for help or h for supported commands.* Terminate all connections and terminate the SIF component
CM(1027,103)	*REPLY* Get argument of any KARS component
CM(1027,104)	*Goodbye.* Terminate all connections and terminate the SIF component

continued next page ...

Service	Description
CM(1027,105)	*SIF component is terminating. Goodbye.* Terminate all connections and terminate the SIF component
CM(1027,106)	*Alive.* Terminate all connections and terminate the SIF component
CM(1027,112)	*Set command successful.* Terminate all connections and terminate the SIF component
CM(1027,115)	*Group ID defined.* Terminate all connections and terminate the SIF component
CM(1027,116)	*REPLY* Get argument of any KARS component
CM(1027,117)	*Subscription terminated.* Terminate all connections and terminate the SIF component
CM(1027,122)	*Component state set.* Terminate all connections and terminate the SIF component
CM(1027,123)	*REPLY* Get argument of any KARS component
CM(1027,128)	* Terminate all connections and terminate the SIF component

5 Verwertbarkeit der Ergebnisse und Anschlussfaehigkeit

5.1 Verwertbarkeit

Die KARS-Middleware stellt ein Application Programming Interface (API) bereit, über das der Anwender auf Betriebssystem- und Hardware-Ressourcen zugreifen kann. Dadurch kann sich der Anwender auf die Anwendungsentwicklung konzentrieren und muss sich nicht in betriebssystem-spezifische Aufrufe einarbeiten. Die modulare Struktur der Software ermöglicht es, nur die Dienste zu verwenden, die für die jeweilige Anwendung benötigt werden. Konfigurationstools unterstützen den Anwender bei der Auswahl der benötigten Dienste.
Gegenwärtig hat der Liefergegenstand den TRL 5 erreicht.
Damit bietet der Liefergegenstand eine gute Grundlage für die Verwendung in einer zukünftigen Satellitenmission a ber auch in anderen Bereichen in denen sicherheitskritische Funktionen von Software gesteuert werden. Zu diesen Bereichen zählen u.a. :

- UAV
- Rover
- Simulatoren und Testgeräte für die Luft- und Raumfahrt

5.2 Anschlussfähigkeit

Aus jetziger Sicht ergeben sich zwei Ansatzpunkte für eine Weiterentwicklung der KARS-Software:

- Erhöhung des TRL durch Verwendung in und Anpassung an eine Satellitenmission. Folgende Missionen wären geeignet, den TRL von KARSzu erhöhen
  - OPS-SAT (Experiment-Controller in einem CubeSat)
  - Merlin (Verwendung als Operations Simulator für die ICU)
  - DEOS (Verwendung als Basis-Software für GNC und Robotik)
- Entwicklung einer Tool-Suite zu komfortablen Konfiguration der KARS-Software. Im Rahmen von studentischen Praktika wurden bereits kleine Tools entwickelt, die den Anwender bei der Konfiguration der Software unterstützen, z.B.
  - Definition des Datenpools
  - Definition der Ressource, Event, Error IDs

  - Festlegung welche Komponenten genutzt werden

Mit Hilfe dieser Tool-Suite sollten Anwender in die Lage versetzt werden, alle missionsspezifischen Daten benutzerfreundlich einzugeben und die config-Dateien für KARSund das darunter liegende Betriebssystem zu generieren.

6 Fortschritte bei anderen Stellen

In der Automobil- und in der Luftfahrtindustrie gibt es seit Jahren Bestrebungen der steigenden Komplexität der Software in Fahrzeugen und Flugzeugen gerecht zu werden. Zu nennen sind hier AUTOSAR und IMA.
Während der Laufzeit der Vorhabens sind sowohl in den USA als auch auf europäischer Ebene (ESA) neue Software-Architekturkonzepte für die Raumfahrt mit dem Ziel entwickelt worden, die gestiegenen Anforderungen nach Funktionalität, Modularität, Skalierbarkeit, etc. zu erfüllen und gleichzeitig Aspekte wie funktionale und operationelle Sicherheit (Safety and Security) zu berücksichtigen.
So wird z.B. in einem White Paper des NITRD eine Architektur vorgeschlagen wie sie in Abb. 4.7 dargestellt ist. Kernelemente dieser Architektur sind eine domänen-spezifische *Middleware* und ein Echtzeit-Betriebssystem mit einem *Separation Kernel*.
Die ESA schlägt in dem IMA-SP Programm eine ähnliche Architektur vor (s. Abb. 4.8).
KARS ist weitgehend deckungsgleich mit diesen beiden Entwicklungen und stellt somit eine erste Implementierung dieser Konzepte dar.

7 Erstellte Dokumente und Veroeffentlichungen

Nachfolgend sind alle Dokumente aufgelistet, die im Rahmen des Projektes erstellt wurden (Tbl. refTBL:ErstellteDokumente).
Die Projektergebnisse wurden auf der 2. Nationalen Robotik-Konferenz am 6./7. Maerz in Berlin und auf der IEEE Aerospace Tagung (Maerz 2014) vorgestellt.

7.1 Erstellte Dokumente

Table 7.1: Erstellte Dokumente

ID	Title	Version	Date
KAR_ASI_RP_21024	Progress Report	3.0	05.04.12
KAR_ASI_CID_21025	As-built Configuration List	Draft	31.08.10
KAR_ASI_CID_21026	Configuration Item Data List	2.2	31.01.14
KAR_ASI_CID_21027	Configuration Item List	1.0	31.08.10
KAR_ASI_TN_21028	Software Configuration File	1.7	31.01.14
KAR_ASI_RP_21029	ProductTree	1.4	26.07.13
KAR_ASI_RP_21030	Inventory Record	1.0	07.09.11
KAR_ASD_RP_21032	Zwischenbericht	5	15.02.13
KAR_ASI_RP_21032	Zwischenbericht2–9	2.0	31.05.11
KAR_ASI_TN_21033	Software Release Note	2.2	31.01.14
KAR_ASI_RP_21034	Abschlussbericht	1.0	29.08.14
KAR_ASI_TN_21099	Abkuerzungen	0.9	26.11.13
KAR_ASI_TN_22001	Stand der Technik	1.0	19.01.11
KAR_ASI_TN_22002	KARS Anwendungsbereichs-Analyse und Machbarkeit	1.1	31.05.11
kar_ASI_TN_22003	Technical Budget Report	1.6	15.01.14
KAR_ASI_RP_22004	Verification Control Document	2.5	15.01.14
KAR_ASI_DD_22005	Software Design Document	2.3	30.04.13
KAR_ASI_DD_22005-1	Software Design Document Vol1IOH	2.3	30.04.13
KAR_ASI_DD_22005-2	Software Design Document Vol2DM	2.3	30.04.13
KAR_ASI_DD_22005-3	Software Design Document Vol3MW	2.4	30.04.13
KAR_DLR_DD_22005-4	Software Design Document Vol4SV	2.3	30.04.13
KAR_DLR_DD_22005-5	Software Design Document Vol5MPL	2.3	30.04.13
KAR_ASI_DD_22005-6	Software Design Document Vol6EVH	2.3	30.04.13

continued next page ...

ID	Title	Version	Date
KAR_ASI_DD_22005-7	Software Design Document Vol7MTH	2.3	30.04.13
KAR_ASI_DD_22005-8	Software Design Document Vol8LOH	2.3	30.04.13
KAR_DLR_DD_22005-9	Software Design Document Vol9EQH	2.3	30.04.13
KAR_ASI_DD_22005-A	Software Design Document Vol10OBCPH	2.3	30.04.13
KAR_DLR_DD_22005-B	Software Design Document Vol11ANR	2.3	30.04.13
KAR_ASI_DD_22005-F	Software Design Document Vol15CDS	1.1	15.01.14
KAR_ASI_ICD_22006	Interface Control Document	1.5	07.05.13
KAR_ASI_RS_22007	Software Interface Requirements Document (IRD)	1.0	12.09.11
KAR_ASI_MX_22008	Requirements Traceability matrix (RTM)	1.0 !New	18.03.13
KAR_ASI_PL_22009	Software Verification and Validation Plan	2.3	05.04.12
KAR_ASI_PL_22010	Software Validation Plan (SValP)	1.0	12.09.11
KAR_ASI_PL_22011	Software Unit and Integration Test Plan	1.5	30.04.13
KAR_ASI_PL_22012	SW Development Plan	2.3	05.04.12
KAR_ASI_PL_22013	Software Review Plan (SRevP)	1.0	12.09.11
KAR_ASI_RP_22014	Software Unit and Integration Test Report	1.2	08.02.14
KAR_ASI_RP_22015	Software Verification Report (SVR)	1.2	15.01.14
KAR_ASI_RS_22016	Test Requirement Specification (AIT)	Draft	12.09.11
KAR_ASI_RS_22017	Software System Specification	1.3	05.04.12
KAR_ASI_RS_22018	Software Requirements Specification	1.3	05.04.12
KAR_ASI_TS_22019	Software Validation Specification (SVS)	1.3	28.11.13
KAR_ASI_TN_22020	SpecificationTree	1.3	20.07.12
KAR_ASI_TN_22022	Software User Manual (SUM)	1.3	31.01.14
KAR_ASI_TN_22023	System Concept Report	1.2	05.04.12
KAR_ASI_TN_22024	Autonmy And FDIR	1.1	05.04.12
KAR_ASI_doc_22025	UseCases	1.1	05.04.12
KAR_ASI_RS_22026	KARS Packet Utilization Standard	1.3	06.03.13
KAR_ASI_TN_22027	EvaluationOfCppForOnBoardSWSystems	1.0	05.04.12
KAR_ASI_PL_22028	Prototyping Plan	1.0	05.04.12
KAR_ASI_RP_22029	Prototyping Report	1.2	05.04.12
KAR_ASI_TN_22032	Time And Secure Space Partitioning	1.0	05.04.12
KAR_ASI_TN_22033	Software Reuse File	1.2	05.04.12
KAR_ASI_RP_22034	Software Validation Report	1.2	15.01.14
KAR_ASI_PL_23032	Configuration Management Plan	1.1	05.04.12
KAR_ASI_PL_23033	Risk Management Plan	1.1	05.04.12
KAR_ASI_PL_23034	ReviewProcedure-QR	5.0	11.11.13
KAR_DLR_RP_23035	Review Team Report	2.0	05.04.12

continued next page ...

ID	Title	Version	Date
KAR_ASI_PL_23036	Product Assurance Plan	1.0	05.04.12
KAR_ASI_RP_23037	Product Assurance Report	4.1	13.01.14
KAR_ASI_RP_23038	Risk Assessment Report	Draft	12.09.11
KAR_DLR_RP_23039	Review Authority Report	2.0	05.04.12
KAR_ASI_TN_23039	C-Language Coding Standard	1.0	05.04.12
KAR_ASI_TN_23040	UML Usage Guide	1.0	05.04.12

7.2 Veroeffentlichungen

1. S. Jaekel, M. Stelzer, H.-J. Herpel, Robust and Modular On-Board Architecture for Future Robotic Spacecraft, Aerospace Conference, 2014 IEEE, Big Sky, MT, USA (March 2014).

2. G. Willich, H.-J. Herpel, KARS und KONTIPLAN - Kontroller und Planer fÃ¼r autonome Raumfahrtsysteme, 2. Nationale Konferenz zur Raumfahrt-Robotik, Maerz 2012

References

[ASI11a] ASI. Software Design Document. DD 22005, ASI, December 2011. 2.1.

[ASI11b] ASI. Software System Specification. RS 22017, ASI, August 2011. 1.3.

[ASI11c] ASI. Stand der Technik. TN 22001, ASI, January 2011. 1.0.

[ASI12a] ASI. EvaluationOfCppForOnBoardSWSystems. TN 22027, ASI, March 2012. 1.0.

[ASI12b] ASI. Software Design Document Vol1IOH. DD 22005-1, ASI, December 2012. 2.1.

[ASI12c] ASI. Software Requirements Specification. RS 22018, ASI, February 2012. 1.3.

[ASI12d] ASI. Software Unit and Integration Test Plan. PL 22011, ASI, October 2012. 1.3.

[ASI12e] ASI. Software Validation Report. RP 22034, ASI, October 2012. Draft.

[ASI12f] ASI. Software Validation Specification (SVS). TS 22019, ASI, October 2012. Draft.

[ASI12g] ASI. Software Verification and Validation Plan. PL 22009, ASI, October 2012. 2.3.

[ASI12h] ASI. System Concept Report. TN 22023, ASI, October 2012. 1.2.

[ASI12i] ASI. Verification Control Document. RP 22004, ASI, October 2012. 2.3.

[ASI13a] ASI. Autonmy And FDIR. TN 22024, ASI, October 2013. 1.1.

[ASI13b] ASI. C-Language Coding Standard. TN 23039, ASI, October 2013. 1.0.

[ASI13c] ASI. KARS Anwendungsbereichs-Analyse und Machbarkeit. TN 22002, ASI, October 2013. 1.1.

[ASI13d] ASI. KARS Packet Utilization Standard. RS 22026, ASI, October 2013. 1.3.

[ASI13e] ASI. Software Unit and Integration Test Report. RP 22014, ASI, February 2013. Draft.

[ASI13f] ASI. SW Development Plan. PL 22012, ASI, October 2013. 2.3.

[ASI13g] ASI. UseCases. TN 22025, ASI, October 2013. 1.1.

[ASI14] ASI. Software Design Document Vol15DataSheets. DD 22005-F, ASI, January 2014. 1.2.

Berichtsblatt

1. ISBN oder ISSN	2. Berichtsart Schlussbericht

3. Titel des Berichts

KARS - Entwicklung der Systemsteuerungssoftware: Kontroller für autonome Raumfahrtsysteme

4. Autor(en) [Name(n), Vorname(n)]	
Stelzer, Martin Jäckel, Steffen Herpel, Hans Juergen	5. Abschlussdatum des Vorhabens 10.12.2013 6. Veröffentlichungsdatum 10.08.2014 7. Form der Publikation Bericht
8. Durchführende Institution(en) (Name, Adresse) Airbus DS GmbH Claude-Dornier-Str. 1 D-88090 Immenstaad DLR Institut für Robotik und Mechatronik Münchener Straße 20 82234 Weßling	9. Ber. Nr. Durchführende Institution KAR-SYS-ASI-RP-21032 10. Förderkennzeichen 50 RA 1110 11. Seitenzahl
12. Fördernde Institution (Name, Adresse) Raumfahrtmanagement des Deutschen Zentrums für Luft- und Raumfahrt e.V. Königswinterer Str. 522-524 53227 Bonn	13. Literaturangaben 14. Tabellen 15. Abbildungen

16. Zusätzliche Angaben

17. Vorgelegt bei (Titel, Ort, Datum)

18. Kurzfassung

KARS (Kontroller für autonome Raumfahrtsysteme) ist ein flexibles Rahmenprogramm für Anwendungen in der Raumfahrt.

Die Software stellt Basis-Dienste auf bereit des Packet Utilization Standards (PUS) bereit. Die beinhaltet Empfang und Dekodierung von Telekommandos, Versenden von Telemetriepaketen (IOH), Datenverwaltung und -überwachung (DM), Versenden von "Housekeeping"-Daten, Verwaltung von Ereignissen und der Aktionen, die mit Ereignissen verbunden sind (EVH), Aufzeichnung von Telemetrie-Paketen (LOH),
Verwaltung von externen Geräten (Equipments) (EQH), Verwaltung und Kontrolle der Ausführung von On-board Control Procedures (OBCP) sowie Mission Timelines (MTH).

Der strikte komponenten-basierte Ansatz ermöglicht die einfache Integration von missions-spezifischen Komponenten wie das Attitude and Orbit Control System (AOCS), das Thermal Control System (TCS) oder speziellen "Equipment Handlern".

Ein Operating System Abstraction Layer (OSAL) ermöglicht die schnelle Portierung auf verschiedene Betriebssysteme.
Standardmässig werden Linux, PikeOS and VxWorks unterstützt.
Für zahlreiche Rechner gibt es bereits sog. Board Support Packages (BSP): x86, ARM, Leon3FT and Leon4 CPUs.

Die Entwicklung der Software erfolgte nach den strengen Vorgaben des ESA-Standards ECSS-E40.
Ein entsprechender Satz von Dokumenten wurde erstellt und kann Anwendern auf Anfrage zur Verfügung gestellt werden.

19. Schlagwörter
Autonomie, Avionik, Middleware, Echtzeit, Time and Space Partitioning

20. Verlag	21. Preis

Document Control Sheet

1. ISBN or ISSN

2. type of document (e.g. report, publication)
Final Report

3. title

KARS - Development of System Control Software: Controller for autonomous Spacecrafts

4. author(s) (family name, first name(s))

Stelzer, Martin
Jäckel, Steffen
Herpel, Hans Juergen

5. end of project
Dec., 10th 2013

6. publication date
Aug., 10th 2014

7. form of publication
Report

8. performing organization(s) (name, address)
Airbus DS GmbH
Claude-Dornier-Str. 1
D-88090 Immenstaad

9. originator's report no.
KAR-SYS-ASI-RP-21032

10. reference no.
50 RA 1110

11. no. of pages

12. sponsoring agency (name, address)

Raumfahrtmanagement des
Deutschen Zentrums für Luft- und Raumfahrt e.V.
Königswinterer Str. 522-524
53227 Bonn

13. no. of references

14. no. of tables

15. no. of figures

16. supplementary notes

17. presented at (title, place, date)

18. abstract

KARS (Controller for autonomous spacecraft) is a versatile application framework for on-board software. It provides basic services needed widely used in the space domain. This includes data management and monitoring (DM), housekeeping reporting, event and event-action handling (EVH), logging (LOH), basic equipment handling (EQH), interface to the command and control channel (IOH) based on the Packet Utilization Standard (PUS), execution of on-board control procedures (OBCPH) and mission timelines (MTH). The strict component based approach supports the easy integration of third-party components such as Attitude and Orbit Control System (AOCS), Thermal Control System (TCS), etc.

An Operating System Abstraction Layer allows porting to different operating systems with minimal effort. Currently, Linux, PikeOS and VxWorks are supported as standard. Board support packages are available for Intel, ARM, Leon3FT and Leon4 CPU boards.

The development of the software has been performed according to the strict rules of the ECSS-E40. A full set of documentation is available to support certification activities.

19. keywords
Autonomy, Avionics, Middleware, Real-Time, Time and Space Partitioning

20. publisher

21. price