Carl von Ossietzky Universität Oldenburg
Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung Eingebettete Hardware-/Software-Systeme

OFFIS - Institut für Informatik
FuE-Bereich Verkehr

Projektgruppe Guardian

Projektdokumentation

15. Juni 2018

Projektmitglieder:
- Alexander van Düllen
- Paul Hannibal
- Nils Heinig
- Lennart Hoffhues
- Tino Hoffmann
- Marten Horstmann
- Mark Kettner
- Oliver Klemp
- Sven Lampe
- Marvin Menke
- Patrick Uven

Projektbetreuung:
- Prof. Dr.-Ing. Wolfgang Nebel
- Dr. Domenik Helms
- Nils Koppaetzky
- Lars Kosmann
- Malte Metzdorf
- Henning Schlender
- Ralf Stemmer

Zusammenfassung

Das Ziel der Projektgruppe Guardian (PG Guardian) war es, eine Satellitensteuerung für einen simulierten Satelliten zu entwerfen, die mit Hilfe einer Fehlerinjektionskomponente getestet werden kann. Als Simulationsumgebung dient das Spiel Kerbal Space Program (KSP), in welchem eine Mission mit Hilfe des Satelliten durchgeführt wird. Bei der Mission geht es um das Verschieben eines Satelliten aus seiner Umlaufbahn, um den Erdähnlichen Planeten Kerbin zu schützen. Die Satellitensteuerung wird auf vier Xilinx Zynq-7000 (ZedBoard) betrieben, die Fehlerinjektion auf einem PC mit Fehlerinjektionskomponenten im Zielsystem.

Um diese Aufgabe umzusetzen, hat sich die Projektgruppe in drei Teilprojekte gegliedert: *Simulation*, *Satellitensteuerung* und *Fehlerinjektion*. Die Aufgabe der Simulation war es, die Schnittstelle zur Simulationsumgebung und ggf. weitere notwendige Komponenten bereitzustellen. Das Teilprojekt Satellitensteuerung entwarf die Satellitensteuerung, welche unter anderem Komponenten zur Missionskontrolle, zum Ressourcen-Management, zur Orbitplanung und zur Triebwerkssteuerung umfasst. Das Teilprojekt Fehlerinjektion hatte die Aufgabe, eine Benutzerschnittstelle zur Verfügung zu stellen, um Fehler injizieren zu können. Auf diese Weise kann die Satellitensteuerung um Adaptionsverfahren erweitert und getestet werden. Die Projektgruppe nutzte das Verfahren des System Level Design (SLD)s, bei dem aus der Spezifikation ein Systementwurf entsteht und dieser im Zuge von Refinement-Prozessen zur entgültigen Systemarchitektur führt.

Als Ergebnis der einjährigen Projektgruppe konnte ein Demonstrator vorgelegt werden, der die Mission erfolgreich absolviert. Außerdem ist diese Dokumentation entstanden.

Inhaltsverzeichnis

Teil I.

Projektinitialisierung

1. Einleitung

Im Zuge der Masterstudiengänge *Informatik*, *Wirtschaftsinformatik* und *Eingebettete Systeme und Mikrorobotik* an der Carl von Ossietzky Universität Oldenburg werden einjährige Projektgruppen durchgeführt. Die Projektgruppen setzen sich mit der Umsetzung einer Aufgabenstellung aus dem Bereich der Informatik auseinander. Neben der Entwicklung einer Hard- oder Softwarelösung werden Kompetenzen im Bereich des Projektmanagements, des Teamworks und der Problemlösung gefordert und gefördert.

Für eingebettete Systeme gelten in vielen Einsatzgebieten besondere Umstände, wie begrenzte Energiemengen und schlechte Wartbarkeit. Insbesondere im Weltraum unterliegen eingebettete Systeme starken äußeren Einflüssen, die gepaart mit den geringen Wartungsmöglichkeiten die Entwickler solcher Systeme vor große Herausforderungen stellen. Deswegen müssen sie eine hohe Fehlertoleranz, geringen Energieverbrauch und die Fähigkeit zur Adaption an die Umgebungseinflüsse aufweisen.

1.1. Projektidee

Die Aufgabe der PG Guardian unterteilt sich in drei Aufgabenbereiche. Die erste Teilaufgabe ist die Entwicklung und Implementierung einer Satellitensteuerung auf einem oder mehreren ZedBoard. Dabei gilt es, besonderes Augenmerk auf die Toleranz dieser Steuerung auf mögliche Fehler durch, zum Beispiel, äußere Einflüsse und Alterung zu legen. Die zweite Teilaufgabe besteht darin, mit Hilfe der Weltraumsimulation KSP die entwickelte Satellitensteuerung in realistischen Szenarien zu testen. Die dritte Teilaufgabe umfasst die Entwicklung einer Fehlerinjektion, die es ermöglicht Fehlerfälle des Satelliten und die Verhaltensweisen der Satellitensteuerung zu testen.

1.2. Projektziele

Die definierten Ziele der PG Guardian lassen sich in drei Kategorien unterteilen:

Terminziel Ein selbst gestellter oder von außen auferlegter Termin zu dem eine bestimmte Aufgabe abgeschlossen sein muss.

Qualitätsziel Ein Ziel bezüglich der Güte des herzustellenden Produkts.

Kostenziel Ein Ziel bezüglich der Budgetnutzung, dass der Projektgruppe bereitgestellt wurde.

Im Folgenden werden die Ziele der Projektgruppe tabellarisch zusammen mit ihrer zugemessenen Priorität vorgestellt. Dabei stellt die Priorität 1 die höchste Priorität und die Priorität 5 die niedrigste dar. Zusätzlich wurden alle Ziele, die in jedem Fall umgesetzt werden müssen, mit einer Kennzeichnung im Muss-Feld angegeben. Diese Ziele werden bei Änderungen im Projekt nicht angetastet.

Tabelle 1.1.: Terminziele

Bez.	Ziel	Priorität					
		1	2	3	4	5	Muss
T001	Das Proposal muss am 14.12.2016 abgegeben werden.	•					•
T002	Die Inhalte des Proposals werden am 01.12.2016 in einer Präsentation vorgestellt.	•					•
T003	Die Projektorganisation muss am 01.12.2016 feststehen.	•					•
T004	Ein erster Prototyp der Satellitensteuerung ohne Digilent ZedBoard muss zum 31.03.2017 funktionsfähig sein.	•					•
T005	Ein Zwischenbericht zum aktuellen Stand des Projektes muss zum 31.03.2017 übergeben werden.	•					•
T006	Der Zwischenbericht wird am 04.04.2017 in einer Präsentation vorgestellt.	•					•
T007	Die Präsentation des ersten Prototypen der Satellitensteuerung ohne Digilent ZedBoard erfolgt am 04.04.2017.	•					•
T008	Ein zweiter Prototyp der Satellitensteuerung mit Digilent ZedBoard Unterstützung muss bis zum 06.06.2017 funktionsfähig sein.	•					•
T009	Die Präsentation des zweiten Prototypen der Satellitensteuerung mit Digilent ZedBoard Unterstützung erfolgt am 06.06.2017.	•					•
T0010	Der Abschlussbericht muss bis zum 30.09.2017 übergeben werden.	•					•
T0011	Ein funktionsfähiger Demonstrator muss bis zum 30.09.2017 übergeben werden.	•					•
T0012	Die Abschlusspräsentation des Projektes erfolgt am 04.10.2017.	•					•

Tabelle 1.2.: Qualitätsziele

Bez.	**Ziel**	**Priorität**					
		1	2	3	4	5	Muss
T0013	Die Systemkomponenten müssen modular erweiterbar sein.				•		
T0014	Für die Kommunikation zwischen Digilent ZedBoard und Host-PC muss eine Ethernet Schnittstelle verwendet werden.	•					•
T0015	Das Projekt muss fertiggestellt werden. Das bedeutet, dass alle Anwendungsfälle und Anforderungen erfüllt werden sollen, sofern sich nicht im Laufe des Projektes herausstellt, dass diese unter den gegebenen Bedingungen technisch nicht umsetzbar sind.		•				•
Unterziele Fehlerinjektion							
T0016	Es müssen Umwelt- und Systemfehler in das Satellitensystem injiziert werden können.	•					•
T0017	Es soll ein Katalog für mögliche Fehlerinjektionen in die Satellitensteuerung und in die Kommunikation zwischen ZedBoard und Simulator entwickelt werden.			•			
T0018	Die Fehlerinjektionskomponente muss auf die Kommunikation zwischen KSP und dem Satelliten zugreifen können.	•					•
T0019	Die Maßnahmen des Satelliten sollen dem Fehlerinjektionssystem kommuniziert werden.				•		
T0020	Alle injizierten Fehler sollen in der Fehlerinjektionskomponente angezeigt werden.		•				
T0021	Es müssen Systemfehler in das Satellitensystem auf dem ZedBoard injiziert werden können.	•					•
T0022	Der Anwender des Systems muss durch die Fehlerinjektionskomponente Fehler injizieren können.	•					•
Unterziele Satellitensteuerung							
T0023	Der Satellit kann sich autonom einem festgelegten Ziel nähern.	•					
T0024	Der Satellit kann jederzeit durch eine Bodenstation gesteuert werden.				•		

Tabelle 1.2.: Qualitätsziele

Bez.	Ziel	Priorität					
		1	2	3	4	5	Muss
T0025	Der Satellit kann reale Satellitenkomponenten wie ein Solarpanel beinhalten.				•		
T0026	Es muss ein Energiemanagement für den Satelliten erstellt werden.		•				
T0027	Die Konfiguration der Satellitensteuerung muss entsprechend den Anforderungen der Mission erfolgen.	•					
T0028	Fehlertoleranz soll für den Satelliten verwendet werden.			•			
T0029	Der Satellit kann Adaptionsverfahren beherrschen.					•	
Unterziele Simulator-Plugin							
T0030	Es muss eine Mission für den Satelliten definiert werden.	•					•
T0031	Es muss ein Protokoll zur Kommunikation mit dem ZedBoard und der Fehlerinjektion erstellt werden.	•					•
T0032	Es muss eine Mod ausgewählt werden, die eine Anbindung von einem ZedBoard ermöglicht.	•					•
T0033	Es muss eine Schnittstelle erstellt werden, die alle benötigten Informationen aus dem Simulator zur Steuerung des Satelliten bereitstellen kann.	•					•

Tabelle 1.3.: Kostenziele

Bez.	Ziel	Priorität					
		1	2	3	4	5	Muss
T0034	Das Budget von 1.000€ darf nicht überschritten werden.		•				

1.3. Stand der Technik

Um die Projektgruppe in die Forschungslandschaft einzuordnen, wird in diesem Abschnit ein Projekt der Universität Stuttgart vorgestellt, welches einen ähnlichen Fokus wie diese Projektgruppe hat.

Seit 2004 wird an einem Projekt namens „Flying-Laptop“ [76] gearbeitet. Hierbei handelt es sich um die Entwicklung eines Kleinsatelliten zur Erdbeobachtung. Der Satellit ist beinahe fertiggestellt und der Start des Satelliten ist für Anfang 2017 in Indien geplant. Die Gesamtkosten des Projekts belaufen sich auf einen einstelligen Millionenbetrag.

Ein Teil des Projekts besteht daraus, einen kompletten On-Board-Computer, auf Basis eines Field Programmable Gate Array (FPGA), für den Satelliten zu entwickeln. Die Entwicklung des Teilsystems wurde 2013 fertiggestellt. Zum Verifizieren der entwickelten Komponenten wird eine Simulationsumgebung namens Model-based Development & Verification Environment (MDVE) der Firma Astrium GmbH (seit 2014 zu Airbus DS zugehörig) verwendet. Bei MDVE handelt es sich um einen Echtzeitsimulator, der ebenfalls zum Debuggen verlangsamt werden kann. MDVE kann die Weltraumumgebung und Flugdynamik simulieren und beinhaltet thermische und elektrische Verbrauchsmodelle. Des Weiteren kann die Umgebung alle gängigen Satellitenkomponenten simulieren, sodass nur bestimmte Teile des Systems jeweils selbst umgesetzt werden müssen. In der Simulation sind mehrere Übertragungsprotokolle und Antennenmodelle enthalten, wodurch zur Kontrolle der Simulation bereits vorhandene Software zur Missionskontrolle von echten Satelliten verwendet werden kann. Zur Visualisierung der Flugdaten wird das kostenlose 3D-Astronomieprogramm Celestia verwendet.

Zu diesem Projekt wurden unter anderem mehrere Dissertationen geschrieben und veröffentlicht. Hierzu gehören die Implementierungen verschiedener Attitude Control Subsystem (ACS) Algorithmen und die Anbindung verschiedener Antennentechnologien auf Basis eines FPGAs.

Im Vergleich zu dieser Projektgruppe handelt es sich um ein Projekt mit einer viel längeren Laufzeit und einem viel größeren Budget. Die Universität Stuttgart hat einen kompletten Satelliten von der Konzeption bis zum Start des Satelliten entwickelt. Diese Projektgruppe beschäftigt sich lediglich mit der Entwicklung eines Satellitensteuersystems mit Hilfe eines Simulators und hat somit Ähnlichkeiten zum Projekt der Universität Stuttgart.

2. Grundlagen

In diesem Kapitel werden die Grundlagen für die PG Guardian vermittelt. Da an einer Satellitensteuerung gearbeitet wird, müssen zunächst grundlegende Kenntnisse aus der Astrophysik und die Beschaffenheit von Satelliten aufbereitet werden. Anschließend wird die Simulationsumgebung vorgestellt, gefolgt von einer Einführung in die Fehlerinjektion und Fehlertoleranz. Abschließend werden die C++-Bibliothek SystemC, der Begriff der Co-Simulation eingeführt und die zur Verfügung gestellte Hardware vorgestellt.

2.1. Astrophysik

In diesem Abschnitt werden die Grundlagen für die Berechnungen von Flugkörpern im All vorgestellt. Dies ist im Hinblick auf das Ziel, eine Satellitensteuerung zu entwerfen, von besonderer Relevanz. Flugbahnen eines Körpers können alle Formen eines Kegelschnittes annehmen.

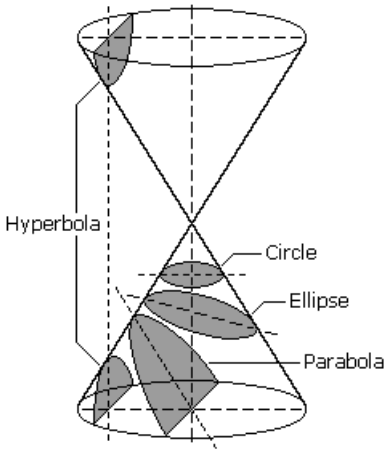


Abbildung 2.1.: Mögliche Flugbahnen im Kegelschnitt. Quelle: [12]

Hierzu gehören Kreise, Ellipsen, Parabeln oder Hyperbeln. Abbildung 2.1 zeigt alle möglichen Laufbahnen. Damit das Objekt auf einer Parabel oder Hyperbel fliegt, muss eine ausreichende Geschwindigkeit erreicht werden, um die Gravitation des Körpers verlassen zu können. Auf einer ellipsenförmigen Flugbahn wird der Punkt mit dem größten Abstand zum anziehenden Körper Apoapsis und der Punkt mit der niedrigsten Entfernung Periapsis genannt. Um einen Orbit vollständig zu beschreiben, werden insgesamt sechs mathematische Eigenschaften des Orbits benötigt, die auch Bahnelemente genannt werden. Abbildung 2.2 zeigt die Bahnelemente anhand eines Beispielorbits. Das erste Bahnelement ist die große Halbachse a, hierfür wird der Abstand zwischen der Periapsis und der Apoapsis bestimmt und anschließend halbiert.

$$a = \frac{r_{min} + r_{max}}{2} \tag{2.1}$$

Das nächste Elemente ist die Exzentrizität e des Orbits. Die Exzentrizität ist ein Maß für den Unterschied zwischen dem Punkt mit dem maximalen und minimalen Abstand zum Planeten.

Anhand des minimalen Abstands r_{min} und des maximalen Abstands r_{max} kann die Exzentrizität wie in Gleichung 2.2 berechnet werden.

$$e = \frac{r_{max} - r_{min}}{r_{max} + r_{min}} \tag{2.2}$$

Die Exzentrizität liegt bei kreisförmigen Laufbahnen bei 0, bei ellipsenförmigen zwischen 0 und 1, bei parabelförmigen bei 1 und bei hyperbolischen über 1.

Jeder Himmelskörper besitzt eine Referenzebene. Bei der Erde handelt es sich dabei um die Ebene, die durch den Äquator verläuft. Die Inklination i und der Winkel Ω des aufsteigenden Knotens geben die Rotation der Orbitebene im Vergleich zu der Referenzebene des Planeten an. Unter der Inklination i wird der Winkel zwischen der Referenzebene und der Orbitebene verstanden. Der Winkel des aufsteigenden Knotens Ω beschreibt die Lage der Orbitbahn im Vergleich zu der Referenzebene. Ein weiteres Bahnelement ist das Argument der Periapsis ω, das den Winkel zwischen dem aufsteigenden Knoten und der Periapsis des Orbits angibt und somit die Rotation des Orbits um die Achse angibt, die orthogonal zur Orbitebene steht. Als letztes Element, das zur vollständigen Beschreibung eines Orbits benötigt wird, ist ein Zeitpunkt T_p, zu der das Objekt die Periapsis passiert hat. Anhand dieser Daten kann nun die Position des Objektes zu jedem Zeitpunkt bestimmt werden [12].

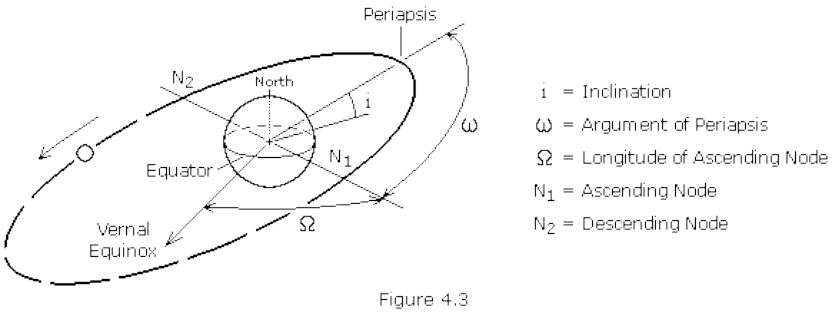


Abbildung 2.2.: Bahnelemente eines Orbits. Quelle: [12]

Die Geschwindigkeit eines Objektes auf einem kreisförmigen Orbit wird durch Gleichung 2.3 gegeben. G bezeichnet die Gravitionskonstante, M die Masse des Himmelskörpers und r den Radius des Orbits. Über Gleichung 2.4 erhält man die Geschwindigkeit in Abhängigkeit von dem aktuellen Abstand r des Objektes zum Himmelskörper.

$$v = \sqrt{\frac{G \cdot M}{r}} \tag{2.3}$$

$$v = \sqrt{G \cdot M \cdot (\frac{2}{r} - \frac{1}{a})} \tag{2.4}$$

Der Drehimpuls h eines Orbits ist eine Erhaltungsgröße und lässt sich anhand des Geschwindigkeitsvektors $\vec{v}$ und dem Richtungsvektor $\vec{r}$ berechnen.

$$h = \vec{r} \times \vec{v} \tag{2.5}$$

Somit ändert sich die Durchschnittsgeschwindigkeit eines Orbits nicht ohne äußere Einwirkung [16].

2.1.1. Orbitanpassungen

Für die Geschwindigkeitsänderung eines Objektes in einem Orbit gelten folgende Faustregeln [36]:

- In einem idealen System ändert sich die Umlaufzeit und die Periode des Orbits nicht, ohne dass eine Kraft auf das Objekt im Orbit angewandt wird.
- Je größer der Radius des Orbits ist, desto langsamer wird die Bewegung des Objektes im Orbit im Vergleich zur Oberfläche des Satelliten.
- Wenn an einem einzigen Punkt die Geschwindigkeit geändert wird, verläuft die Orbitbahn immernoch durch den Punkt an dem die Geschwindigkeit geändert wurde, allerdings ändert sich der Verlauf der restlichen Bahn.
- Wird auf einer Kreisbahn die Geschwindigkeit an einem Punkt reduziert, wird die Bahn zu einer Ellipse mit der Apoapsis an der Stelle an der die Geschwindigkeit geändert wurde.
- Wird auf einer elliptischen Bahn die Geschwindigkeit an der Apoapsis erhöht, wird der Abstand der Periapsis größer. Mit ausreichender Erhöhung der Geschwindigkeit kann aus der elliptischen Laufbahn eine kreisförmige werden. Umgekehrt kann genauso durch eine Verringerung der Geschwindigkeit an der Periapsis der Abstand der Apoapsis verringert werden und somit die elliptische in eine kreisförmige Bahn umgewandelt werden.

Hohmann-Transfer

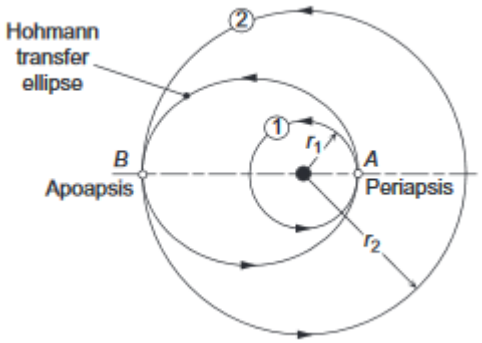


Abbildung 2.3.: Beispielhafter Hohmann-Transfer zwischen den beiden zirkulären Orbits 1 und 2. Quelle: [16]

Beim Hohmann-Transfer handelt es sich um das energieeffizienteste Verfahren, um zwischen zwei kreisförmigen Orbits mit der selben Ebenenneigung zu wechseln. Hierfür wird zuerst an einem beliebigen Punkt der Kreisbahn die Geschwindigkeit erhöht. Durch die Erhöhung der

Geschwindigkeit wird aus der kreisförmigen Umlaufbahn eine ellipsenförmige. Die Geschwindigkeitserhöhung muss so gewählt werden, dass der Abstand der entstehenden Apoapsis zum Himmelskörper dem gewollten Radius entspricht. Anschließend wird solange gewartet, bis der Satellit die Apoapsis erreicht hat. Hier werden nun erneut die Triebwerke, mit einer Geschwindigkeit die den Abstand der Periapsis exakt auf den gewollten Orbitradius bringt, gezündet. Nun fliegt der Satellit auf einem Orbit mit dem gewollten Radius. Dieses Verfahren kann ebenfalls benutzt werden um die große Halbachse eines elliptischen Orbits bei gleich bleibender Exzentrizität zu erhöhen oder um von einer elliptischen Bahn auf eine kreisförmige zu wechseln. Hierbei muss beachtet werden, dass der erste Geschwindigkeitsschub auf der Periapsis geschehen muss. Abbildung 2.3 zeigt einen beispielhaften Transfer zwischen zwei zirkulären Orbits. Die benötigten Geschwindigkeitsschübe lassen sich über den Drehimpuls der Übergangsorbits bestimmen. Gleichungen 2.6 und 2.7 geben an, wie der benötigte Drehimpuls für einen elliptischen beziehungsweise kreisförmigen Orbit berechnet werden kann. Für einen elliptischen Orbit wird $\mu = G \cdot M$, der Abstand der Apoapsis r_{max} sowie der Abstand der Periapsis r_{min} benötigt. Für einen kreisförmigen Orbit wird lediglich μ und der Radius r des Orbits benötigt.

$$h = \sqrt{2\mu} \cdot \sqrt{\frac{r_{min} \cdot r_{max}}{r_{min} + r_{max}}} \tag{2.6}$$

$$h = \sqrt{\mu \cdot r} \tag{2.7}$$

Anhand der vorherigen Gleichungen kann das aktuelle Drehmoment und das gewollte Drehmoment berechnet werden. Da an der Apoapsis und der Periapsis der Geschwindigkeitsvektor exakt orthogonal zum Richtungsvektor verläuft, kann über Gleichung 2.8 die aktuelle Geschwindigkeit und die benötigte Geschwindigkeit und über die Differenz der benötigte Geschwindigkeitsschub berechnet werden.

$$v = \frac{h}{r} \tag{2.8}$$

Eine Spezialform des Hohmann-Transfers ist der bi-elliptische Hohmann-Transfer. Hierfür wird bei der ersten Triebwerkzündung die Geschwindigkeitsänderung so gewählt, dass eine elliptische Bahn mit einem Apoapsisabstand entsteht, der höher als der gewollte Bahnradius ist. Hierdurch muss nun an der Apoapsis weniger Schub gegeben werden, um die Periapsis auf den Radius zu erweitern. Anschließend wird auf der Periapsis derart gebremst, dass der Radius der Apoapsis auf den Bahnradius verringert wird. Bei einem Transfer von einem kreisförmigen Orbit zu einem anderen, lässt sich zeigen, dass der bi-elliptische Transfer genau dann effektiver ist, wenn der Quotient aus dem neuen Bahnradius und dem alten Bahnradius größer als 11.94 ist [16].

Phasenverschiebung des Orbits

Bei einer Phasenverschiebung wird versucht, den Zeitpunkt, an dem ein Satellit einen bestimmten Punkt in einem Orbit passiert, anzupassen. Da jede Geschwindigkeitsänderung zu einer Änderung der Orbitbahn führt, kann eine Geschwindigkeitserhöhung nicht direkt benutzt werden, um die Phasenverschiebung zu erreichen. Für die Phasenverschiebung muss ein Hohmann-Transfer in zwei Phasen ausgeführt werden, um den gewünschten Effekt zu errei-

chen. Zuerst wird auf der Periapsis die Geschwindigkeit derart erhöht oder erniedrigt, dass der Unterschied zwischen der Umlaufzeit des Verschiebungsorbits und der Umlaufzeit des eigentlichen Orbits genau der gewollten Phasenverschiebung entspricht. Anschließend wird ein voller Umlauf des Orbits abgewartet. Ist der Satellit nun erneut an der Periapsis angekommen, dann wird die im ersten Schritt durchgeführte Geschwindigkeitsänderung genau umgekehrt ausgeführt, um den Satelliten wieder auf den ursprünglichen Orbit zurückzubringen. Anhand der Periode T des ursprünglichen Orbits und der gewollten Phasenverschiebung kann die benötigte Periode T_{neu} für den Anpassungsorbit berechnet werden. Mithilfe von T_{neu} und dem Standard Gravitationsparameter $\mu = G \cdot M$ kann nun die benötigte Länge der großen Halbachse des neuen Orbits berechnet werden und anhand dieser Länge die benötigten Geschwindigkeitsänderungen berechnet werden. [16].

$$a = (\frac{T \cdot \sqrt{\mu}}{2 \cdot \pi})^{\frac{2}{3}} \tag{2.9}$$

Lageänderung der Apoapsis und Periapsis

Um das Argument der Periapsis ω zu verändern und somit die Lage der Apoapsis und Periapsis zu rotieren, kann ebenfalls ein mehrstufiger Hohmann-Transfer verwendet werden. Hierfür wird das Raumfahrzeug an der Periapsis derart abgebremst, dass es in einen kreisförmigen Orbit mit dem Radius der Periapsis gelangt. Anschließend wird solange gewartet, bis das Raumfahrzeug an dem Phasenwinkel angelangt ist, an dem die neue Periapsis entstehen soll. Hier wird nun so viel Schub gegeben, dass die Apoapsis den gewollten Abstand erhält. Anschließend wird bis zur Apoapsis gewartet und dort so die Geschwindigkeit geändert, dass die Periapsis den gewollten Abstand erhält.

Inklinationsänderung

Zur Änderung der Inklination eines Orbits muss an einer Stelle des Orbits eine Geschwindigkeitsänderung orthogonal zu der aktuellen Bahnebene ausgeführt werden. Die Gleichung 2.10 berechnet die benötigte Geschwindigkeitsänderung für eine Änderung der Inklination um den Winkel Θ bei der aktuellen Geschwindigkeit V:

$$\Delta V = 2 \cdot V \cdot sin(\Theta/2) \tag{2.10}$$

Da die Geschwindigkeitsänderung abhängig von der aktuellen Geschwindigkeit ist, ist es am effektivsten, die Bahnneigung an dem Punkt mit der geringsten Momentangeschwindigkeit zu ändern, also an der Apoapsis des Orbits [36].

2.2. Satelliten

Satelliten können in natürlicher oder künstlicher Art im All vorkommen. Die Unterschiede, sowie die Besonderheiten von Satelliten sollen hier vorgestellt werden. Das Wort Satellit beschreibt ein Objekt, welches sich in einem Orbit befindet. Dabei wird grundlegend zwischen zwei Arten von Satelliten unterschieden:

Natürliche Satelliten: Natürliche Satelliten sind alle natürlichen Objekte. Darunter fallen Zwergplaneten, Monde und auch größere kleine Himmelskörper, wie einige Kometen und Asteroiden. Die meisten dieser Satelliten sind ähnlich alt wie die Planeten in ihrer Nähe und

wurden im Verlaufe der Zeit in einem stabilen Orbit gefangen. Im Sonnensystem gibt es hunderttausende bekannte Objekte[33], wobei es zur Zeit keine technischen Möglichkeiten gibt, alle Objekte zu überwachen. Trotzdem konnten bereits einige Objekte entdeckt werden, welche möglicherweise in der Zukunft auf Kollisionskurs mit der Erde geraten könnten. In KSP ist das Planetensystem zwar etwas anders aufgebaut, besteht aber aus den gleichen Komponenten. Da in KSP immer nur die Gravitation vom nächsten Planeten beachtet wird, sind die Orbits der natürlichen Satelliten zudem stabil. Im Verlauf des Projekts werden natürliche Satelliten auch immer so bezeichnet.

Künstliche Satelliten sind, soweit bekannt, von Menschenhand geschaffene Satelliten. Diese werden gezielt in verschiedene Orbits um verschiedene Körper gebracht, um diese zu erforschen oder zu überwachen und um Langstreckenkommunikation zu ermöglichen. Das Ziel der Projektgruppe umfasst die Entwicklung eines solchen Satelliten, daher sind im Folgenden unter Satelliten immer künstlichen Satelliten zu verstehen.

Satelliten bestehen aus verschiedenen Komponenten, welche die Funktionalität sicherstellen sollen. Dazu gehören Kontrolle und Überwachung von Telemetriedaten, Energieversorgung, Temperatur sowie der Flugbahn. Zudem spielt bei Satelliten natürlich auch die Sicherheit eine wichtige Rolle. Während der Start in der Regel in die Planung der Trägerrakete fällt und daher bei Satelliten keine große Rolle spielt, muss das Verhalten im Orbit um den Heimatplaneten genau geplant werden. Ein Absturz des Satelliten oder eine Kollision mit anderen Satelliten ist möglich. Gerade die Kommunikation mit anderen Satelliten ist immens wichtig. Da es keine Standards für Satelliten untereinander gibt, wurden unter anderem Richtlinien für die Betreiber entwickelt[71], um den reibungslosen Betrieb zu ermöglichen. Im Rahmen des Projekts wurde die Entscheidung getroffen, den Sicherheitsaspekt zu vernachlässigen, da dies zu einem zu großen Recherche-, Planungs- und Entwicklungsaufwand führen würde.

2.3. Kerbal Space Program

KSP stellt die Simulationsumgebung zur Verfügung, in der sich der zu steuernde Satellit befindet. Diese Simulationsumgebung ist nicht realitätsgetreu, daher werden hier die Einschränkungen und Funktionsweise von KSP vorgestellt.

In dem Spiel KSP wird dem Spieler die Kontrolle über ein Weltraumprogramm gegeben [55]. In KSP steht eine Vielzahl von Bauelementen zur Verfügung, mit denen der Spieler sich eigene Raumfahrzeuge kreieren kann [60]. Ein limitierender Faktor bei dem Designen eigener Raumfahrzeuge stellt der Preis der einzelnen Bauteile dar. Jedes Bauteil besitzt einen Preis [60] einer spielinternen Währung und verbraucht, sofern verwendet, ein Budget, das sich der Spieler auf unterschiedliche Arten erarbeiten kann [50]. Die Verfügbarkeit von Bauteilen stellt einen weiteren limitierenden Faktor dar. Alle Bauteile finden sich in einem Forschungsbaum wieder [66] aus dem für den Spieler zu Beginn des Spiels nur ein kleiner Teil verfügbar ist [66]. Möchte der Spieler neue Bauteile verwenden, muss er diese zunächst erforschen, wofür Science-Punkte benötigt werden. Mit dem Durchführen von Weltraummissionen kann sich der Spieler diese Punkte verdienen [63].

Das Spiel bietet drei Spielmodi, die dem Spieler unterschiedlich viel abverlangen. Im Karriere-Modus sind alle genannten Spielmechaniken aktiv, wodurch dieser der umfangreichste Modus ist [50]. Im Science-Modus werden die finanziellen Anteile des Spiels deaktiviert [64] und im

Sandbox-Modus wird darüber hinaus auch der Forschungsanteil bezüglich der Bauelemente deaktiviert [62].

2.3.1. Das Kerbol-System

Der Ausgangspunkt des Spiels ist das *Kerbal Space Center* auf dem Planeten Kerbin [54], welcher sich in einem Planetensystem mit einem einzelnen Stern, Kerbol, befindet [58]. Dabei handelt es sich um einen Gelben Zwerg, der im Spiel selbst nur als Sonne bezeichnet wird [57]. Das Kerbal System besteht aus fünf Planeten und zwei Zwergplaneten [58]. Im Spiel werden alle Planeten mit einer mondähnlichen Größe und einem Orbit um einen Stern als Zwergplanet klassifiziert [51].

Neben den fest definierten Himmelskörpern existieren zusätzlich Asteroiden, die in der Umgebung von Kerbol oder Dres, einem weiteren Planeten, entdeckt werden können. Asteroiden werden in verschiedene Klassen unterteilt. Die Klassifizierung reicht von Klasse A - Asteroiden mit weniger als 3 Meter Durchmesser und einem Gewicht von wenigen Tonnen, bis hin zu Klasse E - Asteroiden mit mindestens 30 Meter Durchmesser und einem Gewicht von tausenden Tonnen [48]. Im Gegensatz zu den fest definierten Himmelskörpern werden Asteroiden zufällig generiert und unterliegen vollständig der Kontrolle der Spielphysik, besitzen dafür allerdings keine eigene Gravitation. Durch diese Eigenschaft ist es möglich die Flugbahn eines Asteroiden zu beeinflussen. Sie können beispielsweise von einem möglichen Kollisionskurs abgelenkt, in einen Orbit um einen anderen Himmelskörper gebracht oder auf einem anderen Himmelskörper gelandet werden [48].

Jeder Planet, Zwergplanet und auch natürliche Satelliten besitzen so genannte Biome. Dabei handelt es sich um geographische Flächen der Oberfläche eines Himmelskörpers die typischerweise anhand ihrer Geologie unterschieden werden [49]. Der Satellit Gilly wird beispielsweise (siehe Abbildung 2.4) in drei Biome unterteilt: Lowlands, Midlands und Highlands [52]. Die Flächen eines Bioms müssen nicht zusammenhängend sein [49].

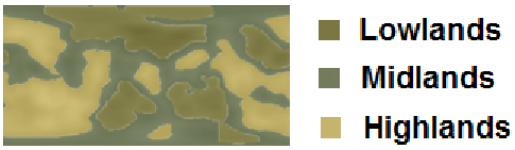


Abbildung 2.4.: Biome des Satelliten Gilly. Quelle: [52]

2.3.2. Ressourcen in Kerbal Space Program

In KSP werden verschiedene Ressourcen simuliert. Jedes Bauteil eines Raumfahrzeugskann eine oder mehrere Ressourcen verbrauchen, produzieren oder beides kombinieren [60]. Ressourcen können von einem Bauteil in ein anderes fließen. Abhängig vom Flussmodell, dem die spezielle Ressource gehorcht (siehe Tabelle 2.1). Im Spiel existieren folgende Flussmodelle [61]:

- **Adjacent**: Die Ressourcen fließen nur zwischen aneinander angrenzenden Bauteilen [61].
- **Everywhere**: Die Ressource ist immer für alle Bauteile zugänglich und wird gleichmäßig aus allen Lagerbehältern verbraucht [61].

Tabelle 2.1.: Liste der Ressourcen in Kerbal und ein Auszug ihrer Eigenschaften. Quelle: [61]

Name:	Flussmodell:	Erneuerbar:	Lagerfähig:
ElectricCharge	Everywhere	Ja	Ja
LiquidFuel	Adjacent	Nein	Ja
Oxidizer	Adjacent	Nein	Ja
IntakeAir	Everywhere	Ja	Nein
SolidFuel	Internal	Nein	Ja
MonoPropellant	Everywhere	Nein	Ja
EVA Propellant	None	Ja	Nein
XenonGas	Everywhere	Nein	Ja
Ore	Everywhere	Nein	Ja
Ablator	Internal	Nein	Nein

- **Internal**: Die Ressource kann nur von dem Bauteil verwendet werden, in dem sie enthalten ist [61].
- **None**: Die Ressource kann nur manuell transferiert werden und fließt nicht automatisch [61].

Abhängig von der Ressource muss zudem berücksichtigt werden, dass manche der Ressourcen nicht lagerfähig oder nicht erneuerbar sind. Es kann daher sein das bestimmte Ressourcen entweder sofort verbraucht werden müssen oder aber besonders sparsam mit ihnen umgegangen werden muss (siehe Tabelle 2.1) [61].

2.3.3. Modell der Gravitation

Um die Flugbahnen der verschiedenen Himmelskörper und auch Raumfahrzeuge zu erhalten ist es notwendig, Gravitation möglichst realistisch zu simulieren. In Realität können, abhängig von der Situation, mehrere Himmelskörper durch ihre Gravitation Einfluss aufeinander haben. Im Spiel wird dies vereinfacht, sodass immer nur ein Himmelskörper Einfluss auf ein anders Objekt haben kann. Diese Designentscheidung wurde getroffen, um das Modell für den Spieler leichter verständlich zu machen [65].

2.3.4. Zeit im Kerbal Space Program

Im Kerbal Space Program wird die Zeit in zwei unterschiedlichen Formen dargestellt. Zum einem eine *Universal Time*, welche die Sekunden seit dem Beginn des verwendeten Spielstands zählt [5], und einer *Mission Elapsed Time* bei der die Sekunden seit beginn einer Mission gezählt werden [74, 6]. In beiden Fällen wird zur Repräsentation der Zeit ein *double* verwendet [6, 5]. Die Zeiteinheiten Sekunde, Minute und Stunde sind im Kerbal Space Program genauso zu interpretieren wie in der Realität, jedoch unterscheiden sich Tag und Jahr (siehe Tabelle 2.2) basierend auf den Charakteristiken des Planeten Kerbin [74].

In welchem Verhältnis Zeit vergeht kann über diverse Optionen beeinflusst werden. Im Folgenden werden die von KSP selbst bereitgestellten Möglichkeiten dargestellt.

Im Allgemeinen wird die Physiksimulation des Kerbal Space Program in definierten Zeitintervallen ausgewertet. Zu diesem Zweck kann ein maximales Intervall in den Einstellungen angegeben werden. Es ist zusätzlich möglich den Verlauf der Zeit zu beschleunigen. Diese

Tabelle 2.2.: Eine Liste der möglichen *Time Warp Faktoren* und ihr Einfluss auf die reale Zeit, die im Laufe eines Tages oder Jahres auf Kerbin vergeht. [75]

time warp factor	*Kerbin Tag*			*Kerbin Jahr*			
1×	6h	0m	0.000s	106d	12h	32m	25.00s
2×	3h	0m	0.000s	53d	6h	16m	12.50s
3×	2h	0m	0.000s	35d	12h	10m	48.33s
4×	1h	30m	0.000s	26d	15h	8m	6.25s
5×	1h	12m	0.000s	21d	7h	18m	29.00s
10×		36m	0.000s	10d	15h	39m	14.50s
50×		7m	12.000s	2d	3h	7m	50.90s
100×		3m	36.000s	1d	1h	33m	55.45s
1.000×			21.600s		2h	33m	23.55s
10.000×			2.160s			15m	20.35s
100.000×			0.216s			1m	32.04s

Funktion wird als *Timewarp* bezeichnet [75]. Hierbei kann ein *time warp factor* gewählt werden, um den die Zeit beschleunigt wird. Eine Liste der möglichen Werte des *time warp factor* befindet sich in der Tabelle 2.2. Es muss zwischen zwei unterschiedlichen Modi des *Timewarp* unterschieden werden:

Physical Timewarp: In diesem Modus wird die Zeit beschleunigt und die Physiksimulation ausgeführt. Da dies zu einer besonders hohen Last auf dem Prozessor führt, wurde dieser Modus auf einen maximalen *time warp factor* von 4 beschränkt. Es ist möglich, dass dieser Modus zu Ungenauigkeiten in der Physiksimulation führt. Es wird daher Empfohlen in diesem Modus einen *time warp factor* von 2 nicht zu überschreiten. [75]

High-Speed Timewarp: In diesem Modus wird die Zeit beschleunigt und die Physiksimulation deaktiviert. Dies ermöglicht erheblich höhere *time warp factor* von bis zu 100000. Da die Physik in diesem Modus jedoch nicht berechnet wird, können während dieses *Timewarp* Modus die Antriebsraketen nicht verwendet werden. Ebenso wird die Rotation der Rakete bei Eintritt in diesen Modus gestoppt. [75]

2.3.5. Kerbal Space Program - API

KSP stellt ein Application Programming Interface (API) bereit, über die sogenannte Mods erstellt werden können, die das Spiel um neue Funktionen, Bauteile oder Spielmechaniken erweitern, beziehungsweise Existierendes ersetzen [47]. Die Dokumentation der API wird durch die Community selbst gepflegt [59, 53]. Für die 3D Umgebung verwendet das Spiel die Unity Engine [56]. Soll das Spiel um eigene Bauteile erweitert werden, ist ein für die Unity Engine geschriebenes Modell des Bauteils notwendig [67].

2.4. Fehlerinjektion

Fehlerinjektion (FI) ist ein Verfahren, um Hard- oder Software zu testen. Im Rahmen dieses Projekts wird Fehlerinjektion eingesetzt, um die Fehlertoleranz des Satelliten zu testen. Bei dem Verfahren werden Schwachstellen des zu testenden Designs ermittelt, indem ausgewählte Fehler in eine Simulation des Systems oder in das kompilierte oder laufende System injiziert

werden. Eine Übersicht über Fehlerinjektion wird in [95, 24] gegeben. In der Literatur beschreiben die Autoren oft ihr Vorgehen und ihre Architektur des Fehlerinjektionssystem. Da für dieses Projekt eine detaillierte Übersicht der Architekturen und Vorgehen weniger eine Rolle spielt, wird im Folgenden nur auf grob klassifizierte Verfahren der Fehlerinjektion und eine vorgeschlagene Grundarchitektur eines Fehlerinjektionssystems aus [24] beschrieben.

Die Verfahren der Fehlerinjektion lassen sich grob in fünf Klassen einteilen.

Hardwarebasierte Fehlerinjektion: Bei der *hardwarebasierten Fehlerinjektion* werden die Fehler direkt in die bestehende Hardware injiziert. Eine Möglichkeit ist, durch Kontakt mit den Pins die elektrischen Signale zu verändern. Neben den Kontaktverfahren werden auch kontaktlose Verfahren zur Injizierung von Fehlern benutzt. Ionenbestrahlung, elektromagnetische Impulse und Ähnliches gehören dazu. Die zeitliche Genauigkeit der Injektion ist ein Problem der kontaktlosen Verfahren. Bei beiden Verfahren besteht eine Gefahr die Hardware zu beschädigen.

Softwarebasierte Fehlerinjektion: *Softwarebasierte Fehlerinjektion* beschäftigt sich mit den Injektionen in der Software. Man kann diese zu zwei unterschiedlichen Zeitpunkten durchführen:

1. Zur *Kompilierzeit* können verschiedene Veränderungen am Quellcode vorgenommen werden. Beispielsweise kann eine Methode eine Kommunikationsnachricht bewusst ignorieren oder eine Speicherzelle mit einem falschen Wert initialiseren.
2. Zur *Laufzeit* lassen sich durch Unterbrechungensroutinen und speziell entwickelte Fehlerinjektionsmethoden Fehler injizieren, wie zum Beispiel Speicherkorruptionen. Anzumerken ist, dass die zeitliche Genauigkeit der Unterbrechungsroutinen aus softwaretechnischen Gründen nicht immer genau genug ist.

Simulationsbasierte Fehlerinjektion Bei der *simulationsbasierten Fehlerinjektion* werden Fehler in eine Simulation eines Systems oder in ein Modell injiziert. Dies erlaubt eine frühzeitige Erkennung potentieller Probleme, da die simulationsbasierte Fehlerinjektion schon kurz nach der Konzeptphase angewendet werden kann. In den meisten Fällen wird Very High Speed Integrated Circuit Hardware Description Language (VHDL) als High-Level-Modell zur simulationsbasierten Fehlerinjektion genutzt.

Emulationsbasierte Fehlerinjektion: In den neueren Arbeiten wird verstärkt auch die *emulationsbasierte Fehlerinjektion* aufgeführt. Dieses Verfahren wurde aufgrund des hohen Zeitaufwands der simulationsbasierten Fehlerinjektion entwickelt. Bei der emulationsbasierten Fehlerinjektion wird das zu testende System auf ein FPGA geladen und bietet eine Schnittstelle zu einem Host-PC, über die die Fehlerinjektionsexperimente steuerbar sind.

Hybride Fehlerinjektion: Die *hybride Fehlerinjektion* ist eine Mischung aus mehreren der zuvor genannten Verfahren. Ein Verfahren hybrider Fehlerinjektion wird in „Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits“[14] vorgestellt.

Die vorgeschlagene Grundarchitektur eines Fehlerinjektionssystems[24] (siehe Abbildung 2.5) sieht wie folgt aus: Der Controller steuert die Fehlerinjektionsexperimente. Die Systemumgebung, sowie alle anderen Eingaben werden durch den Nutzlastgenerator aus seiner Nutzlastbibliothek genommen und dem FPGA übergeben. Je nach Aktivierungsmuster injiziert

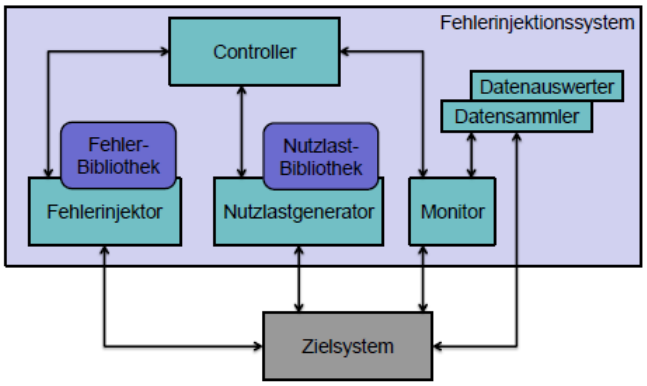


Abbildung 2.5.: Grundarchitektur der Fehlerinjektion mit den Bestandteilen und deren Kommunikationskanälen. Quelle: [28]

der Controller Fehler aus der Fehlerbibliothek mit dem Fehlerinjektor in das zu testende System. Nach einer Fehlerinjektion liest der Datensammler den Zustand des zu testenden Systems aus und vergleicht den Zustand mit dem Golden Run im Datenauswerter. Der Monitor stellt die Schnittstelle zwischen dem Benutzer, dem Controller und dem zu testenden System dar.

2.5. Das FARM-Modell

Das FARM-Modell, in [7] eingeführt, beschreibt die benötigten Komponenten einer Fehlerinjektion. FARM ist dabei ein Akronym und steht für:

F = Fehlermenge (Set of *F*aults)

A = Aktivierungsmuster (Set of *A*ctivations)

R = Ausgabe (Set of *R*eadouts)

M = Bewertung (Set of *M*easurements)

F steht dabei nicht für den gesamten Fehlerraum, sondern für die Fehlermenge, die man untersuchen möchte. Wie F für die Fehlerinjektion in diesem Projekt aussieht, wird in Abschnitt **??** beschrieben. In [7] wird die Generierung von F als ein stochastischer Prozess angesehen. In diesem Projekt, werden jedoch die Fehlerpunkte und Möglichkeiten genau definiert. Wie diese definiert sind, lässt sich erst in der Systemarchitektur und auf Modulentwurfebene

feststellen. Wann ein Fehler injiziert werden soll, wird durch die Aktivierungsmuster A angegeben. Die Ausgabe R ist die Reaktion der Satellitensteuerung auf den Fehler. Das können zum Beispiel Fehlerbehandlungsmaßnahmen oder Systemfehler sein. Diese Maßnahmen sind über den Satellitensteuerungslog sichtbar, der ebenfalls von der Fehlerinjektion zur Ausgabe an den Nutzer ausgelesen wird. Der Satellitensteuerungslog soll nicht von der Fehlerinjektion zum Injizieren von Fehlern betroffen sein. Die Bewertung M bewertet das Verhalten der Satellitensteuerung und wird aus $F \times A \times R$ bestimmt. Ob M von einem Menschen oder automatisch bestimmt wird, ist nicht durch FARM definiert. Für dieses Projekt ist eine automatische Bewertung nicht als Pflichtanforderung vorgesehen und gilt somit als Ausblick des Projektes.

2.6. Fehlertoleranz

In diesem Abschnitt werden Methoden vorgestellt, ein System auf Anfälligkeiten zu untersuchen und Maßnahmen zur Fehlererkennung und Fehlervermeidung aufgeführt. Ein Ziel des Projekts ist ein fehlertolerantes Steuersystem zu entwerfen und zu implementieren. Dazu werden in diesem Abschnitt grundlegende Methoden zur Fehlertoleranz dargestellt, die im Projekt verwendet werden können[30, 18]. Unter Fehlertoleranz versteht man die Fähigkeit eines Systems bei einer begrenzten Anzahl von fehlerhaften Komponenten seine spezifizierte Funktion weiter zu erfüllen. Qualitativ wird hier weiter in Sicherheit und in Zuverlässigkeit unterteilt. Sicherheit meint dabei, dass das System weder sich noch seine Umgebung gefährdet; Zuverlässigkeit bedeutet, dass das System seine Funktion erfüllt. Unter Fehlervermeidung, auf die hier nicht weiter eingegangen wird, versteht man durch beispielsweise geeignetere Materialien, sorgfältigeren Entwurf oder mehr Tests Fehler von vornherein zu vermeiden.

2.6.1. Fehlermodell

Ein Fehlermodell beschreibt die Struktur des Systems aus Komponenten und deren Fehlermöglichkeiten. Aus den im Fehlermodell möglichen Fehlern wird eine Menge zu tolerierender Fehlern bestimmt. Es können nicht alle Fehler toleriert werden, da beispielsweise gleichzeitig auftretende Fehler in allen Komponenten nicht toleriert werden können. Es wird aber festgelegt inwieweit gleichzeitig auftretende Fehler toleriert werden sollen. Es können auch Fehler in den Fehlertoleranzmaßnahmen auftreten, die zu tolerieren sind.

Darüber hinaus wird es ein Modell geben, das die Struktur des Systems genauer beschreibt, sodass funktionale Abhängigkeiten mitberücksichtigt werden. Dieses legt dann die Grundlage zur Beschreibung der Fehlerausbreitung. Darauf aufbauend können Maßnahmen zur Fehlereingrenzung, also die Begrenzung eines Fehlers auf einen bestimmten Bereich des Systems, entwickelt werden.

2.6.2. Redundanz

Damit im Fehlerfall die Funktionen der fehlerhaften Komponente weiter erfüllt werden kann, benötigt es eine Form der Redundanz. Als redundant werden alle Mittel bezeichnet, die im fehlerfreien Fall nicht für die spezifizierten Funktionen benötigt werden. Dabei können verschiedene Arten von Redundanz genutzt werden:

Strukturelle Redundanz: Dies bezeichnet das Hinzufügen von gleich- oder andersartigen Komponenten zum Beispiel die Verwendung mehrerer ZedBoards bietet strukturelle Redundanz der Hardware. Auch auf Softwareebene kann strukturelle Redundanz hergestellt werden.

Funktionelle Redundanz: Dies bezeichnet das Hinzufügen von nicht für die spezifizierte Funktion benötigten Funktionen zum Beispiel das Umgehen von fehlerhaften Knoten bei der Kommunikation oder die Berechnung eines Paritätsbits. Diese sind nur für den Fehlertolerierungsbetrieb.

Informationsredundanz: Informationsredundanz bezeichnet das Vorhandensein zusätzlicher Informationen zum Beispiel die mehrfache Abspeicherung einer Datei oder Hamming-Distanzen in der Codierung.

Zeitredundanz: Als Zeitredundanz wird die zusätzlich zur Verfügung stehende Zeit zur Ausführung bezeichnet, die nicht zur Erfüllung der spezifizierten Funktionen benötigt wird.

Diversität: Diversität liegt vor, wenn eine Funktion durch mehrere verschiedene Implementierungen vorliegt. Dies trägt auch zur Fehlertoleranz bei, da so Implementierungsfehler erkannt werden können.

Allgemein wird hierbei noch zwischen statischer und dynamischer Redundanz unterschieden. Statische Redundanz wird ab Betriebsbeginn eingesetzt, zum Beispiel drei Sensoren für ein 2-von-3-System. Auf dynamische Redundanz wird erst im Fehlerfall zurückgegriffen. Statische Redundanz kann auch durch dynamische Redundanz verändert werden, indem zum Beispiel bei Vorhandensein eines vierten Sensors im Fehlerfall des 2-von-3-Systems der fehlerhafte Sensor ausgetauscht wird.

2.6.3. Fehlererkennung

Die Fehlertoleranzmaßnahmen müssen unterscheiden können, ob alle Komponenten fehlerfrei funktionieren oder ein Fehler aufgetreten ist. Dazu werden Testmethoden eingesetzt. Nach dem Erkennen eines Fehlers muss es weitere Testmethoden geben, um diesen Fehler zu lokalisieren, um entsprechend fehlerhafte Komponenten auszutauschen oder zu reparieren.

2.6.4. Fehlerkorrektur

Durch das Nutzen der Redundanz können viele Fehler schon dadurch korrigiert werden, dass eine Funktion von mehreren Komponenten zur Verfügung gestellt wird und durch Mehrheitsentscheid nur die richtige Lösung an andere Komponenten weitergegeben wird. Die alleinige Behebung der Fehlerursache reicht meistens nicht aus, um in einen funktionsfähigen Zustand zurückzukehren, da sich der Fehler schon ausgebreitet haben kann. Fehlerhafte Komponenten müssen durch Rekonfigurierung ausgetauscht werden und alle anderen Komponenten in einen funktionsfähigen Zustand geführt werden. Dies kann durch Wiederherstellung eines vorherigen fehlerfreien Zustands erfolgen oder durch Bestimmung eines solchen Zustands. Letzteres erfordert sehr genaue Kenntnisse über das System und seine Umgebung.

2.7. SystemC

Um die Satellitensteuerung zu modellieren hat sich die Projektgruppe für die C++-Bibliothek SystemC entschieden. Aus diesem Grund werden nun die Funktionsweise und die Grundlagen von SystemC erläutert.

Für die Entwicklung des Projekts wird zunächst ein logisches Modell der Satellitensteuerung entworfen. Dies geschieht mit der SystemC-Bibliothek für C++. Die Grundlagen und der Nutzen der SystemC-Bibliothek wird daher im Folgenden näher erläutert.

Bevor die Entwicklung mit Hardware/Software-Codesign Strategien möglich wurde, musste die Hardware mit Hardwarebeschreibungssprachen wie VHDL modelliert werden, während die Software mit höheren Programmiersprachen wie C und C++programmiert wurde. Um eine

höhere Parallelität bei der Entwicklung zu schaffen, wurden Konzepte erforscht, die Hardware und Software Entwicklung in einer Sprache ermöglichen. Aus dieser Forschung ist die SystemC-Bibliothek entstanden[29].

SystemC ermöglicht es sowohl Hardware- wie auch Softwarekomponenten zu erstellen, ohne explizit die Komponenten in Hardware und Software zu unterteilen. Ein weiterer Vorteil ist die Ausführbarkeit des Modells, die es ermöglicht bereits frühzeitig Fehler in den Algorithmen zu lokalisieren. Durch das höhere Abstraktionslevel der Modellierung für die Hardware im Vergleich zu VHDL wurden Synthesewerkzeuge entwickelt. Mit Hilfe der Synthesewerkzeuge ist eine Transformation des Modells in eine physikalische Implementierung möglich[29].

In Abbildung 2.6 sind die Komponenten der SystemC-Bibliothek dargestellt. Da C++bereits die Aufgaben bei der Softwareentwicklung erfüllt, erweitert SystemC die Funktionalität hauptsächlich für die Hardwareentwicklung. Daher ist es nicht überraschend, dass der Inhalt sehr an die Konstrukte von VHDL erinnern[11].

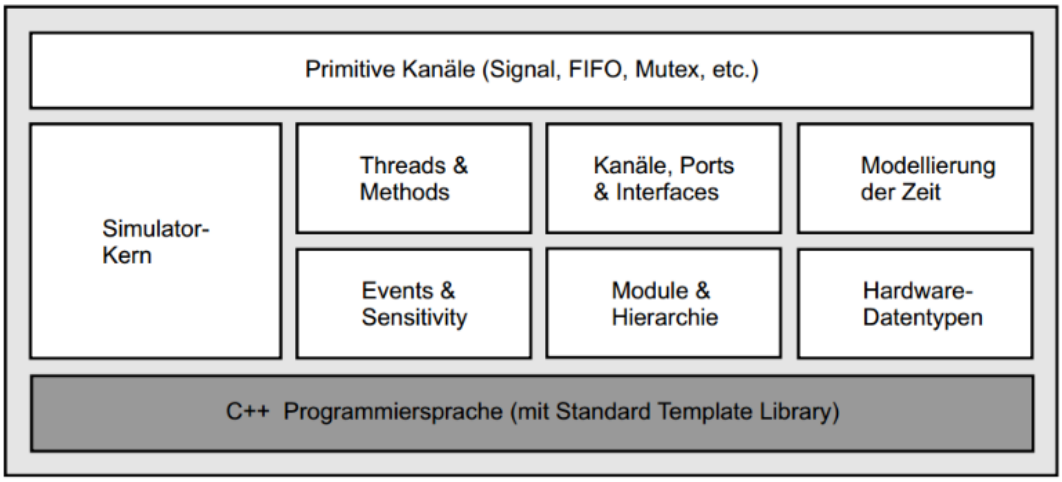


Abbildung 2.6.: Inhalt und Aufbau der SystemC-Bibliothek. Quelle: [29]

In SystemC sind die folgenden Komponenten enthalten[29]:

Simulator-Kern: Der Simulator ist für die Steuerung der Prozesse zuständig. Im Grunde ermöglicht SystemC eine zeitdiskrete ereignisgesteuerte Simulation auszuführen.

Nebenläufigkeit: *Threads* und *Methods* sind die Konstrukte, durch die eine Parallelität der Hardware modelliert wird.

Reaktivität: Die ausgeführten Prozesse werden über *Events* gesteuert und synchronisiert. Durch die Sensitivitätsliste können Prozesse gestartet werden.

Hierarchie: Die Hierarchie in SystemC wird über *Module* vorgenommen. Die Module entsprechen den *Entity* und *Architecture*-Konstrukten aus VHDL.

Kommunikation: Ports und Kanäle haben die gleiche Funktionalität wie in VHDL. Die Module können Ports aufmachen, die eine Kommunikation über Kanäle zu weiteren Modulen ermöglicht.

Primitive Kanäle: Die Kommunikation zwischen den *Modulen* wird über Kanäle realisiert. Die *primitiven Kanäle* sind einfache, vordefinierte Implementierungen. Zu diesen gehören die Signale wie auch die FIFOs.

Hardware-Datentypen: Die *Hardware-Datentypen* sind stark angelehnt an die Möglichkeiten in VHDL. Durch die Verwendung der *Hardware-Datentypen* ist die Abstraktion zur Hardwareentwicklung näher und vereinfacht den Prozess der Synthese. So sind unter anderem Datentypen für mehrwertige Signalzustände vorhanden.

Modellierung der Zeit: Die Modellierung des Zeitverhaltens wird über die SystemC-Klasse *sc_time* ermöglicht. Über bestimmte Befehle ist ein Fortlauf der Zeit modellierbar. Ebenso ist es möglich die aktuelle Zeit abzufragen und über *Events* auf bestimmten Zeitschranken Aktionen auszulösen.

2.8. Co-Simulation

In dieser Projektgruppe wird mit mehreren Modellen gearbeitet. Das miteinander Arbeiten dieser Modelle wird als Co-Simulation bezeichnet. Durch die Verbindung mehrerer Modelle kann so das Gesamtsystem abgebibldet werden. Die einzelnen Modelle kommunizieren miteinander, indem zu definierten Zeitpunkten Daten ausgetauscht werden. Zwischen diesen Zeitpunkten berechnet jedes Modell mit Hilfe der Daten den Teil des Systems, der durch das jeweilige Modell abgebildet wird [vgl. 34].

Nach Lajolo, Lazarescu und Sangiovanni-Vincentelli besteht eine Co-Simulation aus folgenden Phasen:

- *Initialisierung:* Alle notwendigen Informationen über die beteiligten Modelle werden gesammelt.
- *Simulation:* Alle Modelle werden gemeinsam schrittweise gelöst und die sich ergebenden Werteänderungen zwischen allen Modellen kommuniziert.
- *Terminierung:* Alle Modelle werden ordnungsgemäß beendet. [34]

2.9. Hardware

Für die Durchführung des Projekts standen der Gruppe ein Projektbudget von 1000 €, ein handelsüblicher PC, der die Mindestanforderungen von KSP erfüllt, sowie drei ZedBoards zur Verfügung. Von dem Geld wurden im Laufe des Projekts ein weiteres ZedBoard, sowie ein weiterer PC angeschafft, um die Fehlerinjektion und den Simulator zu trennen.

Beim ZedBoard handelt es sich um ein FPGA-Board der Firma *Digilent*. Die genaue Produktbezeichnung ist *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. Die Bezeichnung verrät bereits, dass sich sowohl ein ARM-Prozessor, sowie ein FPGA auf dem Board befinden, genauer handelt es sich um einen *dual-core ARM Cortex-A9*-Prozessor sowie eine programmierbare Logik der 7er-Serie [26]. Es stehen außerdem folgende Speicher zur Verfügung: 512 MB DDR3, 256 MB Quad-SPI Flash und 4 GB SD-Karte. Für dieses Projekt von besonderer Bedeutung ist der 10/100/1000 Ethernet-Anschluss, da das Verwenden von Ethernet Teil der Projektvorgaben war. Weitere Informationen können dem Datenblatt [41] entnommen werden.

3. Mission

Um zu prüfen, ob sich der Satellit so verhält, wie die Satellitensteuerung es vorsieht, wird dieser in der Simulationsumgebung KSP simuliert. Dies geschieht im Rahmen einer Mission, die im Folgenden motiviert und phasenweise erläutert wird.

3.1. Motivation

Bei der Wahl der Mission mussten mehrere Faktoren beachtet werden:

Erfüllung der Projektanforderungen: An die Projektgruppe wurden Anforderungen bezüglich der Simulation gestellt, welche erfüllt werden müssen.

Interessanter Missionsverlauf: Die gewählte Mission soll sowohl für einen externen Betrachter als auch für das Projektteam interessant gestaltet werden.

Realisierbarkeit in der Simulation: Die Mission muss in der zur Verfügung gestellten Simulationsumgebung KSP realisierbar sein. Gerade die Astrophysik von KSP gibt hier einige Grenzen vor.

Zeitliche Realisierbarkeit: Die Mission darf nicht zu komplex oder umfangreich sein, da der Projektgruppe nur eine begrenzte Zeit für den Arbeitsauftrag zur Verfügung steht und neben der Mission auch weitere Bestandteile des Projektes realisiert werden müssen.

Diese Faktoren ermöglichen große Wahlfreiheit bezüglich der Mission, jedoch unter den Einschränkungen, die die Simulationsumgebung mit sich bringt. Im Rahmen der wöchentlichen Projektgruppentreffen konnte mit Hilfe der Informationen aus der Seminarphase und mit dem Feedback der Betreuer eine Eingrenzung der möglichen Missionen durchgeführt werden, indem auf die oben genannten Faktoren hin evaluiert wurde. Im Rahmen der wöchentlichen Treffen wurde die folgende Misison festgelegt.

3.2. Missionsziel

ach der groben Evaluation ist die Wahl der Gruppe auf eine spannende Mission gefallen, welche sich im Rahmen der Mission Sentinel [20] und des Asteroid Days [73] einordnen lässt.

Das Sentinel-Projekt hat es sich zur Aufgabe gemacht, in unserem Sonnensystem Asteroiden zu finden und deren Flugbahn zu berechnen, um frühzeitig einen möglichen Kollisionskurs mit der Erde zu erkennen. Auf Grundlage dieser Daten sollen anschließend Projekte gestartet werden, welche den Asteroiden von seiner Flugbahn abbringen und den Einschlag auf der Erde verhindern.

Diese Idee nimmt sich die KSP-Asteroid Day-Mission als Vorlage und gibt dem Spieler den Auftrag Asteroiden zu finden und diese von ihrem Weg Richtung Kerbin abzubringen.

3.3. Aufgaben des Satelliten

Ist der Satellit am Asteroiden angekommen, kann er mit seinem Missionziel beginnen. Unser Satellit soll sich dabei dem Asteroiden annäheren. Anschließend soll eine Landung stattfinden und der Satellit mit einem Greifer an dem Asteroiden befestigt werden. Zuletzt soll der Asteroid gezielt aus seinem Orbit bewegt werden, um die Fähigkeiten des Satelliten zu de-

monstrieren. Ein solcher Satellit könnte in der Realität eingesetzt werden, um selbstständig Asteroiden umzuleiten, welche für die Erde gefährlich werden würden.

3.4. Systemabgrenzung

Das System und dessen einzelne Komponenten werden unter Entwurf & Implementierung beschrieben. Daraus ergeben sich drei Teilprojekte: Satellitensteuerung, Fehlerinjektion und Simulator. In diesem Abschnitt soll es hauptsächlich um Belange gehen, die dieses Projekt nicht betreffen, beziehungsweise den Rahmen des Projekts oder der Teilprojekte sprengen würden. Im Kontrast dazu werden auch kurz Belange gelistet, die noch im Rahmen des Projekts liegen. Erstere sind in den folgenden Unterabschnitten stichpunktartig festgehalten. Letztere fassen im Grunde dieses Dokument zusammen und dienen nur der Vollständigkeit.

4. Anwendungsfälle

In diesem Abschnitt werden die für dieses Projekt generierten Anwendungsfälle aufgeführt. Für Anwendungsfälle wurde eine tabellarische Form gewählt. Neben einem Namen und eindeutigem Schlüssel enthält die tabellarische Form der Anwendungsfälle folgende Informationen:

Akteur: Als Akteur werden alle involvierten Subkomponenten des erstellten Systems verstanden.

Hauptakteur: Als Hauptakteur wird der Akteur verstanden, der den Anwendungsfall auslöst.

Vorbedingung: Unter dem Punkt Vorbedingung wird beschrieben, von welchem Zustand des Systems ausgegangen wird.

Nachbedingung: Unter dem Punkt Nachbedingungen wird beschrieben, welcher Zustand des Systems erreicht wird, nachdem der Anwendungsfall ausgeführt wurde.

Fachlicher Auslöser: Als fachlicher Auslöser wird die Ursache oder das Ereignis bezeichnet, das den Anwendungsfall auslöst.

Ablauf: Im Ablauf wird in kurzen Stichpunkten erläutert, was in dem Anwendungsfall ausgeführt wird.

Alternativer Ablauf: Bei einigen Anwendungsfällen gibt es einen alternativen Ablauf, der eine alternative Ausführung beschreibt, da diese zu diesem Zeitpunkt noch nicht eindeutig festgelegt werden kann.

Diagram: Wurde der Anwendungsfall in einem Diagramm dargestellt, wird unter diesem Punkt auf die entsprechende Abbildung verwiesen.

Dieses Kapitel ist, angelehnt an die Projektarchitektur, dreigeteilt. Im Abschnitt 4.1 werden Anwendungsfälle aus dem Teilsystem Simulator zur Simulationsumgebung angegeben, Abschnitt 4.2 behandelt die Anwendungsfälle für das Teilsystem Satellitensteuerung und in Abschnitt 4.3 geht es um das Teilsystem Fehlerinjektion, also der Schnittstelle, die dem Anwender erlaubt Fehler einzuspielen. Zunächst sollen jedoch die Akteure der gleich folgenden Anwendungsfälle einmal vorgestellt werden:

Simulator: Hier wird die Umwelt des Satelliten und der Satellit selbst simuliert. Dazu zählen der Weltraum, Himmelsobjekte und deren Physik. Des Weiteren wird hier auch die Sensorik und Aktorik des Satelliten simuliert. Dazu zählen u.A. Antriebsdüsen, Sonnenkollektoren und Treibstoffbehältnisse.

Steuerung: Als abstrakter Oberbegriff für alle Programme und Algorithmen, die in Hard- oder Software die Steuerung des Satelliten übernehmen.

Fehlerinjektion: Als abstrakter Oberbegriff für alle Programme und Algorithmen, die in Hard- oder Software Fehler in die Steuerung des Satelliten injizieren.

Log: Zentraler Speicher für alle eingehenden Befehle durch Steuerung oder Fehlerinjektion, sowie Verhalten des Simulators.

Satellit: Beschreibt den Satelliten und seine Aufgaben als Ganzes. Der Satellit befindet sich virtuell in der Simulationsumgebung, seine Steuerung befindet sich auf den ZedBoards, bzw. als Modell auf dem Host-PC.

Anwender: Der Nutzer kann Fehler über die Benutzerschnittstelle eingeben und das Log-File auslesen, also mit dem Host-PC interagieren.

Bodenstation: Die Bodenstation hält Kontakt zum Satelliten und kann unter Umständen Anweisungen an diesen versenden.

FI-ZedBoard: Dieser Akteur beschreibt den Teil der Fehlerinjektion, der sich auf dem ZedBoard befindet.

4.1. Simulator

Der Simulator stellt die Simulationsumgebung für den Satelliten zur Verfügung. Darüber versorgt er den Satelliten mit Sensordaten und stellt ihm die Umwelt zur Verfügung.

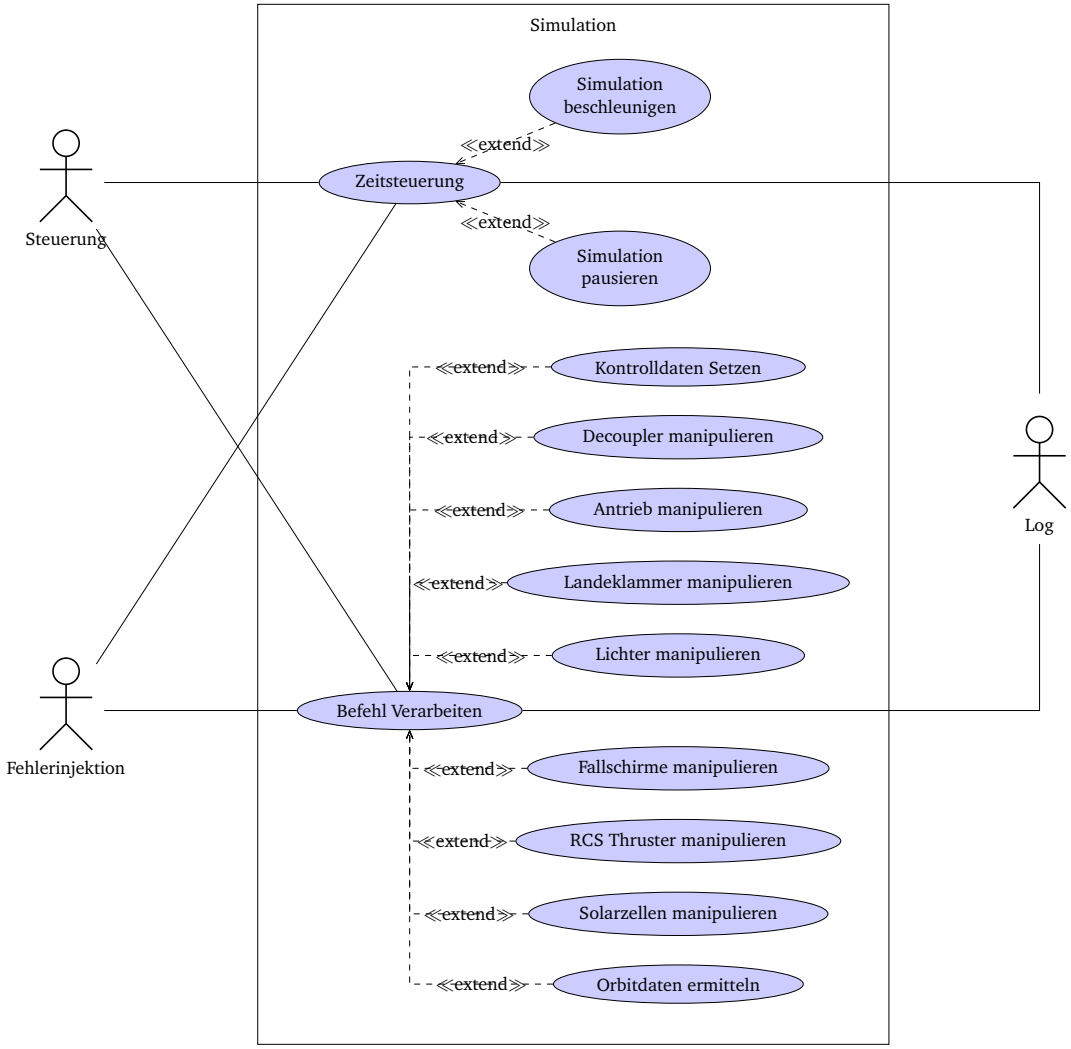


Abbildung 4.1.: Anwendungsfalldiagramm des Simulators

Kennzeichnung:	**UC001**
Bezeichnung:	**Befehl verarbeiten**
Hauptakteur, Akteur:	Steuerung, Fehlerinjektion, Log
Vorbedingungen:	• Es besteht eine Kommunikationsschnittstelle zwischen der Steuerung, Fehlerinjektion, Log und dem Simulator.
Nachbedingungen:	• Die gesendeten Befehle werden von den Adaptern erfolgreich ausgeführt.
Fachlicher Auslöser:	Die Steuerung oder Fehlerinjektion sendet einen Befehl an die Simulation.
Ablauf:	• Der Befehl wird von der Steuerung/Fehlerinjektion and den Simulator gesendet. • Der Befehl wird indentifiziert. • Der Befehl wird an die entsprechenden Adapter weitergeleitet.
Alternativer Ablauf 1:	• Der Befehl wird von der Steuerung/Fehlerinjektion and den Simulator gesendet. • Der Befehl wird nicht indentifiziert. • Der Befehl wird nicht verarbeitet.
Diagram:	siehe Abbildung 4.1

In Abbildung 4.1 sind die Anwendungsfälle des Simulators dargestellt.

Kennzeichnung:	**UC002**
Bezeichnung:	**Kontrolldaten setzen**
Hauptakteur, Akteur:	Simulator
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC003**
Bezeichnung:	**Decoupler manipulieren**
Hauptakteur, Akteur:	Simulator
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC004**
Bezeichnung:	**Antrieb manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC005**
Bezeichnung:	**Landeklammer manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC006**
Bezeichnung:	**Lichter manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC007**
Bezeichnung:	**Fallschirme manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC008**
Bezeichnung:	**RCS Thruster manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC009**
Bezeichnung:	**Solarzellen manipulieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC010**
Bezeichnung:	**Orbitdaten ermitteln**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC011**
Bezeichnung:	**Zeitsteuerung**
Hauptakteur, Akteur:	System, Simulator, Log
Vorbedingungen:	• Es besteht eine Kommunikationsschnittstelle zwischen der Steuerung, Fehlerinjektion, Log und dem Simulator.
Nachbedingungen:	• Die gesendeten Befehle werden von den Adaptern erfolgreich ausgeführt.
Fachlicher Auslöser:	Die Steuerung oder Fehlerinjektion sendet einen Befehl an die Simulation.
Ablauf:	• Der Befehl wird von der Steuerung/Fehlerinjektion and den Simulator gesendet. • Der Befehl wird indentifiziert. • Der Befehl wird an die entsprechenden Adapter weitergeleitet.
Alternativer Ablauf 1:	• Der Befehl wird von der Steuerung/Fehlerinjektion and den Simulator gesendet. • Der Befehl wird nicht indentifiziert. • Der Befehl wird nicht verarbeitet.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC012**
Bezeichnung:	**Simulation beschleunigen**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

Kennzeichnung:	**UC013**
Bezeichnung:	**Simulation pausieren**
Hauptakteur, Akteur:	
Vorbedingungen:	• Vom Adapter aus kann auf die Komponenten des Simulators dediziert zugegriffen werden. • Es wurde ein Befehl von der Steuerung/Fehlerinjektion an den Simulator gesendet.
Nachbedingungen:	• Die gesendeten Befehle werden von den Komponenten ausgeführt.
Fachlicher Auslöser:	Der Adapter der Kontrolldaten hat einen Befehl von der Steuerung/Fehlerinjektion bekommen.
Ablauf:	• Der Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die betreffende Komponente führt den Befehl aus.
Alternativer Ablauf 1:	• Ein fehlerhafter Befehl wird vom Adapter an die betreffende Komponente gesendet. • Die Simulation führt den Befehl aus.
Diagram:	siehe Abbildung 4.1

4.2. Satellit

Die Satellitensteuerung hat zwei primäre Aufgaben (Abbildung 4.2). Zum einen ist sie für die Flugsteuerung verantwortlich, zum anderen für die Energieverwaltung. Beide Aspekte sollen nun in Form von Anwendungsfällen mit ergänzenden Anwendungsfalldiagrammen vorgestellt werden.

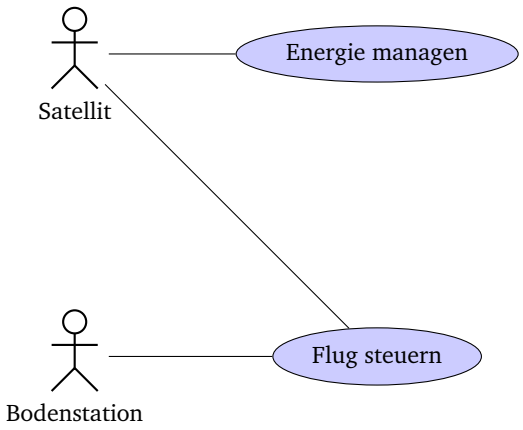


Abbildung 4.2.: Anwendungsfalldiagramm des Satelliten

4.2.1. Flugsteuerung

Um die Mission erfolgreich durchführen zu können müssen bestimmte Aktionen durch den Satelliten oder gegebenenfalls die Bodenstation durchgeführt werden können. Diese werden im Anwendungsfalldiagramm in Abbildung 4.3 dargestellt.

Ein Teil der dort aufgeführten Aktionen, nämlich Geschwindigkeit ändern (Anwendungsfall UC014), Orientierung ändern (Anwendungsfall UC015), Orbit anpassen (Anwendungsfall UC016)und Route fliegen (Anwendungsfall UC017), beziehen sich dabei auf die Fortbewegung des Satelliten. Weiter wird mit dem Anwenden von Adaptionsverfahren (Anwendungsfall UC018) die Reaktion des Satelliten auf auftretende Fehler beschrieben. Die Aktionen mit Bodenstation kommunizieren (Anwendungsfall UC019) und Instrumente ausrichten (Anwendungsfall UC020) gehen auf die Benutzung von Satellitenkomponenten ein.

Kennzeichnung:	**UC014**
Bezeichnung:	**Geschwindigkeit ändern**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Satellit ist abgehoben. • Der Orbit muss angepasst werden. • Der Satellit besitzt genügend Treibstoff um die Geschwindigkeitsänderung durchzuführen.
Nachbedingungen:	• Der Satellit hat die gewollte Geschwindigkeit erreicht.
Fachlicher Auslöser:	System fordert Geschwindigkeitsänderung an.
Ablauf:	• Eine Anfrage die Geschwindigkeit zu ändern wird bei dem Steuersystem des Satelliten eingereicht. • Der Satellit interpretiert die Signale und vergleicht die Soll- mit der Ist-Stellung. • Der Satellit regelt die Aktuatoren, um die gewollte Geschwindigkeit zu erreichen.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC015**
Bezeichnung:	**Orientierung ändern**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Satellit ist abgehoben. • Eine Orientierungsanpassung ist notwendig. • Der Satellit besitzt über genügend Energie um die Orientierungsänderung durchzuführen.
Nachbedingungen:	• Der Satellit besitzt die gewollte Orientierung.
Fachlicher Auslöser:	System fordert Orientierungsänderung an.
Ablauf:	• Eine Anfrage die Orientierung zu ändern wird bei dem Steuersystem des Satelliten eingereicht. • Der Satellit interpretiert die Signale und vergleicht die Soll- mit der Ist-Stellung. • Der Satellit regelt die Rotationsaktuatoren, um die gewollte Orientierung zu erreichen.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC016**
Bezeichnung:	**Orbit anpassen**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Satellit ist abgehoben. • Der Satellit besitzt genügend Treibstoff um den Orbit anzupassen.
Nachbedingungen:	• Der Satellit befindet sich auf der gewollten Umlaufbahn.
Fachlicher Auslöser:	Das System fordert eine Orbitbahnanpassung an oder ein Befehl zur Orbitanpassung wird von der Bodenstation gesendet.
Ablauf:	• Eine Anfrage die Orbitbahn zu ändern wird bei dem Steuersystem des Satelliten eingereicht. • Die benötigten Steuervorgänge zum Erreichen des Orbits werden bestimmt. • Der Satellit führt die Steuervorgänge nacheinander zur im richtigen Moment aus, um die gewollte Orbitbahn zu erreichen.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC017**
Bezeichnung:	**Route fliegen**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Die autonome Steuerung des Satelliten ist aktiviert. • Der Satellit besitzt genügend Treibstoff um die gewollte Route vollständig durchzuführen.
Nachbedingungen:	• Der Satellit befindet sich auf einer Umlaufbahn zum Zielpunkt.
Fachlicher Auslöser:	Ziel ist in der Systemkonfiguration festgelegt.
Ablauf:	• Alle benötigten Orbitanpassungen zum Erreichen des Zieles werden berechnet. • Die Orbitanpassungen werden nacheinander, am jeweils zuvor bestimmten Ort, durchgeführt.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC018**
Bezeichnung:	**Adaptionsverfahren anwenden**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Satellit ist außerhalb der Atmosphäre. • Der Satellit steuert auf eine Gefahrensituation zu oder es liegt ein Fehler vor, der zu einer Gefahrensituation führen kann.
Nachbedingungen:	• Der Satellit befindet sich nicht in der Gefahrensituation.
Fachlicher Auslöser:	Eine Gefahrensituation für den Satelliten ist bevorstehend.
Ablauf:	• Art der Gefahrensituation wird bestimmt und mögliche Vermeidungsstrategien werden bestimmt. • Die nötigen Orbitanpassungen oder Orientierungsänderungen um die Vermeidungsstrategie durchzuführen werden berechnet. • Die Anpassungs- und Änderungsschritte werden nacheinander ausgeführt.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC019**
Bezeichnung:	**Mit Bodenstation kommunizieren**
Hauptakteur, Akteur:	Satellit, Bodenstation
Vorbedingungen:	• Es ist genügend Energie vorhanden um die geforderte Aktion auszuführen.
Nachbedingungen:	• Die geforderte Aktion wurde ausgeführt.
Fachlicher Auslöser:	Der Satellit einen Steuerbefehl von der Bodenstation.
Ablauf:	• Deaktiviere autonome Steuerung des Satelliten. • Der Steuerbefehl wird ausgeführt. • Der Status der Ausführung wird zurückgesendet.
Diagram:	siehe Abbildung 4.2

Kennzeichnung:	**UC020**
Bezeichnung:	**Instrument ausrichten**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Es ist genügend Energie vorhanden um die geforderte Aktion auszuführen. • Ein Instrument benötigt eine Änderung der Ausrichtung.
Nachbedingungen:	• Die Orientierung des Satelliten entspricht der benötigten Orientierung, damit das Instrument ausgerichtet ist.
Fachlicher Auslöser:	Ein Messinstrument oder ein Kommunikationsinstrument benötigt eine Änderung der Ausrichtung.
Ablauf:	• Bestimme die benötigte Orientierungsänderung des Satelliten, damit die Ausrichtung des Instrumentes erreicht wird. • Drehe den Satelliten, bis die berechnete Orientierung erreicht ist.
Diagram:	siehe Abbildung 4.2

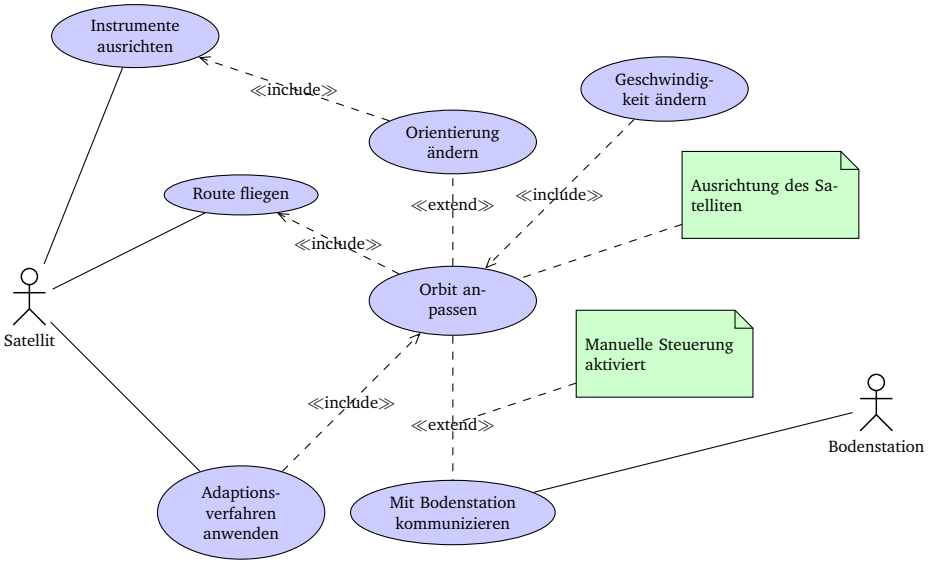


Abbildung 4.3.: Anwendungsfalldiagramm: Flugsteuerung

4.2.2. Energieverwaltung

Die Energieverwaltung spielt bei der gewählten Mission eine entscheidende Rolle. Aus diesem Grund wurden die wesentlichen Merkmale im Anwendungsfalldiagramm in Abbildung 4.4 festgehalten. Hinzu kommen erste Aspekte, welche in der Fehlerinjektion auftreten können, wie beispielsweise „Temperatur aufzeichnen“ (Anwendungsfall UC024) und „Sonnenkollektoren ausrichten“ (Anwendungsfall UC025), da diese das Verhalten des Satelliten beeinflussen. Weiter wird das Energiemanagement des Satelliten, im speziellen das Aktivieren (Anwendungsfall UC021) und das Deaktivieren (Anwendungsfall UC022) von Verbrauchern, sowie das Aufzeichnen des Energieverbrauchs (Anwendungsfall UC023), eingegangen.

Kennzeichnung:	**UC021**
Bezeichnung:	**Verbraucher aktivieren**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Verbraucher ist im Ruhemodus. • Die Nutzung des Verbrauchers steht bevor.
Nachbedingungen:	• Der Verbraucher kann seine Aufgabe verrichten.
Fachlicher Auslöser:	Satellit benötigt Verbraucher, um spezifische Aufgaben durchführen zu können.
Ablauf:	• Der Satellit sendet einen Befehl an den Verbraucher, dass dieser aktiviert werden muss.
Diagram:	siehe Abbildung 4.4

Kennzeichnung:	**UC022**
Bezeichnung:	**Verbraucher deaktivieren**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Der Verbraucher ist aktiviert. • Die Energiereserven erlauben ein weiteres Betreiben des Verbrauchers nicht oder der Verbraucher wird nicht mehr benötigt.
Nachbedingungen:	• Der Verbraucher ist im Ruhezustand oder im niedrigsten Energieverbrauch.
Fachlicher Auslöser:	Der Satellit benötigt den Verbraucher nicht mehr im aktuellen Zustand oder die Energiereserven lassen ein weiteres Betreiben nicht zu.
Ablauf:	• Der Satellit sendet einen Befehl an den Verbraucher, dass dieser deaktiviert werden muss.
Diagram:	siehe Abbildung 4.4

Kennzeichnung:	**UC023**
Bezeichnung:	**Energieverbrauch aufzeichnen**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Energie wird verbraucht. • Es liegen Parameter zur Aufzeichnung des Energieverbrauchs vor. • Ein Kommunikationskanal zum Simulator ist vorhanden.
Nachbedingungen:	• Der Energieverbrauch wurde zur Auswertung erfolgreich übertragen.
Fachlicher Auslöser:	Aktivierung eines Verbrauchers oder Start des Satellitensystems
Ablauf:	• Das Satellitensystem wird gestartet oder ein Verbraucher wird aktiviert. • Die Parameter zur Erfassung der Daten werden gesetzt. • Der Energieverbrauch wird an den Simulator gesendet.
Diagram:	siehe Abbildung 4.4

Kennzeichnung:	**UC024**
Bezeichnung:	**Temperatur aufzeichnen**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Die Sensoren sind funktionstüchtig. • Es liegen Parameter zur Aufzeichnung der Temperatur vor. • Ein Kommunikationskanal zum Simulator ist vorhanden.
Nachbedingungen:	• Die Temperatur wurde zur Auswertung erfolgreich kommuniziert.
Fachlicher Auslöser:	Aktivierung eines Verbrauchers oder Start des Satellitensystems
Ablauf:	• Das Satellitensystem wird gestartet oder ein Verbraucher wird aktiviert. • Die Parameter zur Erfassung der Temperaturdaten werden gesetzt. • Die Temperaturdaten werden an den Simulator gesendet.
Diagram:	siehe Abbildung 4.4

Kennzeichnung:	**UC025**
Bezeichnung:	**Sonnenkollektoren ausrichten**
Hauptakteur, Akteur:	Satellit
Vorbedingungen:	• Die Sonnenkollektoren sind funktionstüchtig. • Die Sonnenkollektoren sind nicht optimal ausgerichtet. • Der Satellit befindet sich nicht im Schatten eines anderen Körpers. • Die Energie zum weiteren Betrieb des Satelliten wird benötigt.
Nachbedingungen:	• Die Sonnenkollektoren sind ausgerichtet.
Fachlicher Auslöser:	Sonnenkollektoren sind nicht optimal ausgerichtet oder Energie wird zum Erreichen des Missionsziels benötigt.
Ablauf:	• Die Sonnenkollektoren werden neu ausgerichtet.
Diagram:	siehe Abbildung 4.4

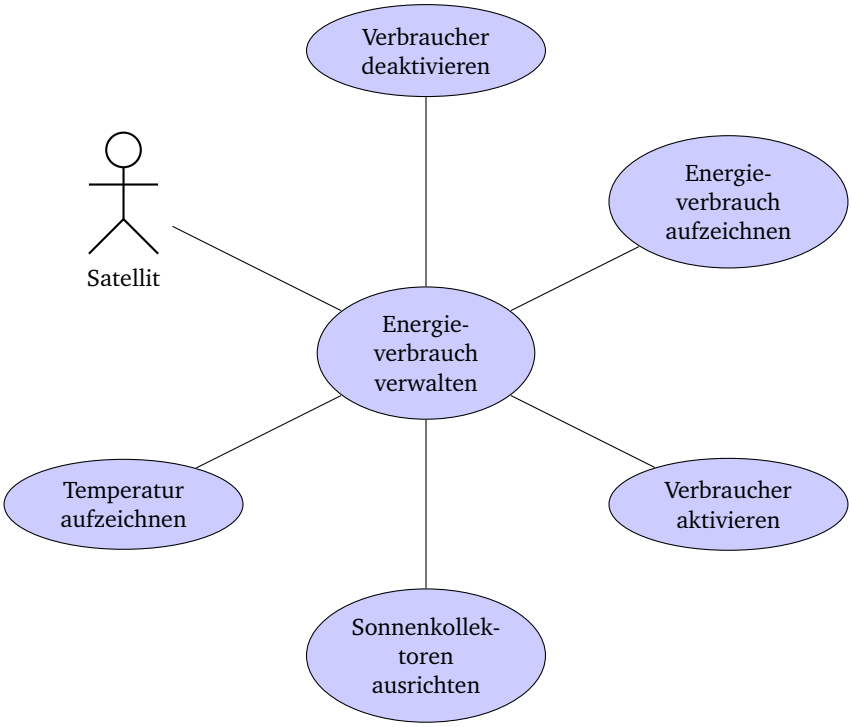


Abbildung 4.4.: Anwendungsfalldiagramm: Energieverwaltung des Satelliten

4.3. Fehlerinjektion

Die Fehlerinjektion (FI) ist das Kettenglied zwischen dem Satelliten und der Simulation. Sie kann sowohl Sensordaten als auch Aktuatoren beeinflussen. Die unterschiedlichen Anwendungsfälle werden nun vorgestellt und durch Anwendungsfalldiagramme ergänzt.

Dazu werden zunächst die Anwendungsfälle möglicher injizierbarer Fehlertypen dargestellt und beschrieben. Diese Fehlertypen sind das Manipulieren von Umweltsignalen (Anwendungsfall UC026 & Abbildung 4.5), das Manipulieren von Steuersignalen (Anwendungsfall UC027 & Abbildung 4.6) und das Manipulieren des Satellitenzustandes (Anwendungsfall UC028 & Abbildung 4.2).

Weiter wird die Erzeugung von Fehlerexperimenten aus den zuvor aufgeführten Fehlertypen in Abbildung 4.8 und Anwendungsfall UC029 veranschaulicht und erläutert.

Außerdem wird die Erstellung und die Ausgabe eines Fehlerkatalogs, in dem alle injizierbaren Fehler aufgeführt werden, in den Anwendungsfällen UC030 und UC031 beschrieben und in Abbildung Abbildung 4.9) dargestellt.

Zuletzt werden das Laden eines Systemzustandes auf dem FI-ZedBoard in Anwendungsfall UC032 und Abbildung 4.10, sowie das Monitoring der Fehlerinjektion in Anwendungsfall UC033 und Abbildung 4.11 dargestellt und beschrieben.

Kennzeichnung:	**UC026**
Bezeichnung:	**Umweltsignale manipulieren**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, Satellit, Simulator
Vorbedingungen:	• Die Umweltsignale werden vom Simulator empfangen. • Die Verbindung zur Fehlerinjektion ist initialisiert worden. • Die Verbindung zum Satelliten ist initialisiert worden.
Nachbedingungen:	• Die Reaktion auf die manipulierten Umweltsignale ist im Simulator oder in den Systemlogs sichtbar.
Fachlicher Auslöser:	Der Anwender hat den Befehl zur Umweltsignalmanipulation gegeben.
Ablauf:	• Die Fehlerinjektion erhält den Befehl zur Manipulation von Umweltsignalen vom Anwender. • Die Fehlerinjektion manipuliert die Umweltsignale. • Die manipulierten Umweltsignale werden an den Satelliten weitergeleitet. • Die Reaktion ist in der Fehlerinjektion oder im Simulator sichtbar.
Diagram:	siehe Abbildung 4.5

Kennzeichnung:	**UC027**
Bezeichnung:	**Steuersignale manipulieren**
Hauptakteur, Akteur:	Fehlerinjektion, Satellit, Simulator, Anwender
Vorbedingungen:	• Die Steuersignale werden vom Simulator empfangen. • Die Verbindung zur Fehlerinjektion ist initialisiert worden. • Die Verbindung zum Satelliten ist initialisiert worden.
Nachbedingungen:	• Die Reaktion auf die manipulierten Steuersignale ist im Simulator oder in den Systemlogs sichtbar.
Fachlicher Auslöser:	Der Anwender hat den Befehl zur Steuersignalmanipulation gegeben.
Ablauf:	• Die Fehlerinjektion erhält den Befehl zur Manipulation von Steuersignalen vom Anwender. • Die Fehlerinjektion manipuliert die Steuersignale. • Die manipulierten Steuersignale werden an den Simulator weitergeleitet. • Die Reaktion ist in dem Simulator sichtbar.
Alternativer Ablauf 1:	Anstatt den Befehl zu verändern, wird der Befehl nicht an den Simulator weitergeleitet.
Diagram:	siehe Abbildung 4.6

Kennzeichnung:	**UC028**
Bezeichnung:	**Satellitenzustand manipulieren**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, Satellit, FI-ZedBoard
Vorbedingungen:	• Die Verbindung zu den Steuersignalen des Satelliten ist initialisiert worden. • Die Verbindung zur Fehlerinjektion ist initialisiert worden. • Die Verbindung zum Satelliten ist initialisiert worden.
Nachbedingungen:	• Die Reaktion auf die manipulierten Satellitenzustand ist im Simulator oder in den Systemlogs sichtbar.
Fachlicher Auslöser:	Der Anwender möchte Fehlverhalten innerhalb des Satelliten erzeugen, um Strahlung, Hitze, Alterung von Komponenten und ähnliche Einflüsse zu simulieren. Dazu sendet er einen Befehl zur Manipulation des Satellitenzustandes.
Ablauf:	• Die Fehlerinjektion erhält den Befehl zur Manipulation des Satellitenzustandes vom Anwender. • Die Fehlerinjektion manipuliert den Satelitenzustand. • Der manipulierte Satellitenzustand wird an den Satelliten weitergeleitet. • Die Reaktion ist in der Fehlerinjektion sichtbar.
Diagram:	siehe Abbildung 4.7

Kennzeichnung:	**UC029**
Bezeichnung:	**Fehlerexperimente erzeugen**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, FI-ZedBoard, Simulator
Vorbedingungen:	• Die Verbindung zu der Fehlerinjektion ist initialisiert worden.
Nachbedingungen:	• Die Fehlerinjektion hat das Fehlerexperiment gespeichert.
Fachlicher Auslöser:	Der Anwender möchte mehrere Fehler gleichzeitig oder sequenziell injizieren. Bei Fehlerexperimenten handelt es sich immer um eine Teilmenge aller verfügbaren Fehler im Fehlerkatalog. Der Anwender erstellt diese Fehlerexperimente und übergibt diesen der Fehlerinjektion.
Ablauf:	• Die Fehlerinjektion erhält das Fehlerexperiment des Anwenders. • Die Fehlerinjektion speichert das Fehlerexperiment ab.
Diagram:	siehe Abbildung 4.8

Kennzeichnung:	**UC030**
Bezeichnung:	**Fehlerkatalog erstellen**
Hauptakteur, Akteur:	Fehlerinjektion, FI-ZedBoard, Anwender
Vorbedingungen:	• Die Verbindung zum Satelliten ist initialisiert worden. • Die Fehlerinjektion kennt noch nicht den Fehlerkatalog.
Nachbedingungen:	• Die Fehlerinjektion besitzt den Fehlerkatalog.
Fachlicher Auslöser:	Das System wird initialisiert.
Ablauf:	• Die FI-Komponente auf dem ZedBoard sendet Informationen über Systemkomponenten und möglichen Fehlerinjektionen über das Protokoll. • Die Fehlerinjektion speichert den Fehlerkatalog ab.

Kennzeichnung:	**UC031**
Bezeichnung:	**Fehlerkatalog ausgeben**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, FI-ZedBoard
Vorbedingungen:	• Das System ist initialisiert. • Der Anwender hat die Fehlerinjektion um den Fehlerkatalog angefragt.
Nachbedingungen:	• Die Fehlerinjektion gibt den Fehlerkatalog aus.
Fachlicher Auslöser:	Fehlerkatalog-Anfrage des Anwenders.
Ablauf:	• Die Fehlerinjektion erhält die Anfrage des Anwenders. • Die Fehlerinjektion gibt den Fehlerkatalog aus.
Diagram:	siehe Abbildung 4.9

Kennzeichnung:	**UC032**
Bezeichnung:	**Systemzustand laden**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, FI-ZedBoard
Vorbedingungen:	• Die Fehlerinjektion ist bereit Befehle zu erhalten. • Die Kommunikation zwischen Simulator, Fehlerinjektion und FI-ZedBoard ist initialisiert worden. • Der Simulator hat einen Spielstand geladen.
Nachbedingungen:	• Das System ist im hergestellten Systemzustand und der Anwender hat Rückmeldung über den erreichten Zustand bekommen.
Fachlicher Auslöser:	Der Anwender hat den Befehl zum Laden eines Systemzustand mit zugehörigem Spielstand gegeben.
Ablauf:	• Die Fehlerinjektion erhält den Befehl des Anwenders. • Die Fehlerinjektion gibt den Befehl an das FI-ZedBoard, auf dem der gewünschte Zustand hergestellt wird. • Das FI-ZedBoard gibt Rückmeldung über den erreichten Zustand an die Fehlerinjektion, die diese an den Anwender weitergibt.
Diagram:	siehe Abbildung 4.10

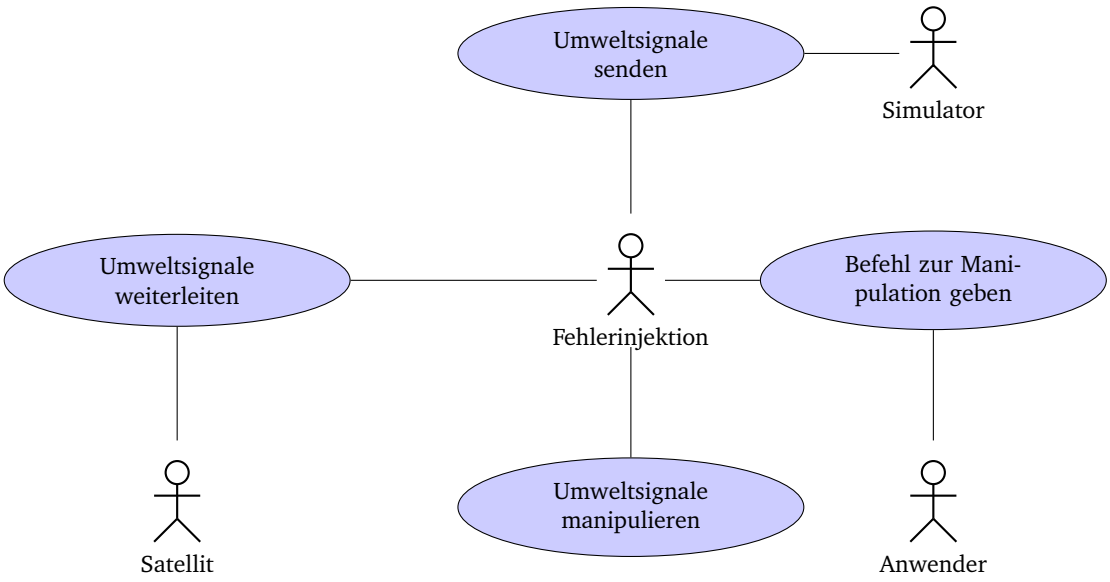


Abbildung 4.5.: Anwendungsfalldiagramm: Umweltsignale manipulieren

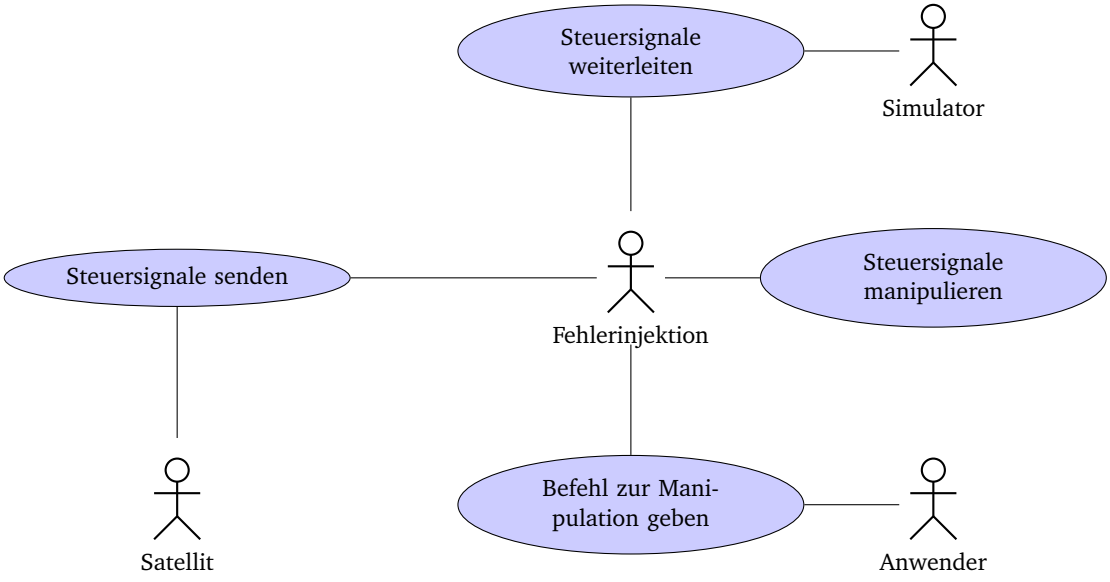


Abbildung 4.6.: Anwendungsfalldiagramm: Steuersignale manipulieren

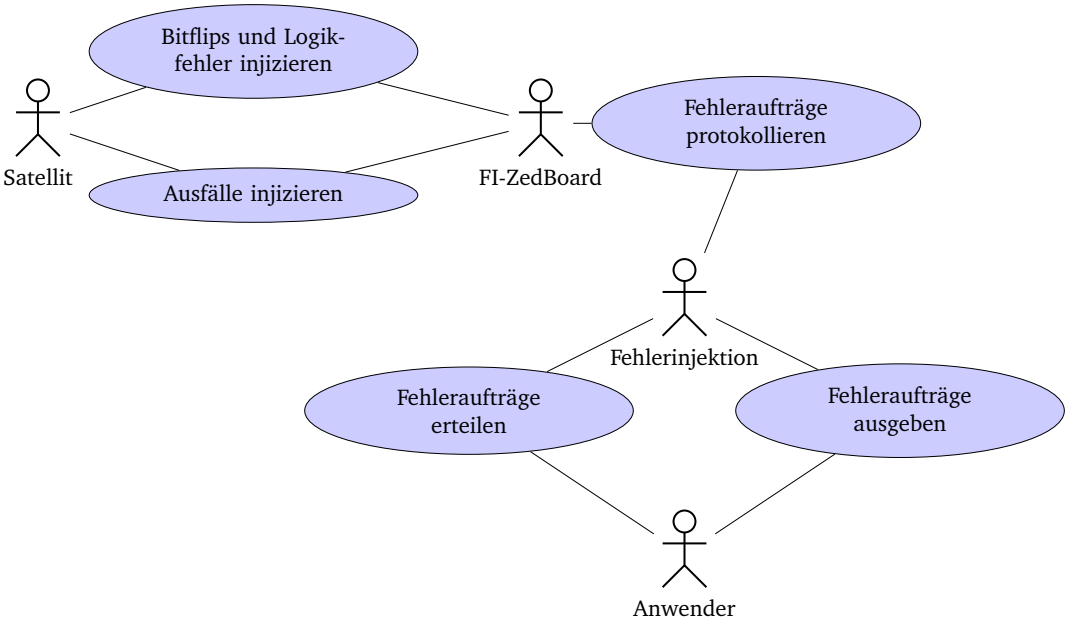


Abbildung 4.7.: Anwendungsfalldiagramm: Satellitenzustand manipulieren

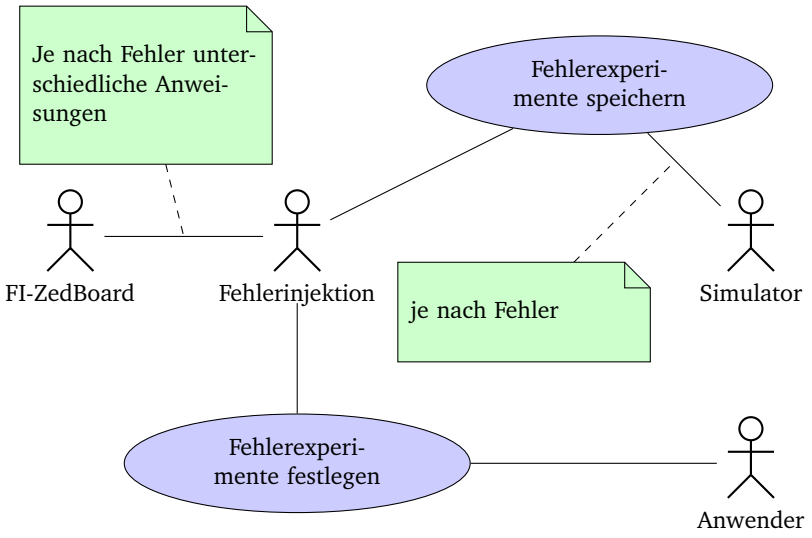


Abbildung 4.8.: Anwendungsfalldiagramm: Fehlerexperimente erzeugen

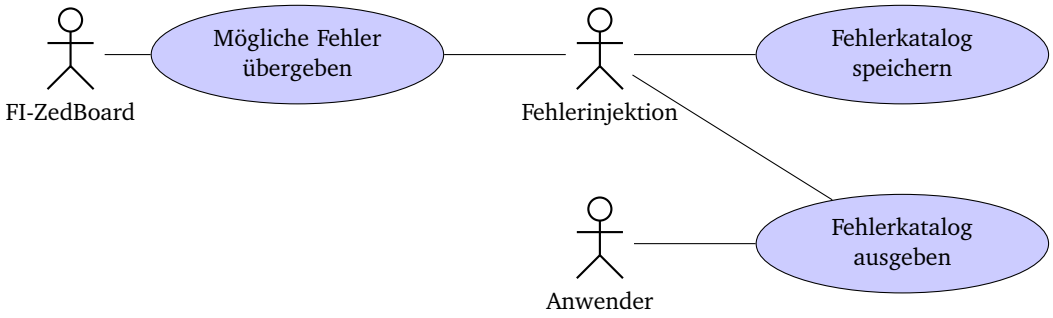


Abbildung 4.9.: Anwendungsfalldiagramm: Fehlerkatalog ausgeben

ZedBoard

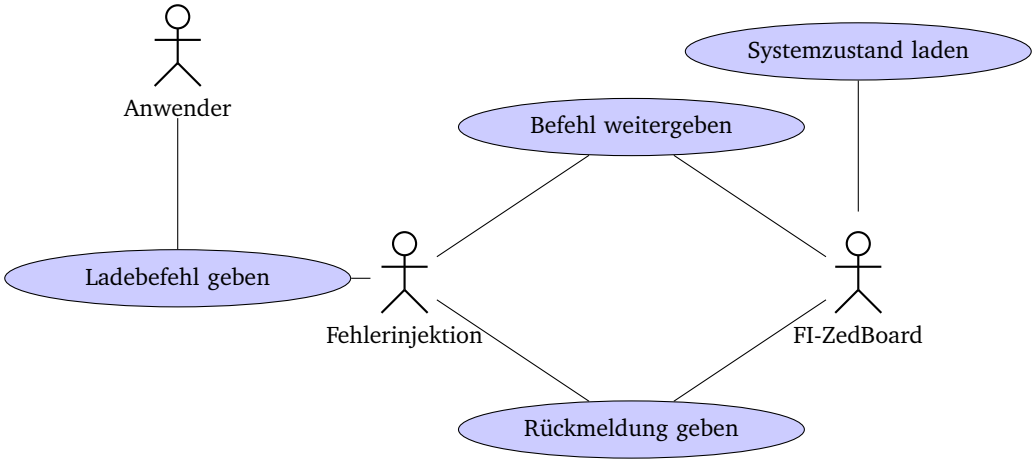


Abbildung 4.10.: Anwendungsfalldiagramm: Systemzustand laden

Kennzeichnung:	**UC033**
Bezeichnung:	**Monitoring der Fehlerinjektion**
Hauptakteur, Akteur:	Anwender, Fehlerinjektion, Satellit, Simulator
Vorbedingungen:	• Die Kommunikation zwischen Simulator, Fehlerinjektion und Satellit sind initialisiert worden. • Der Satellit führt ein Log-Protokoll.
Nachbedingungen:	• Die Fehlerinjektion speichert das Log-Protokoll und kann es anzeigen.
Fachlicher Auslöser:	Der Satellit sendet einen Logeintrag.
Ablauf:	• Der Satellit trägt Aktivitäten ins Log-Protokoll ein. • Der Log-Eintrag wird an den Simulator kommuniziert. • Die Fehlerinjektion liest Kommunikation mit und speichert das Log-Protokoll. • Auf Befehl des Anwenders wird das Log-Protokoll angezeigt.
Diagram:	siehe Abbildung 4.11

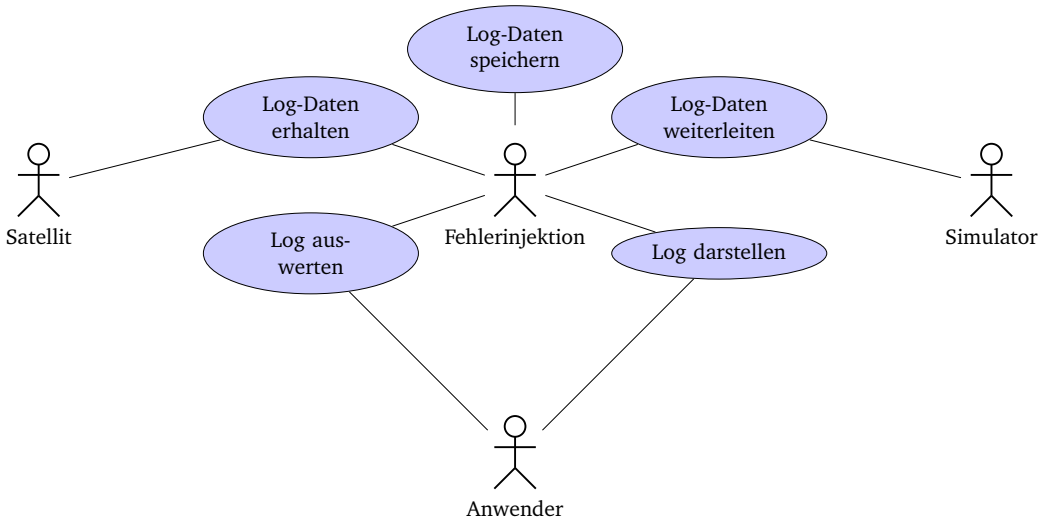


Abbildung 4.11.: Anwendungsfalldiagramm: Monitoring der Fehlerinjektion

5. Anforderungen

Dieses Kapitel gibt einen Überblick über die funktionalen, als auch die nichtfunktionalen Anforderungen des Systems, die aus den Anwendungsfällen abgeleitet wurde. Wie die Anwendungsfälle, ist auch dieses Kapitel in die drei Teilsysteme Fehlerinjektion, Satellitensteuerung und Simulator aufgeteilt.

5.1. Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die gewünschten Funktionalitäten des Systems. Diese sind aufgeteilt in die funktionalen Anforderungen der Fehlerinjektion, des Satelliten und des Simulators.

5.1.1. Simulator

Im Folgenden werden die funktionalen Anforderungen an den Simulator und an die Erweiterung des Simulators (Tabelle 5.1) aufgeführt. Die Erweiterung stellt die Schnittstelle zur Simulation dar. Dazu muss ein Kommunikationsprotokoll von beiden Seiten implementiert werden, was bedeutet, dass hier auch Anforderungen an das Protokoll gelistet werden, welche letztendlich von Satellitensteuerung und Simulator erfüllt werden müssen.

Tabelle 5.1.: Funktionale Anforderungen an den Simulator

Bez.	**Anforderung**	**Ref.**
FA001	Die Mission soll einen zeitlich in Phasen strukturierten Ablauf haben	T0030
FA002	Die Mission muss so ausgelegt sein, dass die Steuerung durch einen Satelliten mit einer Fehlerinjektion aufgezeigt werden kann.	T0030
FA003	Komponenten müssen über das Protokoll identifizierbar sein.	T0031
FA004	Die Komponenten des Satelliten, wie Antriebseinheiten und Sensoren, müssen in der Simulation einzeln über das Protokoll adressierbar und ansteuerbar sein.	T0031, UC001
FA005	Die Zustände aller im Simulator befindlichen Komponenten, die zur Durchführung der Mission erforderlich sind, müssen über das Protokoll auslesbar sein.	T0031 , UC001

Tabelle 5.1.: Funktionale Anforderungen an den Simulator

Bez.	Anforderung	Ref.
FA006	Um die Erweiterbarkeit zu gewährleisten soll jede verwendete Baugruppe in einer eigenen Klasse gekapselt werden.	T0032, UC002, UC003, UC004, UC005, UC006, UC007, UC008, UC009, UC010
FA007	Die Zustände aller im Simulator befindlichen Komponenten, die zur Durchführung der Mission erfordlich sind, müssen durch die Satellitensteuerung abgerufen werden können.	T0032, UC001, UC002, UC003, UC004, UC005, UC006, UC007, UC008, UC009, UC010
FA008	Einzelne Komponenten auf dem Satelliten müssen ein und ausschaltbar sein.	T0032, T0026, UC001

Tabelle 5.1.: Funktionale Anforderungen an den Simulator

Bez.	Anforderung	Ref.
FA009	Die Simulation soll pausiert und fortgesetzt werden können.	T0033, UC012 , UC013
FA010	Die Reaktionszeit der Mod muss die zeitlichen Vorgaben der Satellitensteuerung einhalten.	T0033, UC011

5.1.2. Satellit

Im Folgenden werden die funktionalen Anforderungen an die Satellitensteuerung (siehe Tabelle 5.2) aufgeführt.

Tabelle 5.2.: Funktionale Anforderungen an die Satellitensteuerung

Bez.	Anforderung	Ref.
FA011	Die Satellitensteuerung muss seine aktuelle Geschwindigkeit bestimmen und ändern können.	UC014, T0023
FA012	Die Satellitensteuerung muss in der Lage sein Steuerwerte an die Triebwerke weiterzugeben.	UC014
FA013	Die Satellitensteuerung muss seine aktuelle Rotation bestimmen und ändern können.	UC015
FA014	Die Satellitensteuerung muss in der Lage sein Steuerwerte an die Rotationsaktoren weiterzugeben.	UC015
FA015	Die Satellitensteuerung muss in der Lage sein seine Position auf der aktuellen Orbitbahn zu ermitteln.	UC016
FA016	Die Satellitensteuerung muss in der Lage sein die benötigten Geschwindigkeitsänderungen und Orientierungsänderungen zur Anpassung der Orbitbahn zu berechnen.	UC016, T0023
FA017	Die Satellitensteuerung muss in der Lage sein die nötigen Orbitanpassungen zum Ausführen der Mission zu berechnen.	UC017, T0023
FA018	Die Satellitensteuerung muss über ein Adaptionsverfahren verfügen.	UC018, T0029
FA019	Der Satellit muss eine Kommunikationsschnittstelle zur Bodenstation besitzen.	UC019, T0024
FA020	Die Bodenstation muss in der Lage sein die autonome Steuerung des Satelliten abzuschalten.	UC019, T0024
FA021	Die Satellitensteuerung kennt die Orientierung der verbauten Instrumente im Vergleich zum Satelliten.	UC020
FA022	Die Satellitensteuerung muss mit den einzelnen Satellitenkomponenten kommunizieren können.	T0026
FA023	Die Satellitensteuerung muss die einzelnen Satellitenkomponenten starten und stoppen können.	T0026

Tabelle 5.2.: Funktionale Anforderungen an die Satellitensteuerung

Bez.	Anforderung	Ref.
FA024	Die Satellitensteuerung muss die Energieverbräuche der einzelnen Satellitenkomponenten dokumentieren können.	T0026
FA025	Die Satellitensteuerung weiß wie lange eine Satellitenkomponente benötigt wird.	T0026
FA026	Die Satellitensteuerung weiß wieviel Energie eine Satellitenkomponente zum Ausführen ihrer Tätigkeit braucht.	T0026
FA027	Die Satellitensteuerung weiß wie oft eine bestimmte Satellitenkomponente noch benötigt wird.	T0026
FA028	Die Satellitensteuerung muss die gesammelten Daten kommunizieren können.	T0024
FA029	Die Satellitensteuerung muss über den Energiestand Bescheid wissen	T0026

5.1.3. Fehlerinjektion

Im Folgenden werden die funktionalen Anforderungen an die Fehlerinjektion (siehe Tabelle 5.3) aufgeführt.

Tabelle 5.3.: Funktionale Anforderungen an die Fehlerinjektion

Bez.	Anforderung	Ref.
FA030	Umweltsignale die von der Simulation an die Satellitensteuerung gesendet werden, müssen manipuliert werden können.	UC026, T0016
FA031	Zur Manipulation von simulierten Komponenten, wie Triebwerken, müssen die gesendeten Steuerungsbefehle von der Satellitensteuerung an die Simulation manipuliert werden können.	UC027, T0016
FA032	Fehler zur Manipulation des Satellitenzustands müssen von dem Benutzer an die Satellitensteuerung gesendet werden.	UC028, T0016
FA033	Fehler zur Manipulation des Satellitenzustands müssen von der Satellitensteuerung empfangen werden.	UC028, T0016
FA034	Empfangene Fehler zur Manipulation des Satellitenzustands müssen injiziert werden.	UC028, T0016
FA035	Es müssen Fehler in einzelne Satellitenkomponenten injiziert werden können.	UC028, T0016
FA036	Die Satellitenkomponenten müssen der Fehlerinjektion bekannt sein.	UC028, T0016
FA037	Informationen über die Satellitenkomponenten müssen vorhanden sein.	UC028, T0017
FA038	Die Kommunikation zwischen dem Simulator und der Satellitensteuerung muss zwischengespeichert oder abgefangen werden.	T0018
FA039	Es muss ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellit bestehen.	UC028, T0018

Tabelle 5.3.: Funktionale Anforderungen an die Fehlerinjektion

Bez.	Anforderung	Ref.
FA040	Ein an die Fehlerinjektionskomponente auf dem Host-PC zu sendenden Satellitenlog soll existieren.	UC033, T0019
FA041	Es kann eine Systemzustandsüberwachung der Satellitensteuerung für den Fehlerinjektionszugriff erstellt werden.	T0019
FA042	Die Maßnahmen der Satellitensteuerung sollen dem Anwender zur Verfügung gestellt werden.	UC033, T0019
FA043	Es soll ein Timestamp zu jedem Log-Eintrag existieren.	UC033, T0019
FA044	Eine Speichereinheit aller injizierten Fehler muss vorhanden sein.	T0020
FA045	Auf die Speichereinheit von allen injizierten Fehlern muss zugegriffen werden können.	T0020
FA046	Eine UI zur Anzeige aller injizierten Fehler soll zur Verfügung gestellt werden.	T0020
FA047	Durch die Kommandozeile muss auf die Speichereinheit von allen injizierten Fehlern zugegriffen werden können.	T0020
FA048	Der Fehlerinjektionszugriff muss auf Speicherblöcken oder durch speziell definierte Komponenten geschehen.	T0021
FA049	Eingabebefehle der Fehlerinjektion müssen dem Anwender zur Verfügung gestellt werden.	T0022
FA050	Eine GUI zur Steuerung der Fehlerinjektionen kann zur Verfügung gestellt werden.	T0022

5.2. Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen beschreiben die gewünschten Anforderungen an die Qualität des Systems. Diese sind aufgeteilt in die nichtfunktionalen Anforderungen für den Simulator und die Fehlerinjektion.

5.2.1. Simulator

Im Folgenden werden die nichtfunktionalen Anforderungen an den Simulator **??** (siehe Tabelle 5.4) aufgeführt.

Tabelle 5.4.: Nichtfunktionale Anforderungen an den Simulator

Bez.	Anforderung	Ref.
NA001	Das Protokoll muss in der Lage sein, alle Datenformate der Simulation zu übertragen.	T0031
NA002	Das Protokoll muss sowohl vom Zedboard als auch von der Fehlerinjektion verstanden und verwendet werden können.	T0031

Tabelle 5.4.: Nichtfunktionale Anforderungen an den Simulator

Bez.	Anforderung	Ref.
NA003	Das Protokoll muss modular erweiterbar sein.	T0013
NA004	Das Medium des Protokolls muss Ethernet sein.	T0014
NA005	Das zu erarbeitende Protokoll muss sich daran orientieren, wie die Komponenten eines realen Satelliten in der Regel angesteuert werden.	T0031, T0025
NA006	Das Protokoll soll mittels Hashfunktionen gegen Kommunikationsstörungen abgesichert werden.	T0031
NA007	Das vom Protokoll verwendete Signal Encoding soll energiespaarend gewählt werden.	T0031
NA008	Das Protokollmodell soll das OSI-Schichtenmodell auf einer definierten Ebene erweitern.	T0031

5.2.2. Fehlerinjektion

Im Folgenden werden die nichtfunktionalen Anforderungen an die Fehlerinjektion (siehe Tabelle 5.5) aufgeführt.

Tabelle 5.5.: Nichtfunktionale Anforderungen an die Fehlerinjektion

Bez.	Anforderung	Ref.
NA009	Die Nutzerschnittstelle der Fehlerinjektion soll einfach bedienbar sein.	T0016
NA010	Es sollen Fehler in unter einer Sekunde injiziert werden können.	T0016
NA011	Es soll möglichst wenig Platz auf dem FPGA gebraucht werden.	T0016
NA012	Die Satellitenkomponenten müssen auf mögliche Fehlerquellen analysiert werden.	T0017
NA013	Die Auswahl an injizierbaren Fehlern muss definiert werden.	T0017
NA014	Die Kommunikation zwischen der Fehlerinjektion und dem Simulator und der Satellitensteuerung soll nicht länger als 1 s Verzögerung verursachen.	T0018
NA015	Die Verzögerung der Kommunikation soll möglichst konstant sein.	T0018
NA016	Der Satellitensystemzustand soll übersichtlich angezeigt werden.	T0019
NA017	Ein Log-Eintrag soll für den Anwender direkt verständlich sein.	T0019
NA018	Die injizierten Fehler sollen übersichtlich angezeigt werden.	T0020

6. Systemkonzept

In diesem Kapitel wird das Systemkonzept vorgestellt. Das System besteht aus den drei Teilsystemen Simulator, Satellitensteuerung und Fehlerinjektion (vgl. Abbildung 6.1). Jedes Teilsystem setzt sich aus Komponenten zusammen (vgl. Abschnitte 14.1,15.1 und 16.1). Die Komponenten setzen sich wiederum aus Modulen zusammen, die in den Abschnitten 14.2, 15.2 und 16.2 näher beschrieben werden.

Im Folgenden wird die Kommunikation der Teilsysteme untereinander näher beleuchtet. Im Anschluss werden die Teilsysteme vorgestellt.

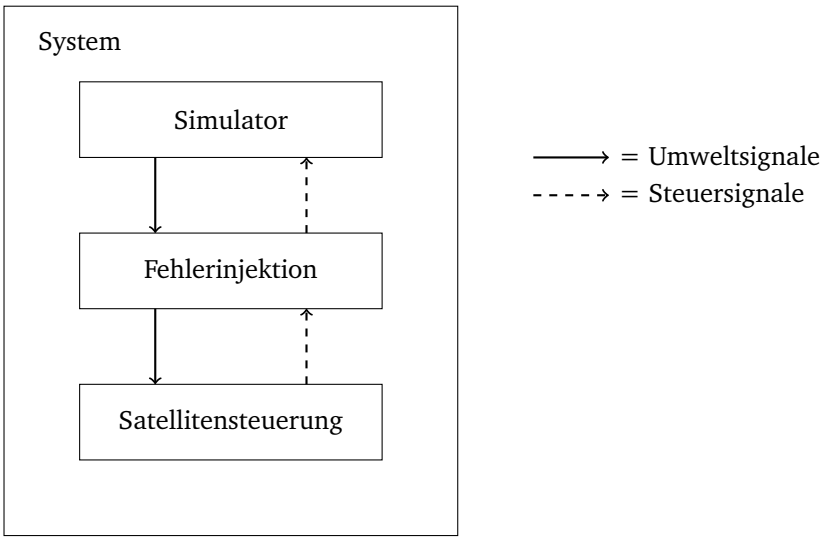


Abbildung 6.1.: Das System besteht aus drei Teilsystemen: Simulator, Fehlerinjektion und Satellitensteuerung.

6.1. Kommunikation

Dieser Abschnitt behandelt die Kommunikation mit den ZedBoards. Dazu werden drei Kommunikationsarten vorgeschlagen, Hub-Kommunikation, Broadcast-Kommunikation und Master-Kommunikation.

Hub-Kommunikation: Bei der Hub-Kommunikation wird der Datenfluss auf dem Host-PC durch einen Hub auf die ZedBoards verteilt (vergleiche Abbildung 6.2). Auf diese Weise muss kein ZedBoard zwischengeschaltet werden, dass sich um die Kommunikation kümmert.

Broadcast-Kommunikation: Bei der Broadcast-Kommunikation werden alle Daten an alle ZedBoards verschickt, sodass diese dann die für sie relevanten Daten selbst herausfiltern müssen. Nachteilig ist hier die hohe Datendichte auf den Kommunikationskanälen.

Master-Kommunikation: Bei dieser Variante wird ein ZedBoard als Master-Board vor die anderen geschaltet (vergleiche Abbildung 6.3). Dieses filtert die Daten und leitet die gefilterten Daten an das entsprechende ZedBoard weiter. Auch hier ist der Nachteil, das wertvolle Ressourcen, wie Rechenzeit des Mikrocontrollers oder Platz auf dem FPGA des Master-Boards gebunden werden.

Im Folgenden werden die Teilsysteme in Einheiten gegliedert. Diese Bezeichnung wurde gewählt, da zu diesem Zeitpunkt noch nicht feststeht, welche Einheiten tatsächlich in Komponenten umgesetzt werden. Die Grundfunktionen sollen zwar übernommen werden, die Gliederung der Komponenten kann sich jedoch Problembedingt ändern.

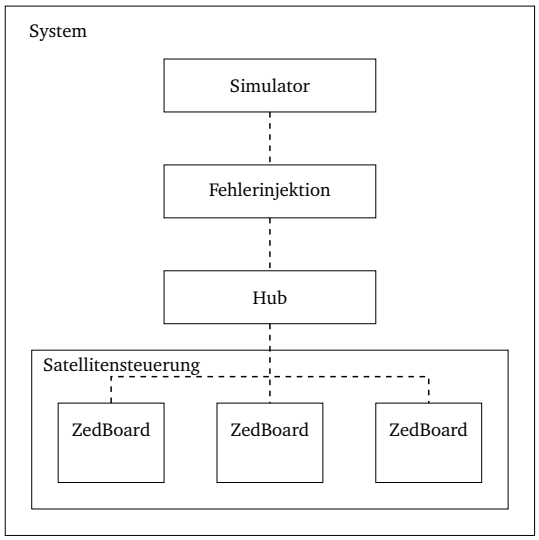


Abbildung 6.2.: Hub-Kommunikation

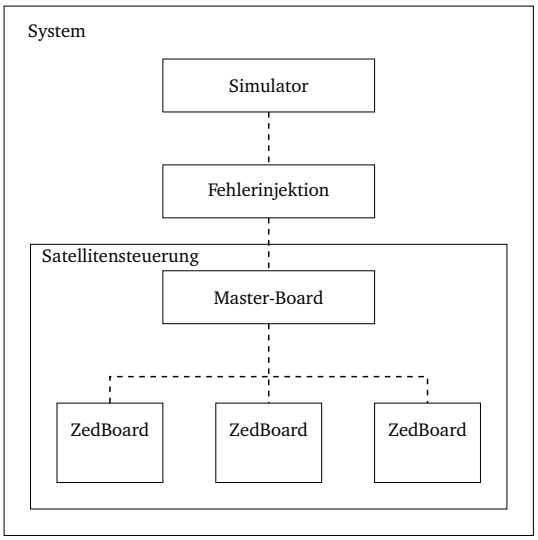


Abbildung 6.3.: Master-Kommunikation

6.2. Simulator

Als Simulationsumgebung wird das Spiel KSP genutzt. Es empfängt die Steuersignale der Satellitensteuerung und generiert Umweltdaten, die der Satellit an die Satellitensteuerung weiterleitet. Der Simulator soll ebenfalls das Log-Protokoll des Satelliten empfangen und ausgeben.

6.3. Satellit

Hier wird das Teilsystem Satellitensteuerung vorgestellt, welches den simulierten Satelliten verwaltet. Es besteht aus fünf Einheiten: Housekeeping, Orbitplanung, Kommunikation, Fehlertoleranz und Fortbewegung.

Fortbewegung: Diese Eiheit übernimmt die Steuerung der Fortbewegung. Das kann beispielsweise die Ausrichtung der Steuerdüsen sein, wenn sie Daten von der Orbitplanung erhält.

Housekeeping: Die Aufgabe dieser Einheit ist die Überwachung des Satelliten und das Ressourcenmanagement. Energiegewinnung gehört zu ihren Aufgaben, aber auch die Verteilung der Energie an andere Komponenten.

Orbitplanung: Die Orbitplanungseinheit berechnet den Kurs, den der Satellit fliegen soll und leitet die entsprechenden Befehle zur Ausrichtung der Triebwerke an die Fortbewegungseinheit weiter.

Kommunikation: Die Kommunikationseinheit hat die Aufgabe, die Kommunikation mit der Simulation zu organisieren. Sie übernimmt außerdem das Steuern der Aufgaben, die nicht durch andere Einheiten übernommen werden.

Fehlertoleranz: Diese Einheit überwacht ebenfalls den Satelliten, aber nicht mit dem Ziel Ressourcen zu verwalten, sondern um auftretende Fehler zu detektieren und Gegenmaßnahmen einzuleiten. Dies kann zum Beispiel durch den Vergleich der Ausgaben redundanter Einheiten geschehen.

6.4. Fehlerinjektion

Das Teilsystem Fehlerinjektion besteht aus zwei Einheiten. Eine befindet sich auf einem Host-PC, über welchen sich Fehler injizieren lassen. Die zweite Einheit soll sich je auf einem ZedBoard befinden.

Die Einheit zur Fehlerinjektion auf dem Host-PC ähnelt der aus [24]. Der Anwender kann über eine Steuerungseinheit Fehler injizieren und Injektionsdaten, sowie Daten über den Satellitenzustand abfragen. Zur Ausgabe des Satellitenzustandes wird eine Überwachungseinheit des Satellitenlogs benötigt. Diese Aufgabe übernimmt der *Monitor*. Weiterhin ist die Einheit, welche sich zwischen dem Simulator und der Satellitensteuerung befindet, dafür zuständig, Fehler in Umwelt- bzw. Steuersignale zu injizieren. Die Einheit auf dem ZedBoard ist für die Injektion von Fehlern auf der Hardware zuständig. Um solche Aufgaben wahrnehmen zu können, werden unter Umständen Unterbrechungsroutinen auf dem Mikrocontroller benötigt.

Um Fehler gezielt in die Hardware zu injezieren ist es wichtig, dass der Fehlerinjektion die Einheiten der Satellitensteuerung bekannt sind. Es werden zwei Ansätze vorgeschlagen, dieses Wissen zu erlangen.

Statistischer Ansatz: Bei diesem Ansatz sind der Fehlerinjektion die Einheiten der Satellitensteuerung bereits bekannt. Die Kommunikation ist problemlos möglich.

Variabler Ansatz: Die Fehlerinjektion kennt zu Beginn keine Einheiten der Satellitensteuerung. Zum Kommunikationsbeginn werden alle notwendigen Daten über die Einheiten und Datenflüsse an die Fehlerinjektion übergeben.

7. Entwicklungsprozess

Im Folgenden wird die Vorgehensweise bei der Entwicklung behandelt, begonnen bei den Entwicklungsabläufen der einzelnen Systembestandteile sowie den dazu nötigen Tools. Der Entwicklungsprozess wird das Projekt bis zum Ende begleiten und wurde daher so präzise wie möglich geplant (Siehe Teil II). Trotzdem kann es im Verlauf der Entwicklung notwendig werden, Anpassungen vorzunehmen, um auf unerwartete Probleme oder Inkompatibilitäten zu reagieren. Daher sind vor allem die in Abschnitt 7.1 genannten Tools als geeignete Vorauswahl zu sehen, welche bei Bedarf ersetzt werden können.

7.1. Vorgehen und Tools

Dem Systemkonzept in Abschnitt **??** folgend, besteht das System aus drei Teilsystemen. Da jede dieser Teilsysteme in einer speziellen Umgebung arbeiten soll, gibt es ebenso spezielle Entwicklungsvorgehen und Tools. In den nächsten drei Abschnitten werden daher diese Entwicklungsvorgehen behandelt.

7.1.1. Entwicklung einer Satellitensteuerung

Die Satellitensteuerung soll einen (teil-)autonomen Satelliten steuern, welcher aus den Umweltsignalen des Simulators Steuerbefehle errechnet und diese zurück an den Simulator überträgt oder gegebenenfalls Befehle von der Bodenstation zur Steuerung ausführt und enstprechende Steuerbefehle an den Simulator schickt. Zudem soll die Satellitensteuerung die für die Mission benötigte Funktionalität des Satelliten garantieren. Die Satellitensteuerung wird dabei auf einem ZedBoard von Xilinx laufen, bestehend aus Mikropozessor und FPGA. Um eine abstrahierte Entwicklung von diesen Bestandteilen zu ermöglichen, soll diese auf Systemebene entworfen werden (vgl. [21, Kapitel 3, insb. Abschnitt 3.3]. Dabei wird zuerst ein Komplettsystem entwickelt und in weiteren Schritten detaillierter beschrieben. Im Laufe dieses Prozesses findet die Zuteilung (*partitioning*) statt, welche Bestandteile auf welche Hardwarekomponenten übertragen werden.

Als Grundlage der Satellitensteuerung sollen grafische Modelle erstellt werden, welche Verhalten und Abhängigkeiten der Bestandteile des Satelliten zeigen. Diese werden anschließend in C++ mit Hilfe von SystemC [1, 11] umgesetzt und verfeinert (in Abbildung 7.1 *Refinement* genannt). Im Schritt *Transaction Level Modelling (TLM)* werden aus den Verfeinerungen TLMs generiert. Die Komponenten dieser Modelle werden auf Hardware- und Softwareprozesse abgebildet, getestet und möglicherweise neu partitioniert. Der erstellte Code wird anschließend in zwei Arten weiterverarbeitet: Zum Einen wird der Teil des Codes, welcher später auf der programmierbaren Logik laufen soll, in Xilinx Vivado HLS (vgl. Abschnitt 7.1.1) exportiert, das aus dem SystemC-Code VHDL-Code erzeugt und aus diesem abschließend ein Abbild zur Bespielung des ZedBoards generiert. Der andere Teil des Codes wird in der Xilinx SDK weiterverarbeitet, um auf den Mikroprozessoren ausgeführt zu werden. In Abbildung 7.1 ist der Ablauf der Entwicklung grafisch dargestellt.

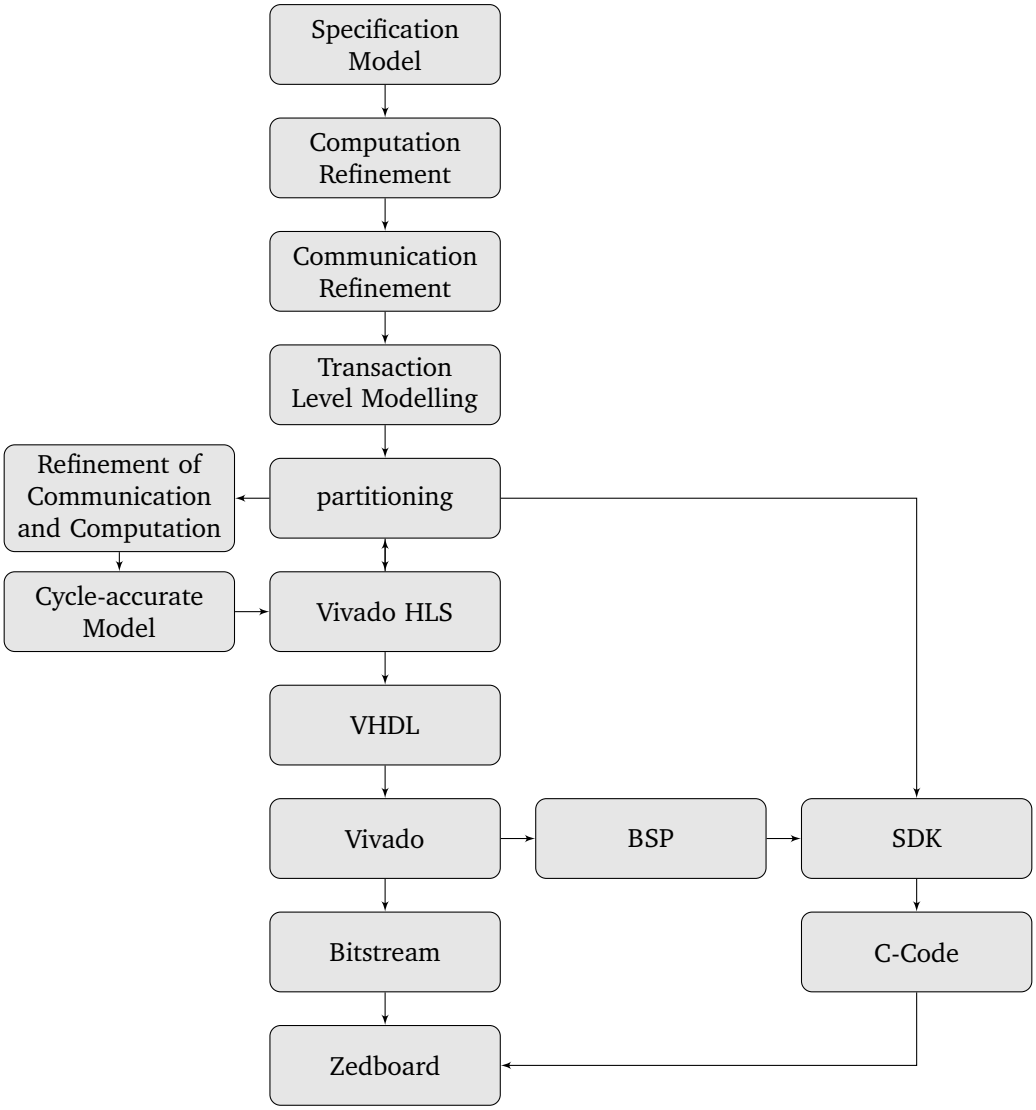


Abbildung 7.1.: Die Entwicklung des Satelliten ist als Fluss vom Spezifikationsmodell bis zum Zedboard aufgebaut. Im Fluss werden die einzelnen Zwischenergebnisse sowie die verwendeten Tools ersichtlich.

Xilinx Toolchain

Xilinx ist führender Entwickler und Hersteller von FPGAs und bietet für die Arbeit mit der Hardware die Xilinx Vivado Design Suite an. Darin enthalten sind eine Reihe von Werkzeugen, welche sich einzeln oder ergänzend einsetzen lassen, um Hardwaredesigns zu entwickeln, zu simulieren und zu testen oder verifizieren. Im Folgenden werden die Werkzeuge vorgestellt, welche im Rahmen der Entwicklung (gemäß Abbildung 7.1) verwendet werden sollen. Vivado wird der Projektgruppe in der, zum Zeitpunkt des Projektbeginns, neusten Version 2016.3 zur Verfügung gestellt.

Xilinx Vivavo HLS: Mit Xilinx Vivado HLS bietet die Xilinx Toolchain Funktionen zur High-Level Synthese. Mit diesem Werkzeug können IP-Cores entwickelt werden, welche anschließend zu Vivado exportiert und in RTL-Projekte eingebunden werden. Vivado HLS

unterstützt die Entwicklung in C oder C++ mit SystemC und stellt eine große Bibliothek zur Komponentenentwicklung zur Verfügung.

Xilinx Vivado: Das Herzstück der Xilinx Toolchain ist Xilinx Vivado. Es enthält den Vivado Integrator, den Vivado Simulator und Funktionen zur Synthese, Implementierung und Programmierung der Hardware. Zudem stellt Vivado ein GUI zur Entwicklung von VHDL-Code zur Verfügung. In Vivado gibt es mehrere Arten von Projekten, davon sind mindestens zwei für die Projektgruppe interessant: Zum einen Register Transfer Level (RTL)-Projekte, in welchen Intelectual Property (IP)-Cores mit dem eingebetteten System verbunden werden und der Bitstream für die Hardware erstellt wird, zum anderen Projekte für die IP-Cores selbst.

Xilinx Vivado Integrator: Der Vivado Integrator ist Teil von Vivado und wird in RTL-Projekten verwendet, um sogenannte Block-Designs zu erstellen. Block-Designs sind Blockschaltbilder, in welchen einzelne Komponenten des Projektes miteinander verbunden werden. Dabei handelt es sich in der Regel um einen Baustein für die Hardware (Processing System), einen Verbindungsbaustein (zum Beispiel ein Advanced Microcontroller Bus Architecture (AMBA)-Interconnect) und beliebigen IP-Cores. Diese Komponenten werden mittels Signalen miteinander verbunden oder mit eben diesen mit externen Pins auf der Hardware verknüpft. IP-Cores können dabei entweder selbst erstellt werden (hierzu wird ein neues Vivado-Projekt angelegt, welches die notwendige Advanced eXtensible Interface (AXI) zum AMBA besitzt und anschließend unabhängig von dem RTL-Projekt entwickelt werden kann) oder importiert werden. Dabei können natürlich bereits selbst erstellte IP-Cores verwendet werden, oder auch fremde IP-Cores importiert werden. Diese müssen dabei nicht als Quelltext vorliegen, sondern dürfen eine Blackbox sein, welche lediglich ihre Signale nach außen bekannt gibt.

Xilinx Vivado Simulator: Mit Vivado besitzt die Xilinx Toolchain, im Gegensatz zu dem Vorgänger Xilinx ISE, einen eigenen Simulator. Dieser ist in Vivado eingebunden und kann zur taktgenauen Simulation der RTL-Projekte, aber auch einzelner Komponenten wie IP-Cores verwendet werden. Hierzu werden Testbenches geschrieben, welche die die Ausführumgebung der Simulation vorgeben. Dabei können auch Wartezeiten angegeben werden, um einen Takt zu erzeugen und Eingaben zu simulieren. Während der Simulation können sämtliche Signale als Waveform angezeigt werden.

Xilinx SDK: Basierend auf Eclipse 4.5 bietet das Xilinx SDK eine in die Toolchain integrierte Möglichkeit, Programme für die Mikroprozessoren zu entwickeln. Das Xilinx SDK kann aus Vivado heraus gestartet werden und das zuvor erstellte Board Support Package in ein Projekt einbinden. Dieses Projekt kann entweder ein simples C- oder C++-Projekt sein, aber auch auf *FreeRTOS* oder *PentaLinux* basieren. Anschließend kann über das SDK das FPGA-Board programmiert und das Programm darauf ausgeführt werden.

7.1.2. Entwicklung des Fehlerinjektionssystems

Die Programmiersprache für die Fehlerinjektionsschnittstelle ist nicht vorgegeben und steht nicht in Abhängigkeit zum Simulator und zu dem ZedBoard. Die Schnittstelle zum ZedBoard

wird durch eine Ethernetverbindung und die Kommunikation zum Simulator durch ein Protokoll realisiert. Die Fehlerinjektionsschnittstelle muss also fähig sein, den Datenfluss zwischen dem Simulator und dem ZedBoard manipulieren zu können. Dies schränkt die Auswahl an Programmiersprachen nicht ausschlaggebend ein. Es wäre demnach sinnvoll eine Programmiersprache zu benutzen, die schon im Projekt benutzt wird oder eine Programmiersprache, die sich am besten für eine Implementierung der Fehlerinjektionsschnittstelle eignet.

Da die SystemC-Simulation auf C++ aufbaut, erscheint eine Implementierung der Fehlerinjektionsschnittstelle in C++ sinnvoll. Für die Komponenten der Fehlerinjektion auf dem ZedBoard wird die gleiche Entwicklung und Toolchain wie die der Satellitensteuerung gewählt.

7.1.3. Entwicklung der Simulatorerweiterung

Die Kommunikation des Host-PCs mit dem Simulator wird über das API von KSP hergestellt. Das API liegt für die Programmiersprache C# vor, daher wird eine Integrated Development Environment (IDE) für C# benötigt. In der Community sind MonoDevelop und Visual Studio weit verbreitet. Während Visual Studio kommerziell von Microsoft vertrieben wird, ist MonoDevelop frei erhältlich. In beiden Entwicklungsumgebungen kann die Dokumentation [4] der API integriert werden. Da die Dokumentation von der aktiven Community [4] gepflegt wird, ist diese jedoch nicht vollständig. Ein weiterer Vorteil von MonoDevelop ist die Kompatibilität unter Windows, Linux und Mac OS X.

Um den Entwicklungsaufwand der Kommunikation des Simulators zu verringern, können bereits bestehende Mods für KSP eingesetzt werden. Insbesondere bieten sich dafür kRPC [44] und Telemachus [78] an, die als Grundlage für die Kommunikation dienen können. kRPC ermöglicht das Auslesen von Statusinformationen über das Raumfahrzeug, sowie das Setzen von Kontrolldaten über eine TCP/IP-Netzwerkschnittstelle. Telemachus hingegen implementiert eine Antenne als Bauteil in KSP [78]. Durch Verwendung der Antenne an einem Raumfahrzeug lassen sich Informationen des Raumfahrzeugs über eine HTTP-Schnittstelle auslesen. Die gleiche Schnittstelle ermöglicht es auch Steuersignale an KSP zu senden [79].

Sollte es notwendig sein, neue Bauteile in KSP einzufügen, empfiehlt sich eine 3D - Grafiksoftware, wie das frei erhältliche Blender [67] zu verwenden. Außerdem wird für das Einbinden ins Spiel die Grafikengine Unity [67] benötigt.

Teil II.

Projektmanagement

8. Projektmanagementkonzept

Zur Projektumsetzung musste ein Konzept erarbeitet werden. Dabei galt es, verschiedene Vorgehensmodelle gegeneinander abzuwägen, aber auch eine Sinnvolle Projektaufteilung zu finden. Da sich das Projekt in drei Komponenten gliedert, wurde schnell eine Aufteilung der Projektgruppe in drei Teilprojekte beschlossen. Jedes Teilprojekt hat die Verantwortung für ein Teilsystem, sodass das (Gesamt-)System mit Abschluss der Arbeiten an den Teilsystemen fertiggestellt wird (vgl. Abschnitt 9).

Das Projektmanagement erfolgt über das Open Source Tool *Redmine* [68]. In Redmine wird die Projektstruktur angelegt, sodass jedes entstehende Arbeitspaket an Mitglieder der Gruppe zugewiesen werden kann. Redmine bietet außerdem nützliche Funktionen wie Benachrichtigungen, Dateiverwaltung, Kalender, Zeiterfassung, Integration von Versionsverwaltungssoftware, ein Rollenkonzept für Mitglieder sowie Unterstützung verschiedenster Datenbanken, ein Wiki und die Möglichkeit der Visualisierung und des Exports des Projektfortschritts.

Die Kommunikation innerhalb der Gruppe erfolgt über E-Mail-Verteiler, aufgeteilt in einen Verteiler für die Betreuer, einen für die Projektgruppe und einen, der alle erreicht.

Als Vorgehensmodell wurde ein modifiziertes V-Modell gewählt. Die Modifikation besteht darin, dass bereits vor Projektabschluss Prototypen benötigt werden, die zur Zwischenpräsentation bestimmte Anforderungen erfüllen sollen. Für das V-Modell sprechen seine hervoragende Eignung für große Projekte, für eingebettete Systeme, sowie sein Fokus auf das Produkt. Dadurch lassen sich, über die Meilensteine, die (Teil-)Projektergebnisse überprüfen und sinnvoll in diese Dokumentation eingliedern.

Als Designmethode hat sich die Projektgruppe für den Ansatz des *SLDs* entschieden. Bei diesem Ansatz wird von der Hardware abstrahiert und zunächst ein abstraktes Systemmodell entwickelt, welches durch iteratives Vorgehen der Hardware angepasst wird. Auf diese Weise kann bereits ein funktionierendes Modell mit der Simulation interagieren, ohne dass die Hardware benötigt wird.

9. Planung

In diesem Kapitel wird die Projektplanung näher erläutert. Es wird sowohl auf den Strukturplan, als auch auf die erarbeitete Phasenplanung und die sich daraus ergebenden Meilensteine eingegangen.

9.1. Strukturplanung

Um das Projekt effektiv erfassen und verwalten zu können, wurde ein Strukturplan angelegt. Dieser Teilt das Projekt in acht Phasen:

Projektvorbereitung: In der Projektvorbereitung hatte die Gruppe die Aufgabe, bereits eine Anforderungsanalyse durchzuführen, da diese für die Anschlussphase, der Proposalausarbeitung benötigt wurde. Es wurden außerdem Rollen verteilt und ein Kick-off-Workshop durchgeführt ([vgl. 9]). Das Ergebnis der Workshops ist der Phasenplan aus Abbildung 9.2 entstanden. Aus den Projektphasen lassen sich auch die Meilensteine aus Tabelle 9.1. Weiterer Bestandteil war das Abhalten einer Seminarphase, um mögliche Wissenslücken der Mitglieder der Projektgruppe zu schließen. Außerdem wurde eine Risikoanalyse durchgeführt und ein Zielkatalog angefertigt.

Proposalausarbeitung: Es standen ca. zwei Wochen für diese zur Verfügung. In dieser Zeit wurde ein Proposal vorbereitet, welches zur Projektpräsentation abgegeben werden musste. Bei der Projektpräsentation konnte die Projektgruppe ihre Vorstellung von der Verwirklichung des Projekts präsentieren, die Gruppe der Betreuer hatte die Möglichkeit, Feedback zu geben, sodass die Projektgruppe aus den Erfahrungen schöpfen konnte und die Möglichkeit bestand, die Möglichkeiten der Umsetzung zu reflektieren.

Feinplanung: In der Feinplanungsphase wurden die Arbeitspakete des in Abbildung 9.1 gezeigten Strukturplans erarbeitet. Parallel dazu wurde eine finale Version des Proposals erstellt, in der das Feedback der Projektpräsentation aufgenommen wurde. Eine weitere Erkenntnis dieser Phase, in die auch Vorüberlegungen eingeflossen sind, war die Aufteilung des Projektes in drei Teilprojekte.

Teilprojekt Simulator: Diese Phase verläuft parallel zu den Phasen der anderen Teilprojekte. Sie ist in sechs Abschnitte aufgeteilt, die jeweils mit einem Teilprojekt-Meilenstein abschließen. Wie die anderen Teilprojekte auch, muss das Teilprojekt Simulator die Gesamtspezifikation, einen Systementwurf und den Modulentwurf erstellen, ihr Teilsystem implementieren und testen. Die Daten der Fertigstellung können Tabelle 9.1 entnommen werden.

Teilprojekt Fehlerinjektion: Analog zum Teilprojekt Simulator.

Teilprojekt Satellitensteuerung: Analog zum Teilprojekt Simulator.

Systemtest: Im Anschluss an die Phasen der Teilprojekte soll das System soweit sein, dass es als Gesamtsystem getestet werden kann. Zu diesem Zeitpunkt müssen die Komponenten- und Modultests abgeschlossen sein, sodass die Teilsysteme im Verbund getestet werden können.

Projektabschluss: In dieser Phase sollen diese Dokumentation fertiggestellt, die Abschlusspräsentation vorbereitet, gehalten und letzte Fehler im System, die nach der Systemtest-Phase auftreten, behoben werden.

Generell gilt, dass die Phasen auf der Zeitachse sequentiell angeordnet sind. Eine Ausnahme bilden in diesem Fall die Phasen der Teilprojekte; diese laufen parallel zueinander. Abbildung 9.1 zeigt den Projektstrukturplan, die Meilensteine sind farblich hervorgehoben.

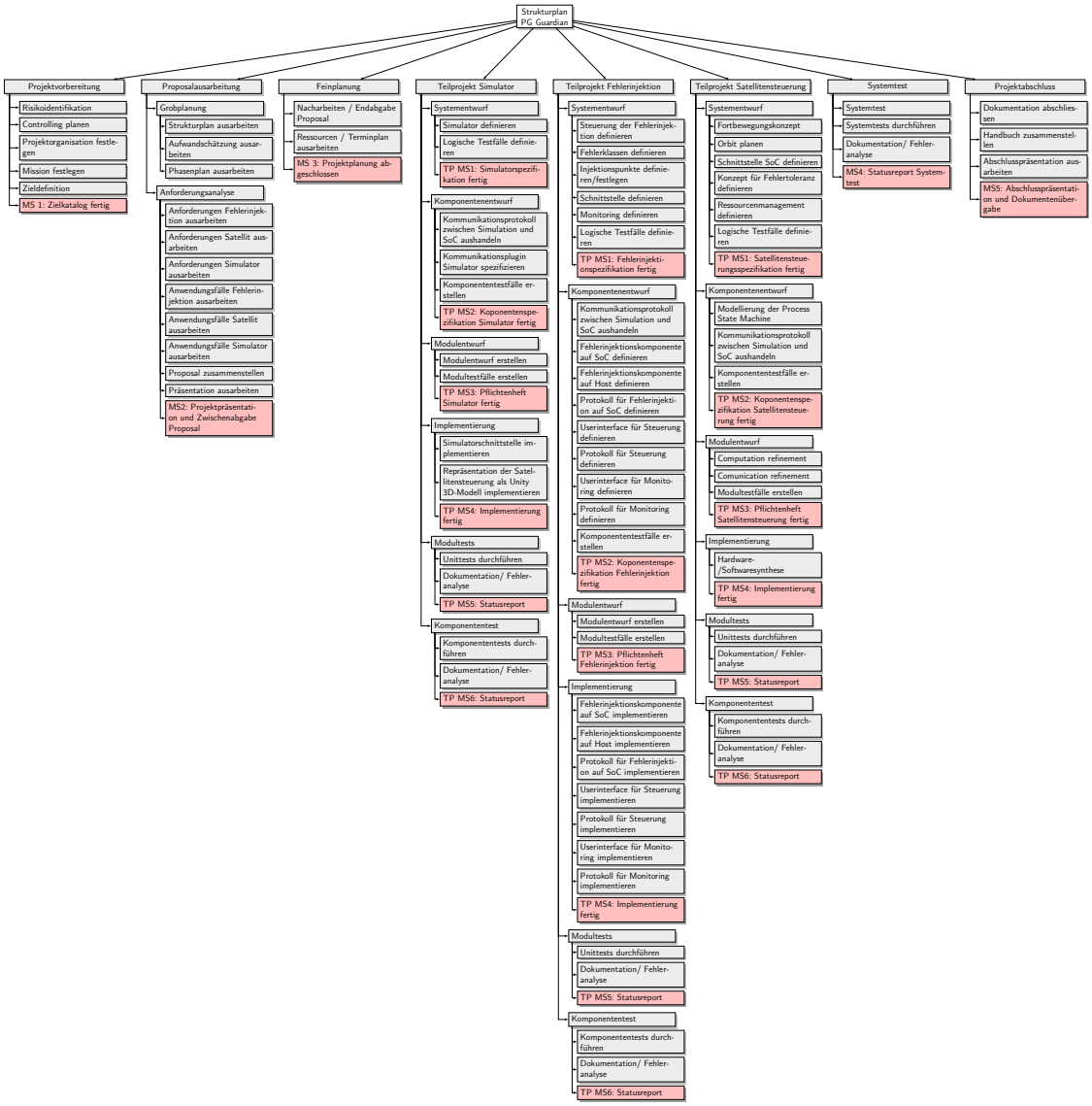


Abbildung 9.1.: In diesem detaillierten Strukturplan der PG Guardian lassen sich die einzelnen Arbeitspakete sowie deren Strukturierung erkennen.

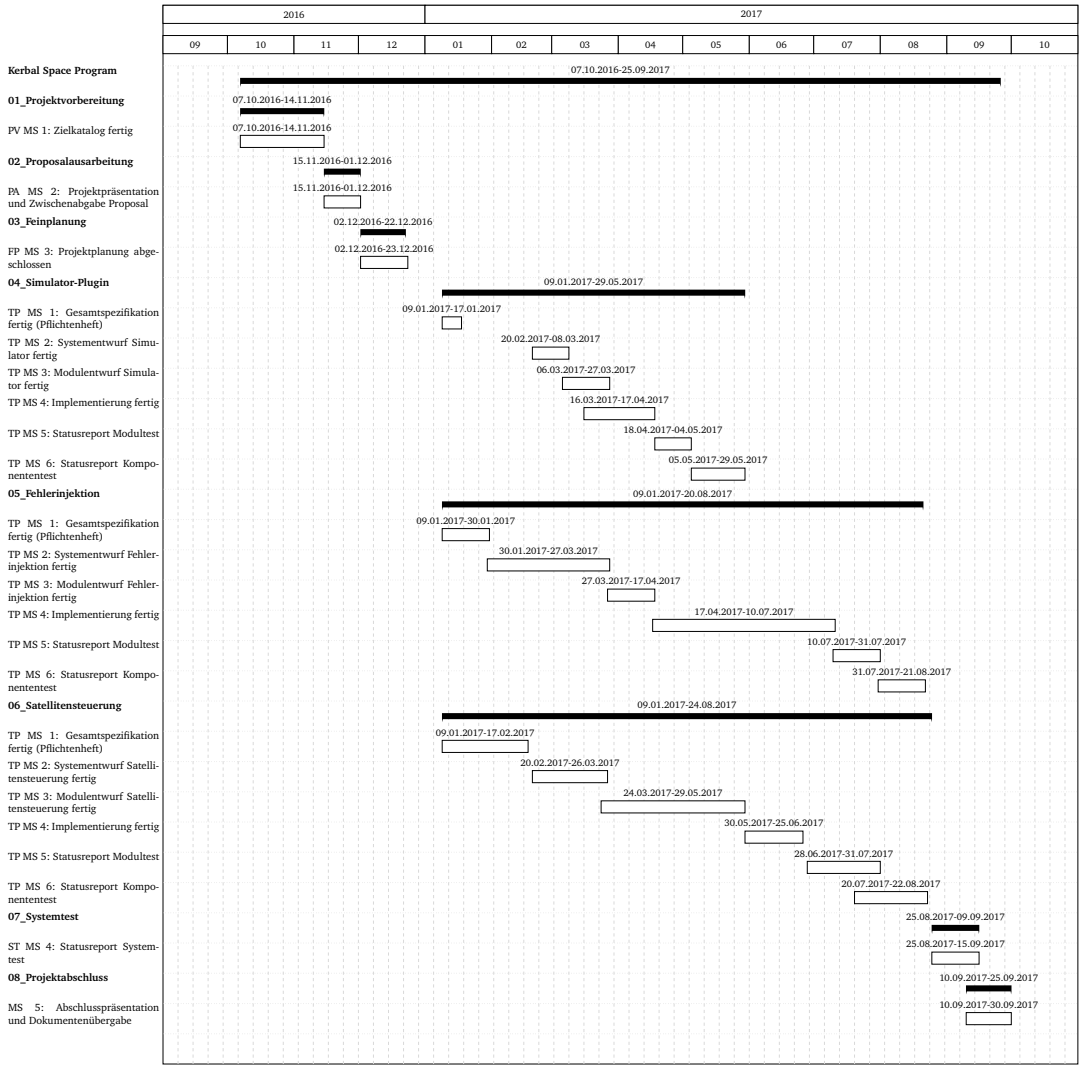


Abbildung 9.2.: Projektphasenplan der PG Guardian

9.2. Meilensteine

Aus dem Kick-off Workshop sind der Phasenplan und der Strukturplan entstanden. Hieraus lassen sich Meilensteine ableiten, die zum jeweiligen Phasenende erreicht werden müssen. Tabelle 9.1 listet die identifizierten Meilensteine für jede Phase mit dem entsprechenden Datum und einer knappen Beschreibung auf.

Tabelle 9.1.: Liste der identifizierten Meilensteine.

Meilensteinname	Datum	Beschreibumg
Projekt: Kerbal Space Program		
Projekt: 01_Projektvorbereitung		
PV MS 1: Zielkatalog fertig	14.11.2016	Zielkatalog
Projekt: 02_Proposalausarbeitung		
PA MS 2: Projektpräsentation und Zwischenabgabe Proposal	01.12.2016	Projektpräsentation & Proposal

Meilensteinname	Datum	Beschreibumg
Projekt: 03_Feinplanung		
FP MS 3: Projektplanung abgeschlossen	23.12.2016	Gantt-Diagramm, Auflistung der Arbeitpakete & finales Proposal
Projekt: 06_Satellitensteuerung		
TP MS 1: Gesamtspezifikation fertig (Pflichtenheft)	17.02.2017	Pflichtenheft Satellitensteuerung
TP MS 2: Systementwurf Satellitensteuerung fertig	26.03.2017	Systementwurf Satellitensteuerung
TP MS 3: Modulentwurf Satellitensteuerung fertig	29.05.2017	Mdulentwurf Satellitensteuerung
TP MS 4: Implementierung fertig	25.06.2017	Prototyp für Satellitensteuerung; Code-Dokumentation
TP MS 5: Statusreport Modultest	31.07.2017	Statusreport Modultest der Satellitensteuerung
TP MS 6: Statusreport Komponententest	22.08.2017	Statusreport Komponententest des Satellitensteuerung
Projekt: 05_Fehlerinjektion		
TP MS 1: Gesamtspezifikation fertig (Pflichtenheft)	30.01.2017	Pflichtenheft Fehlerinjektion
TP MS 2: Systementwurf Fehlerinjektion fertig	27.03.2017	Systementwurf Fehlerinjektion
TP MS 3: Modulentwurf Fehlerinjektion fertig	17.04.2017	Modulentwurf Fehlerinjektion
TP MS 4: Implementierung fertig	10.07.2017	Prototyp für Fehlerinjektion; Code-Dokumentation
TP MS 5: Statusreport Modultest	31.07.2017	Statusreport Modultest der Fehlerinjektion
TP MS 6: Statusreport Komponententest	21.08.2017	Statusreport Komponententest des Fehlerinjektion
Projekt: 04_Simulator-Plugin		
TP MS 1: Gesamtspezifikation fertig (Pflichtenheft)	17.01.2017	Pflichtenheft Simulator
TP MS 2: Systementwurf Simulator fertig	08.03.2017	Systementwurf Simulator
TP MS 3: Modulentwurf Simulator fertig	27.03.2017	Modulentwurf Simulator
TP MS 4: Implementierung fertig	17.04.2017	Prototyp für Symultor-Plugin; Code-Dokumentation
TP MS 5: Statusreport Modultest	04.05.2017	Statusreport Modultest des Simulators
TP MS 6: Statusreport Komponententest	29.05.2017	Statusreport Komponententest des Simulator-PlugIns
Projekt: 07_Systemtest		

Meilensteinname	Datum	Beschreibumg
ST MS 4: Statusreport Systemtest	15.09.2017	Abschlussbericht des Gesamtsystems
Projekt: 08_Projektabschluss		
Zwischenabnahme: Prototyp & Projektstand	31.03.2017	Host-onlyPrototyp, Kommunikation Host-Zynq, Präsentation & Projektplan
Präsentation(öffentlich): Prototyp & Projektstand	04.04.2017	Technischer Fortschritt & Projektplan
Präsentation(intern): Prototyp & Projektstand	06.06.2017	Struktur mit lokalen Dummyclients, Architektur auf Zynq-Plattform & Projektplan
MS 5: Abschlusspräsentation und Dokumentenübergabe	30.09.2017	Dokumentation, Handbuch, Präsentation & Gesamtsystem
Abschlusspräsentation(öffentlich): Demonstratorprojekt	04.10.2017	Projektplan, Projektverlauf & Demo

10. Qualitätssicherung

Die Qualitätssicherung erfolgt auf mehreren Ebenen. Für die Dokumentation werden geschriebene Texte innerhalb eines Teilprojektes einem *1. Review* unterzogen. Texte, die nach dem 1. Review überarbeitet wurden, werden für ein *2. Review* durch ein anderes Teilprojekt freigegeben. Auf diese Weise soll sichergestellt werden, dass die Texte verständlich und frei von Fehlern sind.

Die Qualitätssicherung des Systems erfolgt über Tests. Das System ist in Teilsysteme gegliedert, jedes Teilsystem besteht aus Komponenten, die wiederum aus Modulen bestehen. Daher werden zu jedem Modul Modultests, zu jeder Komponente Komponententests, sowie Systemtests geschrieben. Das Vorgehen beim Testen ist in einem Testplan festgehalten.

10.1. Testplan

Der Testplan soll zunächst Auskunft darüber geben, welche Rollen es im Bereich des Testens geben wird und welche Aufgaben damit verbunden sind. Des Weiteren wird das Testvorgehen näher beschrieben und die einzelnen Testphasen genauer beschrieben.

10.1.1. Rollen, Aufgaben und Verantwortlichkeiten

Tabelle 10.1 zeigt die aktuell vorgesehenen Rollen für die Testausführung.

Tabelle 10.1.: Rollen und Aufgaben in der Testorganisation

Rolle	**Aufgabe**
Testmanager	Stand aktuell: • Führen des Teilprojektteams • Erstellung High-Level Test Plan (HLTP) (Festlegung der notwendigen Teststufen, Testendekriterien) • Bericht an den Projektleiter (Statusmeeting) • Abstimmung der Testaktivitäten mit dem Projektleiter • Abstimmung mit den anderen Teilprojektleitern im Projekt • Anlage, Verwaltung und Überwachung der für die Arbeiten des Teilteams Testvorbereitung: • Erstellen des Release-Testplan • Detaillierte Planung und Tracking der Arbeitspakete • Abstimmung des internen Integrationstests sowie des Abnahmetests mit dem Kunden Testdurchführung: • Tägliche Kontrolle und Aktualisierung des Teststatus (Stand: Testfälle/Defekte) • Abstimmung mit dem Entwicklungsleiter zum Status der Defekte bezüglich der Bereitstellung des Releases • Freigabe des Releases entsprechend den definierten Testkriterien
Tester	• Erstellen und Pflegen der Testfälle • Durchführung der Tests • Erstellen der Defekteberichte und deren Nachverfolgung/Nachtest • Informieren des Testmanagers über Status und Testergebnisse aus den System-, Integrations- und Abnahmetests

Arbeitsergebnisse von Tests

Liste der Arbeitsergebnisse von Tests:

- HLTP
- Testpläne
- Testfälle
- Testzusammenstellungen basierend auf einem entsprechenden Testzyklus

- Protokolle von Testläufen
- Fehlerbeschreibungen (Defekt)
- Teststatusbericht (Statusmeeting)

Diese Dokumente gilt es im Verlauf des Projektes zu erstellen und zu verwalten. Dies wird in der Regel durch den Testmanager erledigt oder durch ihn an entsprechende Ressourcen verteilt. Zu den verteilten Aufgaben gehören beispielsweise die Protokolle der Testläufe und die Fehlerbeschreibungen beim Fund eines Defekts.

Teststatusbericht

Der Testfortschritt wird durch die folgenden Messgrößen für den aktuellen Testzyklus dargestellt:

- Anzahl aktuell durchgeführter Testfälle
- Anzahl aktuell gefundener Defekte
- Anzahl aktuell bestandener Testfälle
- Anzahl aktuell fehlgeschlagener Testfälle
- Anzahl aktuell blockierter Testfälle (Testfälle die auf Grund von Fehlern nicht durchgeführt werden können)
- Aktive Defekte nach Status

Durch die Erfassung dieser Daten kann der Fortschritt der Entwicklung in den unterschiedlichen Phasen nachvollzogen werden. An ihnen kann der Verlauf der Qualität in gewisser Form abgebildet werden, da eine stetige Verbesserung der Software nachzuweisen ist, welche sich in einem Rückgang der gefundenen und noch zu bearbeiteten Defekte widerspiegelt.

10.1.2. Teststrategie und Teststufen

Im Folgenden wird auf die im Projekt gewählte Teststrategie eingegangen, sowie auf die einzelenen Phasen des Testens (Teststufen) eingegangen. Die Teststufen beschreiben die Phasen des V-Modells, welches als Vorgehensmodell für das Projekt gewählt wurde. Zunächst soll in einer kurzen Erläuterung festgehalten werden, in welchem Umfang und in welcher Genauigkeit die Testfälle von dem jeweiligen Tester durchgeführt werden sollen.

Risikobasierte Teststrategie

Die Zeit und die Ressourcen für das Testen sind limitiert. Nicht alles kann daher mit gleichmäßiger Durchdringung getestet werden. Das bedeutet, dass betreffend der Testtiefe beziehungsweise des Testumfangs immer wieder Entscheidungen getroffen werden müssen. Auch muss das Bestreben bestehen, das Testteam und seine Aufgaben so effizient und effektiv wie möglich über das ganze Testprojekt aufzuteilen. Diese Leitlinie ist Grundlage der Teststrategie.

Als Teststrategie wird für alle genannten Komponenten des Systems das risikobasierte Testen eingesetzt. Dies bedeutet, dass Testumfang und Testintensität so gewählt werden, dass

der Testaufwand bei den gegebenen Rahmenbedingungen (Zeit, Ressourcen und Risiken) optimiert ist. Bei der Einschätzung der Risiken muss auch berücksichtigt werden, dass die meisten Fehler tendenziell in folgenden Komponenten auftreten:

- Komplexen Komponenten
- Ganz neu entwickelten Komponenten
- Häufig geänderten und optimierten Komponenten
- Komponenten, bei denen der Entwickler mehrfach gewechselt hat oder Entwickler unerfahren ist
- Komponenten, die unter extremen Zeitdruck entwickelt wurden
- Komponenten mit vielen Schnittstellen
- Komponenten, in denen in der Vergangenheit schon viele Fehler entdeckt wurden.

Die Risikoanalyse wird so früh wie möglich im Software-Lebenszyklus begonnen und wird durch die Experten, die das zu testende System und die Infrastruktur gut kennen, durchgeführt. Die risikobasierte Teststrategie erlaubt trotz knapper Ressourcen und Zeit nur die Testfälle auszuführen, die ein hohes Risiko für das System bedeuten. Die restlichen Testfälle können anschließend getestet werden, wenn es die Rahmenbedingungen (Zeit, Ressourcen und Risiken) erlauben.

Teststufen

Die nachfolgend beschriebenen Teststufen sind die, die in diesem HLTP beschreiben werden und mit der Projektleitung abgestimmt sind. Die Teststrategie für dieses Projekt ist in Tabelle 10.2 dargestellt. Diese ist an das V-Modell angelehnt und wird somit in den jeweiligen Phasen des Modells durchgeführt.

Tabelle 10.2.: Übersicht Teststufen

Teststufe	**Testart**	**Involviertes Team/Rolle**
Entwicklertest	**Explorativer- und Modultest:** Test der implementierten Units und Klassen durch den Entwickler selbst. Überprüfung der fachlichen Anforderungen auf unterster Ebene, zum Beispiel Berechnungsalgorithmus, Schnittstellenversorgung bezüglich Pflichterfüllung, Wertebereiche und weitere, Überprüfung der IT-Architektur, zum Beispiel Kommunikation und Schnittstellen zwischen Komponenten. Prüfung, ob die Testreife erreicht wurde.	Entwicklung
	Funktionale Verifikation: Test der implementierten Hardwarebeschreibungen durch den Entwickler selbst. Überprüfen ob die implementierte Hardwarebeschreibungen auf funktionaler Ebene der funktionalen Spezifikation der Komponente entspricht.	Entwicklung
	Automatisierte Tests: Test der sehr häufigen, fehlerträchtigen Softwarebestandteile	
Komponententests	**Explorative Tests:** Feststellung der Abnahmereife	Testteam
	Weiterentwicklungstest (Progressiver Test) : Test der geänderten Funktionalität auf Basis definierter Testfälle	Testteam
	Regressiver Test: Es wird sichergestellt, dass die Modifikationen in bereits geänderten Teilen der Software keinen neuen Fehler in dem Gesamtsystem verursachen. Feststellung der Abnahmereife	Testteam
Abnahmetest	Test auf Basis der Kundentestfälle beziehungsweise explorativer Test	Kunde

Modul- und Komponententests

Die grundlegenden Eigenschaften der Modultests sind in Tabelle 10.3 dargestellt. Diese werden im Zeitraum der Modultests, oder auch Unittests, durchgeführt.

Tabelle 10.3.: Kurzübersicht Modul- und Komponententests

Zeitraum	**Entwicklungszeitraum der Softwareentwicklung und beim Bugfixing**
Auslöser	Nach Fertigstellung von Softwareänderungen und nach bereinigten Defekten
Wer	Entwickler. Die Organisation, Durchführung und Dokumentation liegt in der Verantwortung der Entwicklung
Umgebung	Prototypumgebung Drittsysteme stehen für Schnittstellentests auf dieser Umgebung nicht beziehungsweise nur eingeschränkt zur Verfügung
Ergebnis	Die dabei auftretenden Fehler werden direkt an den Entwickler kommuniziert und behoben
Sonstiges	Die Erstellung der Unittests und Komponententests (automatisierte Tests) sind den übrigen Teststufen vorgelagert und dienen als Basis für die nachfolgenden Teststufen.

Die Testziele der Modul- und Komponententests sind folgende:

- Prüfung der Software auf Übereinstimmung mit den Anforderungen
- Die Anwendungsfälle der Systeme müssen fehlerfrei ausgeführt werden können
- Erreichen der Testreife und damit die Voraussetzung für den Übergang in die nächste Teststufe

Die Bezeichnungen auf dieser Ebene sind folgende:

Testgegenstand / Abgrenzung: Testumfang können alle Systeme des Projektes, mit all ihren Teilkomponenten sein. Je nachdem, an was durch die Entwicklung gearbeitet wurde.

Zu testende Leistungsmerkmale: Es werden, soweit nicht anders vereinbart beziehungsweise gefordert, Funktionaltests durchgeführt.

Testmethoden / Testabdeckungen: Die Modultests finden in Form von explorativen Tests, Unittests oder funktionaler Verifikation statt. Darüber hinaus stehen für bestimmte Funktionsumfänge automatisierte Tests zur Verfügung (Komponententests). Die dabei auftretenden Fehler werden direkt an den Entwickler kommuniziert.

Testeingangskriterien: Der notwendige Quellcode wurde erstellt.

Testausgangskriterien: Alle Modultests sind gelaufen. Nicht lauffähige Tests wurden bewertet, klassifiziert und gegebenenfalls dokumentiert. Der Modultest kann dann verlassen werden, wenn alle Weiterentwicklungsthemen getestet und bewertet wurden. Es liegen keine Test verhindernden Fehler (Blocker) vor. (Ein Codereview wurde durchgeführt). Testreife muss erreicht worden sein.

Testabbruchkriterien: Es ist ein Vorzeitiger Abbruch möglich, wenn keine weiteren Fehler mit hoher Kritikalität gefunden werden oder Fehler auftauchen, die eine saubere Ausführung der Modultests verhindern. Diese müssten dann vorab korrigiert werden.

Systemintegrationstest

Eine Übersicht der grundlegenden Eigenschaften des Systemintegrationstest sind in Tabelle 10.4 dargestellt. Diese werden im Zeitraum der Integrationsphase durchlaufen.

Tabelle 10.4.: Kurzübersicht Systemintegrationstest

Zeitraum	Ab Bereitstellung des Prototypen zum Meilenstein
Auslöser	Die Entwicklung des Prototypen ist grundlegend abgeschlossen und alle Weiterentwicklungen sind implementiert und haben Testreife für den Systemintegrationstest erreicht
Wer	Testteam
Umgebung	Prototypumgebung
Basis	Testspezifikationen und Testfälle für die Weiterentwicklungen.
Ergebnis	Abweichungen, Kenntnis über den Status des endgültigen Entwicklungsstandes → Entscheidungspunkt: Software bereit zur Abnahme

Die Testziele des Systemintegrationstest sind folgende:

- Überprüfung des Systems unter möglichst abnahmefähigen Bedingungen
- Progressive und regressive Tests umfassen die fehlerfreie Durchführung der Testfälle, sowie explorative Tests
- Definierte Testabdeckung aller neu entwickelten Themen
- Definierte Testabdeckung des Systems (Regressionstest)
- Nichtfunktionale Anforderungen werden getestet
- Sicherstellung der Abnahmetestreife

Testgegenstand / Abgrenzung: Komplette Applikation mit allen Schnittstellen zu externen Systemen

Zu testende Leistungsmerkmale: System mit Schwerpunkt auf den definierten Anforderungen

Testmethoden: Funktionale Blackbox-Tests (Anwendungsfall basiert), testfallgestützt, basierend auf der Anforderung und explorativer Test

Testabdeckungen: Für die Integrationstests wird eine fachliche Testabdeckung eingesetzt. Die fachliche Testabdeckung sagt aus, inwieweit die gesamte Funktionalität im System durch einen Test geprüft wurde.

Absolute Testabdeckung: Alle einzelnen Strukturen des Systems werden bewertet. Momentan wird noch keine Gewichtung für Strukturen vorgenommen.

Relative Testabdeckung: Zwingende Voraussetzung für die Bestimmung der relativen fachlichen Testabdeckung ist eine Risikobewertung auf funktionaler Ebene.

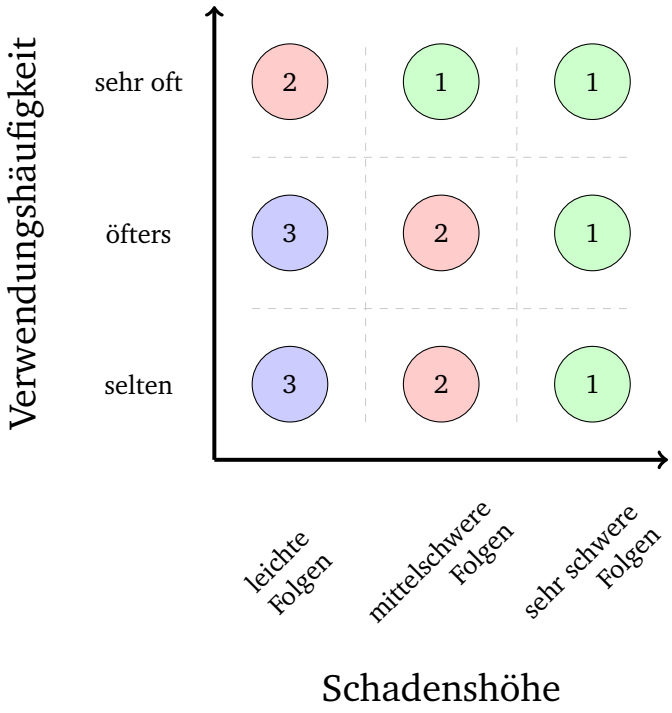


Abbildung 10.1.: Risikobasiertes Testen (Risikomatrix)

Testeingangskriterien: Die Entwicklung meldet den kompletten Prototypen als fertig und stellt diesen ohne bekannte testverhindernde Fehler (Blocker) bereit.

Testausgangskriterien: Es existieren keine abnahmehinderlichen Fehler.

Testabbruchkriterien: Ein vorzeitiger Abbruch ist möglich, wenn keine weiteren abnahmeverhindernde Fehler im gesamten Prototypen gefunden werden oder wenn schwerwiegende Fehler (Blocker) im System vorhanden sind.

Risikomanagement der Entwicklertests

- Verwendungshäufigkeit der Funktion in der Praxis (Stand heute: in dem Testfall definierte Verwendungshäufigkeit des beschriebenen Testfallablaufs in der Praxis)
- Potenzieller Schaden, der bei Eintritt eines Fehlers entstehen kann

$$\text{Risiko} = \text{Verwendungshäufigkeit} * \text{Schadenshöhe}$$

Im folgenden sind Risiken und deren Beispiele dazu dargestellt:

1. **Hoch:** Beispiel: Verbindung zum Board geht nicht: Häufige Verwendung, Große Schadenshöhe, da Vorhaben nicht abgewickelt werden kann → Risikogruppe 1.
2. **Mittel:** Beispiel: Rotation um 720° statt 360°: Rotationen werden oft benötigt, aber Schadenshöhe gering, da korrigiert werden kann → Risikogruppe 2.
3. **Niedrig:** Beispiel: Lampe blinkt nicht: wird selten verwendet, kein Schadenspotential → Risikogruppe 3.

Abnahmetest

Eine Übersicht der grundlegenden Eigenschaften des Abnahmetests sind in Tabelle 10.5 dargestellt. Sie wird im Zeitraum der gleichnamigen Phase des V-Modells durchgeführt.

Tabelle 10.5.: Kurzübersicht Abnahmetest

Zeitraum	Nach erfolgreichem Abschluss des Integrationstest
Auslöser	Bereitstellung eines Prototypen und Terminfindung zum Start des Abnahmetests mit dem Entwicklungsleiter
Wer	Kunde Begleitung durch Projektgruppe
Umgebung	Prototypumgebung
Basis	Testszenarien auf Basis der Kundentestfälle beziehungsweise explorative Tests des Kunden
Ergebnis	Unterzeichnetes Abnahmeprotokoll Liste von Defekten, die nicht mehr vor dem Abgabedatum behoben werden Potentielle Weiterentwicklungswünsche werden durch den „Kunden“ als entsprechendes erfasst

Die Testziele des Abnahmetests sind folgende:

- Ausreichende Testabdeckung aller entwickelten Themen
- Explorative Tests
- Nichtfunktionale Anforderungen aus Kundensicht
- Sicherstellung der Qualität des Gesamtsystems aus Kundensicht vor der Abnahme

Testgegenstand / Abgrenzung: Alle Systeme mit allen Schnittstellen zu den Drittsystemen

Zu testende Leistungsmerkmale: Gesamtes System mit Schwerpunkt auf dem definierten Abnahmeumfang

Testmethoden / Testabdeckungen: Explorative Tests

Testeingangskriterien: Es existieren keine abnahmeverhindernden Fehler

Testausgangskriterien: Es existieren keine Fehler, die eine Abnahme verhindern. Der Kunde erteilt die Abnahme, möglicherweise auch unter Vorbehalt, wenn ein Fehler vorliegt der noch behoben werden muss.

Testabbruchkriterien: Vorzeitiger Abbruch möglich, wenn keine weiteren abnahmeverhindernden Fehler in der Abnahme gefunden werden oder wenn schwerwiegende Fehler (Blocker) im System vorhanden sind oder Fehler auftauchen, die eine weitere Ausführung der Abnahmetest als nicht sinnvoll erscheinen lassen.

10.2. Testfälle

Im Folgenden werden die Testfälle aufgeführt, welche die funktionale Qualität des Produktes der PG Guardian sichern sollen. Dabei werden die Testfälle in logische Testfälle (Abschnitt 10.2.1), Komponententestfälle (Abschnitt 10.2.2) und Modultestfälle unterteilt. Die Modultestfälle sind umfassend in den Testbenches dokumentiert und werden dementsprechend an dieser Stelle nicht komplett aufgelistet.

10.2.1. Logische Testfälle

Im Folgenden werden die logischen Testfälle der Fehlerinjektion (Abschnitt 10.2.1), der Satellitensteuerung (Abschnitt 10.2.1) und der Simulation (Abschnitt 10.2.1) aufgeführt. Dazu wird jeder logische Testfall und dessen erwartetes Resultat mit einem Bezeichner, bestehend aus einem Indikator (hier LTC für logical testcase), und einer fortlaufenden Nummer aufgeführt. Weiter werden die Anforderungen referenziert, welche im jeweiligen Testfall auf Umsetzung getestet werden sollen. Da implizite Anforderungen nicht referenziert werden können, werden diese durch „implizit“ in der Referenzspalte gekennzeichnet.

Simulation

In diesem Abschnitt werden die logischen Testfälle der Simulation (Tabelle 10.6) festgelegt und beschrieben. Dabei werden die logischen Testfälle in logische Testfälle zu funktionalen Anforderungen und logische Testfälle zu nichtfunktionalen Anforderungen unterteilt.

Tabelle 10.6.: Logische Testfälle der Simulation

Bez.	Testfall	Erwartetes Resultat	Ref.
Logische Testfälle der funktionalen Anforderung			
LTC001	Die Schnittstelle prüfen, ob alle festgelegten Funktionalitäten vorhanden sind	Alle Funktionalitäten gemäß der Schnittstellenspezifikation sind implementiert.	FA003
LTC002	Prüfen, ob die Schnittstelle die Pakete im vordefinierten Format überträgt.	Alle Antworten müssen gemäß der Schnittstellenspezifikation strukturiert sein.	FA003
LTC003	Prüfen, ob die Schnittstelle auf Pakete reagiert, die einen inkonsistenten Inhalt hat.	Die Schnittstelle darf nicht abstürzen bei fehlerhaften oder unvollständigen Paketen.	FA003
LTC004	Prüfen, ob die Schnittstelle es erlaubt jede von der Steuerung verwendete Komponente einzeln anzusprechen.	Die Schnittstelle muss jede Komponente einzeln adressieren können und die dafür angefragten Eigenschaften zurückliefern beziehungsweise Steuerbefehle annehmen.	UC001
LTC005	Prüfen, ob die Schnittstelle alle Zustände des Satelliten auf Anfrage übermittelt.	Die Schnittstelle muss auf Anfrage die jeweiligen Werte übermitteln.	UC002

Tabelle 10.6.: Logische Testfälle der Simulation

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC006	Prüfen, ob die Schnittstelle alle Zustände der Trägerrakete auf Anfrage übermittelt.	Die Schnittstelle muss auf Anfrage die jeweiligen Werte übermitteln.	UC002
LTC007	Prüfen, ob die Schnittstelle alle Umweltzustände auf Anfrage übermittelt.	Die Schnittstelle muss auf Anfrage die jeweiligen Werte übermitteln.	UC002
LTC008	Prüfen, ob die Schnittstelle alle Zustände des Zielsasteroiden auf Anfrage übermittelt.	Die Schnittstelle muss auf Anfrage die jeweiligen Werte übermeitteln.	UC002
LTC009	Prüfen, ob die Schnittstelle es ermöglicht einen aktuellen Energieverbrauch der Mikrocontroller zu akzeptieren.	Die Schnittstelle bietet die Möglichkeit einen Stromverbrauch zu definieren.	UC009
LTC010	Prüfen, ob die Simulation den derzeit bekannten Stromverbrauch der Mikrocontroller in der Simulation verbraucht.	Der definierte Stromverbrauch wird in der Simulation umgesetzt.	UC009
LTC011	Prüfen, ob Komponenten, welche Electric-Charge verbrauchen und nicht die Kernfunktionalität des Satelliten beeinflussen, sich abschalten lassen.	Die Komponenten konnten abgeschaltet werden. Die Kernfunktionen des Satelliten wurden nicht beeinflusst.	FA007
LTC012	Prüfen, ob die Schnittstelle alle ansprechbaren Aktoren und Sensoren Anfrage übermittelt.	Die Schnittstelle muss auf Anfrage die jeweiligen Werte übermitteln.	FA009
LTC013	Prüfen, das Simulator-Plugin in der Lage ist die Simulation zu pausieren oder zu beschleunigen abhängig von dem Zustand der Mikrocontroller.	Abhängig von den Zuständen der Mikrocontroller wird die Simulation beschleunigt oder pausiert.	FA010
Logische Testfälle der nichtfunktionalen Anforderung			
LTC014	Prüfen, ob die Schnittstelle die von der Steuerung angefragten Werte in der vorgegebenen Reaktionszeit liefert oder gegebenenfalls die gegebenen Werte in der vorgegebenen Reaktionszeit setzt.	Alle vorgegebenen Reaktionszeiten werden von der Schnittstelle eingehalten.	NA004

Satellitensteuerung

In diesem Abschnitt werden die logischen Testfälle der Satellitensteuerung festgelegt und beschrieben. Dabei werden die logischen Testfälle in logische Testfälle des Fortbewegungskonzeptes (Tabelle 10.7), logische Testfälle des Fehlertoleranzkonzeptes (Tabelle 10.8), logische Testfälle des Ressourcenmanagements (Tabelle 10.9), logische Testfälle der Orbitplanung (Tabelle 10.10) und logische Testfälle der Kommunikation (Tabelle 10.11) unterteilt. Logische Testfälle ohne geeignete Referenz werden mit „/“ in der Referenzspalte markiert.

Tabelle 10.7.: Logische Testfälle des Fortbewegungskonzeptes

Bez.	Testfall	Erwartetes Resultat	Ref.
Aufstieg in den Orbit von Kerbin			
LTC015	Die Thrust-to-Weight-Ratio der Trägerrakete ist größer als Eins.	Die Trägerrakete kann abheben.	implizit
LTC016	Das ΔV der Trägerrakete reicht aus, um ebendiese in einen Orbit von 80km um Kerbin zu bringen.	Der Satellit befindet sich in einem Orbit von 80km um Kerbin.	FA026, FA029
LTC017	Beim Flug von der Startrampe bis in den Orbit von Kerbin wird der Gravity Turn eingesetzt.	Die Rakete wird nicht im rechten Winkel in die Umlaufbahn geschossen. Es wird ein Kompromiss zwischen Energieverlust durch Schwerkraft und durch Luftwiderstand gefunden.	FA012, FA013
LTC018	Es werden die Kräfte durch den Luftwiderstand und der Schwerkraft berechnet und somit die Grenzgeschwindigkeit ermittelt.	Beim Aufstieg in den Orbit 80km um Kerbin passt sich die Trägerrakete die Geschwindigkeit der Grenzgeschwindigkeit an.	FA016, FA024
Orbittransfer von Kerbin zu Kerbol			
LTC019	Der Satellit passt seine Inklination des Orbits an die Inklination des Orbits des Asteroiden an.	Die maximale Änderung überschreitet nicht 90 Grad. Die Inklinationen sind gleich.	FA016
Veränderung des Asteroidenorbits			
LTC020	Der Satellit greift den Asteroid.	Der Asteroid zählt als zusätzliche Masse des Satelliten.	implizit
LTC021	Der Satellit passt den Schub auf die zusätzliche Masse an.	Der Orbit des Asteroiden ändert sich.	FA016, FA017
Orientierung des Satelliten			
LTC022	Der Satellit verfügt über ein Reaction Control System (RCS) und Monopropellant-Treibstoff.	Der Satellit kann seine Orientierung ändern. Der Satellit besitzt die Möglichkeit zur Feinjustierung, um sich dem Asteroiden anzunähern.	FA016, FA021, FA023

Tabelle 10.7.: Logische Testfälle des Fortbewegungskonzeptes

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC023	Der Satellit verfügt über ein Reaktionsrad und electric charge.	Der Satellit kann seine Orientierung ändern.	FA016, FA021, FA023

Tabelle 10.8.: Logische Testfälle des Fehlertoleranzkonzeptes

Bez.	Testfall	Erwartetes Resultat	Ref.
Systemebene			
LTC024	das Steuerungssystem besitzt Monitore, Controller und Adapter, die adaptives Verhalten implementieren.	Adaptives Verhalten ist möglich.	FA018
LTC025	Das Satellitensteuerungssystem muss ideal (d.h.fehlerfrei) und mit Fehlern und deren Auswirkungen modelliert werden.	Es ist möglich Fehler und deren Auswirkungen und das Adaptionsverhalten zu modellieren.	FA018
LTC026	Eine Adaption wird notwendig und die entsprechende Komponente sendet Events.	Diese Events werden über Monitore erfasst.	FA018
LTC027	Der Controller reagiert auf das Event auf seinen Monitoren.	Ein Adapter führt eine entsprechende Fehlertoleranzmaßnahme durch.	FA018
LTC028	Eine Komponente bemerkt selbstständig einen Fehler.	Diese Komponente führt Fehlertoleranzmechanismen durch, um auf diese zu reagieren.	FA018
LTC029	Es soll fehlertolerant auf Alterung regiert werden.	Das Scheduling wird verändert.	FA018
Komponentenebene und Modulebene			
LTC030	Es wird Monitoring von Komponenten eingesetzt.	Watchdogs überwachen die Aktivitäten der Komponenten.	implizit
LTC031	Es werden Acceptance tests eingesetzt.	Watchdogs kontrollieren Output, Input und Timings dieser Komponenten.	implizit
LTC032	Es wird checkpointing eingesetzt.	Komponenten können bei Fehlern in funktionsfähige Zustände zurückgesetzt werden.	implizit
LTC033	Es wird n-Modulares Multiplexing eingesetzt.	Komponenten besitzen mehrere Module für dieselbe Funktion. Aus diesen wird per Mehrheitsentscheid ein gültiger Wert bestimmt.	implizit

Tabelle 10.8.: Logische Testfälle des Fehlertoleranzkonzeptes

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC034	Es wird multiplicated Multiplexing eingesetzt.	Eine Funktion wird nur von einem Modul eingesetzt. Beim Erkennen von Fehlern wird auf andere Module zurückgegriffen.	implizit
LTC035	Es wird ein Hybrider Ansatz aus n-modularem und multiplicated Multiplexing eingesetzt.	Es werden die zuvor genannten Möglichkeiten kombiniert.	implizit
LTC036	Es wird Informationsredundanz eingesetzt.	Der Kommunikationsoverhead erhöht sich. Hierzu werden Paritybits oder Checksums verwendet.	implizit
LTC037	Es wurde Monitoring, Acceptance Tests, Checkpointing, modulares, multiplicated, hybrides Multiplexing oder Informationsredundanz eingesetzt.	Die Satellitensteuerung handelt fehlertolerant und besitzt Adaptionsverfahren.	FA018
LTC038	Die Komponentenentwürfe liegen ohne Berücksichtigungen von Fehlern als Statemachines vor.	Es ist möglich ein Fehlermodell aufzustellen.	/
LTC039	Die Komponentenentwürfe besitzen Fehlerzustände.	Es werden Fehler erkannt und als permanent oder transient klassifiziert.	/
LTC040	Die Fehlerzustände wurden identifiziert.	Es wird entschiedenen, welche Fehler gleichzeit toleriert werden können.	/
LTC041	Die Satellitensteuerung gerät in einen Fehlerzustand.	Der Fehler wird klassifiziert, und dann behoben oder toleriert.	/

Tabelle 10.9.: Logische Testfälle des Ressourcenmanagement

Bez.	Testfall	Erwartetes Resultat	Ref.
Verbraucher			
LTC042	Ein Verbraucher wird eingeschaltet	Der Verbrauch einer physikalischen Ressource steigt.	FA023, FA022
LTC043	Ein Verbraucher wird ausgeschaltet	Der Verbrauch einer physikalischen Ressource sinkt.	FA023, FA022
LTC044	Ein Verbraucher wird in Bereitschaft versetzt	Der Verbrauch einer physikalischen Ressource sinkt.	FA023
LTC045	Ein Verbraucher wird aus Bereitschaft aufgeweckt	Der Verbrauch einer physikalischen Ressource steigt.	FA023
LTC046	Die Leistung eines Verbrauchers wird gedrosselt.	Der Verbrauch einer physikalischen Ressource sinkt.	FA023

Tabelle 10.9.: Logische Testfälle des Ressourcenmanagement

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC047	Die Leistung eines Verbrauchers wird gesteigert.	Der Verbrauch einer physikalischen Ressource steigt.	FA023
Erzeuger			
LTC048	Ein Erzeuger wird eingeschaltet.	Die Produktion einer physikalischen Ressource steigt.	FA023, FA022
LTC049	Ein Erzeuger wird ausgeschaltet.	Die Produktion einer physikalischen Ressource sinkt.	FA023, FA022
LTC050	Ein Erzeuger wird in Bereitschaft versetzt	Die Produktion einer physikalischen Ressource sinkt.	FA023
LTC051	Ein Erzeuger wird aus Bereitschaft aufgeweckt	Die Produktion einer physikalischen Ressource steigt.	FA023
Speicher			
LTC052	Ein physikalischer Ressourcenspeicher ist voll.	Der zugehörige Erzeuger wurde ausgeschaltet oder in Bereitschaft versetzt.	/
LTC053	Ein physikalischer Ressourcenspeicher ist nicht voll.	Der zugehörige Erzeuger ist eingeschaltet und nicht in Bereitschaft.	/
LTC054	Ein physikalischer Ressourcenspeicher leert sich.	Ein zugehöriger Verbraucher ist eingeschaltet. Verbrauch ist größer als Erzeugung.	/
LTC055	Ein physikalischer Ressourcenspeicher füllt sich.	Ein zugehöriger Erzeuger ist eingeschaltet und nicht in Bereitschaft. Erzeugung ist größer als Verbrauch.	/
Ressourcen			
LTC056	Jede Ressource besitzt Erzeuger, Verbraucher und ein Leitungsnetzwerk.	Es kann Ressourcenverbrauch geleitet werden.	/
LTC057	Ein Verbraucher einer physikalischen Ressource ist nicht mehr kontrollierbar.	Die Leitung zu diesem Verbraucher wird gesperrt und es werden Maßnahmen unternommen, diesen zurückzusetzen.	/
LTC058	Ein Verbraucher einer physikalischen Ressource wird vom Satelliten entfernt.	Die Leitung zu diesem Verbraucher wird gesperrt und bleibt gesperrt.	/
LTC059	Das Satellitensteuerungssystem startet.	Alle Ressourcenleitungen sind gesperrt.	/

Tabelle 10.10.: Logische Testfälle der Orbitplanung

Bez.	Testfall	Erwartetes Resultat	Ref.
Ausgangszustand			
LTC060	Der Satellit befindet im Ausgangsorbit.	Die aktuelle Position bestimmt durch Orbitradius (r_{LEO}), Inklination (i_{LEO}) und Anomalie Θ_{LEO} sind bekannt.	FA015
LTC061	Der abzufangende Asteroid wird mitgeteilt.	Die Parameter, Abstand der Apoapsis $r_{Ap,Ast}$, der Abstand der Periapsis $r_{Pe,Ast}$, das Argument der Periapsis ω_{Ast}, die Länge des aufsteigenden Knotens Ω_{Ast}, die Inklination i_{Ast} und die aktuelle wahre Anomalie Θ_{Ast} des abzufangenden Asteroiden sind bekannt.	FA017
LTC062	Die Flugbahn des Asteroid kreuzt keine Sphere of Influence (SOI), außer Kerbin.	Der Asteroid wird nicht von anderen Himmelskörpern beeinflusst.	/
LTC063	Die aktuelle wahre Anomalie Θ_{Kerb} des Planeten Kerbins ist bekannt.	Der Satellit kann sich ausrichten.	FA015, FA013
Patched Conic Annährung			
LTC064	Es wird der Orbit von Kerbin über eine (hyperbolische) Fluchtbahn verlassen.	Der Satellit befindet daraufhin in einem elliptischem Orbit um Kerbol.	/
LTC065	Der Satellit befindet sich im Orbit von Kerbol.	Die Annährung an den Asteroiden ist möglich.	/
Fluchtorbit			
LTC066	Der Satellit befindet sich im Warteorbit um Kerbin.	Der Satellit beschleunigt auf Fluchgeschwindigkeit und verlässt Kerbin auf der Apsenlinie des Asteroiden.	/
Phasenanpassung			
LTC067	Der Satellit befindet sich auf demselben Orbit wie der Asteroid.	Der Satellit passt seine Phase an, sodass er auf den Orbit trifft.	FA015
Finale Annährung			
LTC068	Satellit und Asteroid befinden sich auf derselben Apoapsis.	Der Satellit nähert sich dem Asteroid durch gezieltes Einsetzen des Antriebes an.	FA011, FA012

Tabelle 10.11.: Logische Testfälle der Kommunikation

Bez.	Testfall	Erwartetes Resultat	Ref.
Feldbus			
LTC069	Alle Systeme starten.	Field Bus Master ist unbekannt und wird gewählt.	implizit
LTC070	Der Field Bus Master fällt aus.	Es wird ein neuer gewählt.	implizit
LTC071	Der Field Bus Master sendet falsche Befehle.	Es wird ein neuer gewählt. Das System des Field Bus Masters leitet Fehlerbehandlungsmaßnahmen ein.	implizit
LTC072	Eine Field Bus Komponente (nicht Master) akzeptiert keine Befehle mehr.	Das fehlerhafte System leitet Fehlermaßnahmen ein.	FA018
Kommunikation der Satellitensteuerungssysteme untereinander			
LTC073	Alle Systeme starten.	Leader ist unbekannt und wird gewählt.	implizit
LTC074	Der Leader fällt aus.	Ein neuer Leader wird gewählt.	FA018
LTC075	Der Leader sendet Missionsübergänge, die von der Mehrheit nicht akzeptiert werden.	Ein neuer Leader wird gewählt. Das alte Leader-System leitet Fehlerbehandlungsmaßnahmen ein.	FA018
LTC076	Unterschiedliche Komponenten der einzelnen Systeme sind nicht mehr herstellbar.	Es wird versucht per *Data Exchange* fehlende Daten zwischen den Systemen zu ergänzen.	FA018

Fehlerinjektion

In diesem Abschnitt werden die logischen Testfälle der Fehlerinjektion (Tabelle 10.12) festgelegt und beschrieben. Dabei werden die logischen Testfälle in allgemeine logische Testfälle, logische Testfälle zu funktionalen Anforderungen und logische Testfälle zu nichtfunktionalen Anforderungen unterteilt.

Tabelle 10.12.: Logische Testfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
Allgemeine logische Testfälle der Fehlerinjektions-Software (FI-Software)			
LTC077	Prüfen, ob die FI-Software gestartet werden kann.	Die FI-Software wird gestartet und das Hauptmenü wird angezeigt.	implizit
LTC078	Prüfen, ob die FI-Software geschlossen werden kann	Alle Prozesse der FI-Software wurden beendet.	implizit
logische Testfälle zu den funktionalen Anforderungen			
LTC079	Prüfen, ob Umweltsignale manipuliert werden können.	Umweltsignale können manipuliert werden.	FA030

Tabelle 10.12.: Logische Testfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC080	Prüfen, ob die von der Satellitensteuerung an die Simulation gesendeten Steuerungsbefehle manipuliert werden können.	Steuerungssignale, die von der Satellitensteuerung an die Simulation gesendet werden, können manipuliert werden.	FA031
LTC081	Prüfen, ob Fehler zur Manipulation des Satellitenzustands vom Benutzer an die Satellitensteuerung gesendet werden können.	Fehler zur Manipulation des Satellitenzustands können vom Benutzer an die Satellitensteuerung gesendet werden.	FA032
LTC082	Prüfen, ob die Satellitensteuerung Fehler zur Manipulation des Satellitenzustands erhält.	Die Satellitensteuerung erhält Fehler zur Manipulation des Satellitenzustands.	FA033
LTC083	Prüfen, ob empfangene Fehler zur Manipulation des Satellitenzustands injiziert werden.	Empfangene Fehler zur Manipulation des Satellitenzustands werden injiziert.	FA034
LTC084	Prüfen, ob Fehler in einzelne Satellitenkomponenten injiziert werden können.	Fehler können in einzelne Satellitenkomponenten injiziert werden.	FA035
LTC085	Prüfen, ob die Satellitenkomponenten der Fehlerinjektion bekannt sind.	Der Fehlerinjektion sind die Satellitenkomponenten bekannt.	FA036
LTC086	Prüfen, ob Informationen über die Satellitenkomponenten vorhanden sind.	Es sind Informationen über die Satellitenkomponenten vorhanden.	FA037
LTC087	Prüfen, ob die Kommunikation zwischen Simulator und Satellitensteuerung abgefangen oder zwischengespeichert wird.	Die Kommunikation zwischen Simulator und Satellitensteuerung wird abgefangen oder zwischengespeichert.	FA038
LTC088	Prüfen, ob es ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellitensteuerung gibt.	Es existiert ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellitensteuerung.	FA039
LTC089	Prüfen, ob ein Satellitenlog existiert, welches an die Fehlerinjektionskomponente auf dem Host-PC gesendet wird.	Es existiert ein solcher Log.	FA041
LTC090	Prüfen, ob es eine Systemzustandsüberwachung der Satellitensteuerung für den Fehlerinjektionszugriff gibt.	Es existiert eine solche Systemzustandsüberwachung.	FA041

Tabelle 10.12.: Logische Testfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC091	Prüfen, ob die Maßnahmen der Satellitensteuerung dem Anwender zur Verfügung stehen.	Die Maßnahmen der Satellitensteuerung stehen dem Anwender zur Verfügung.	FA042
LTC092	Prüfen, ob jeder Log-Eintrag mit einem Timestamp enthält.	Alle Log-Einträge enthalten einen Timestamp.	FA043
LTC093	Prüfen, ob eine Speichereinheit aller injizierten Fehler existiert.	Es existiert eine solche Speichereinheit.	FA044
LTC094	Prüfen, ob auf die Speichereinheit aller injizierten Fehler zugegriffen werden kann.	Es kann auf die Speichereinheit aller injizierten Fehler zugegriffen werden.	FA045
LTC095	Prüfen, ob ein UI zur Anzeige aller injizierten Fehler existiert.	Es existiert ein UI zur Anzeige aller injizierten Fehler.	FA046
LTC096	Prüfen, ob die Kommandozeile auf die Speichereinheit aller injizierten Fehler zugreifen kann.	Die Kommandozeile kann auf die Speichereinheit zugreifen.	FA047
LTC097	Prüfen, ob der Fehlerinjektionszugriff auf Speicherblöcken oder durch speziell definierte Komponenten geschieht.	Der Fehlerinjektionszugriff geschieht auf Speicherblöcken oder durch speziell definierte Komponenten.	FA048
LTC098	Prüfen, ob dem Anwender Eingabebefehle zur Verfügung gestellt werden.	Den Anwender werden Eingabebefehle zur Verfügung gestellt.	FA049
LTC099	Prüfen, ob eine GUI zur Steuerung der Fehlerinjektion existiert.	Es existiert eine solche GUI.	FA050
logische Testfälle zu den nichtfunktionalen Anforderungen			
LTC100	Prüfen, ob die Nutzerschnittstelle der Fehlerinjektion einfach bedienbar ist.	Die Nutzerschnittstelle der Fehlerinjektion ist einfach bedienbar.	NA009
LTC101	Prüfen ob Fehler in unter einer Sekunde injiziert werden.	Fehler werden in unter einer Sekunde injiziert.	NA010
LTC102	Prüfen, ob die Fehlerinjektionskomponente auf dem FPGA möglichst platzsparend ist.	Die Fehlerinjektionskomponente auf dem FPGA ist so platzsparend wie möglich.	NA011
LTC103	Prüfen, ob die Satellitenkomponenten auf mögliche Fehlerquellen untersucht wurden.	Die Satellitenkomponenten wurden auf mögliche Fehlerquellen untersucht.	NA012
LTC104	Prüfen, ob die Auswahl der injizierbaren Fehler definiert wurde.	Die Auswahl der injizierbaren Fehler wurde definiert.	NA013
LTC105	Prüfen, ob die Kommunikation weniger als 1 s Verzögerung verursacht.	Die Kommunikation verursacht weniger als 1 s Verzögerung.	NA014

Tabelle 10.12.: Logische Testfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
LTC106	Prüfen, ob die Verzögerung der Kommunikation möglichst konstant ist.	Die Verzögerung der Kommunikation ist so konstant wie möglich.	NA015
LTC107	Prüfen, ob der Satellitensystemzustand übersichtlich angezeigt wird.	Der Satellitensystemzustand wird übersichtlich angezeigt.	NA016
LTC108	Prüfen, ob Log-Einträge für den Anwender verständlich sind.	Anwender verstehen alle Log-Einträge.	NA017
LTC109	Prüfen, ob injizierte Fehler übersichtlich angezeigt werden.	Injizierte Fehler werden übersichtlich angezeigt.	NA018

10.2.2. Komponententestfälle

In diesem Kapitel werden mögliche Testfälle betrachtet. Jedem Testfall werden ein Bezeichner, eine Beschreibung und ein erwartetes Verhalten zugeordnet. Mit Hilfe dieser Testfälle sollen die Komponenten im Anschluss getestet werden.

Komponententests der Simulation

Im Folgenden werden die bei dem Ausführen der Komponententest der Simulation festgestellten Fehler aufgelistet. In diesem Abschnitt werden die Komponententestfälle der Simulation (Tabelle 10.13) festgelegt und beschrieben. Dabei werden die Komponententestfälle in Komponententestfälle des Interface (IF) zwischen KSP und kRPC-Server, Komponententestfälle des IF zwischen kRPC-Server und kRPC-Client, Komponententestfälle des IF zwischen kRPC-Client und Adapter-SystemC, Komponententestfälle des IF zwischen kRPC-Client und Adapter-UDP, Komponententestfälle des IF-Adapter-UDP und Komponententestfälle des IF-SystemC unterteilt.

Die Verbindung vom kRPC-Server zu KSP, sowie die Verbindung zwischen kRPC-Client und Adapter SystemC, kRPC-Client und Adapter-UDP, benötigt keine Tests, da wir über dieses Interface keine Kontrolle haben, die Schnittstellen wird risikobasiert durch die Tests der anderen Komponenten mit getestet. Die Referenzen sind somit auch nicht in diesen Komponenten zu finden und mit „/“ gekennzeichnet.

Tabelle 10.13.: Komponenten Testfälle der Simulation

Bez.	Testfall	Erwartetes Resultat	Ref.
Komponententestfälle des IF zwischen KSP und kRPC-Server			

Tabelle 10.13.: Komponenten Testfälle der Simulation

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC001	1. KSP starten. 2. kRPC-Mod im Spiel öffnen. 3. Server der Mod starten.	1. KSP startet ohne Fehlermeldungen. 2. kRPC-Mod ist im Spiel integriert und funktionstüchtig. 3. Der Server des Mods lässt sich starten (Grünes Licht wird angezeigt).	/
Komponententestfälle des IF zwischen kRPC-Server und kRPC-Client			
CTC002	1. Telnet starten. 2. Den spezifizierten Port anpingen.	1. Telnet wird erfolgreich gestartet. 2. Es wird eine Antwort vom kRPC-Server gesendet. Es wird keine Fehlermeldung gesendet. Der Port steht zur Kommunikation bereit.	/
CTC003	1. kRPC-Client sendet einen RPC an den kRPC-Servers.	1. Der kRPC-Server kann die erhaltene Nachricht interpretieren und führt einen RPC aus.	/
Komponententestfälle des IF zwischen kRPC-Client und Adapter-SystemC			
CTC004	Risikobasierter Test	Kommunikation verläuft fehlerfrei.	/
Komponententestfälle des IF zwischen kRPC-Client und Adapter-UDP			
CTC005	Risikobasierter Test	Kommunikation verläuft fehlerfrei.	/
Komponententestfälle des IF-Adapter-UDP			

Tabelle 10.13.: Komponenten Testfälle der Simulation

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC006	1. Telnet starten. 2. Auf den spezifizierten Port verbinden.	1. Telnet startet fehlerfrei. 2. Telnet stellt eine Verbindung her.	LTC001, LTC002, LTC003, LTC004, LTC005, LTC006, LTC007, LTC008, LTC009, LTC010, LTC011, LTC012, LTC013
Komponententestfälle des IF-SystemC			
CTC007	1. Kanal an den Adapter binden 2. Ein Signal auf dem Kanal senden.	1. Binding des Kanals an dem Adapter ist erfolgreich. 2. Das Signal wird erhalten und löst den assoziierten RPC aus.	LTC001, LTC004, LTC005, LTC006, LTC007, LTC008, LTC009, LTC010, LTC011, LTC012, LTC013

Komponententests der Satellitensteuerung

Im Folgenden werden die bei dem Ausführen der Komponententests der Satellitensteuerung festgestellten Fehler aufgelistet. In diesem Abschnitt werden die Komponententestfälle der Satellitensteuerung (Tabelle 10.14) festgelegt und beschrieben.

Tabelle 10.14.: Komponententestfälle der Satellitensteuerung

Bez.	Testfall	Erwartetes Resultat	Ref.
Komponententestfälle des Maneuver Schedulers			
CTC008	1. Manöver mit einem Timestamp wird geschedulet.	1. KSP spult zum Timestamp des Manövers vor. 2. Manöver wird an das Propulsion System übergeben.	

Tabelle 10.14.: Komponententestfälle der Satellitensteuerung

Bez.	Testfall	Erwartetes Resultat	Ref.
Komponententestfälle des Propulsion Systems			
CTC009	1. Maneuver zum Beschleunigen schedulen.	1. Satellit beschleunigt.	LTC015, LTC022
CTC010	1. Maneuver zur Ausrichtung schedulen.	1. Satellit richtet sich aus.	LTC022
CTC011	1. Maneuver zur Ausrichtung schedulen. 2. Maneuver zum Beschleunigen schedulen.	1. Satellit richtet sich zuerst vollständig aus. 2. Satellit beschleunigt danach.	LTC022
CTC012	1. Finales Annäherungsmanöver geplant.	1. Satellit nähert sich dem Asteroiden an und dockt an.	LTC022
Komponententestfälle der Orbitplanung			
CTC013	1. Satellit befindet sich im Warteorbit um Kerbin.	1. Satellitensteuerung plant Austrittsmanöver. 2. Satellit verlässt Orbit um Kerbin.	LTC064, LTC066
CTC014	1. Satellit befindet sich im Orbit um Kerbol.	1. Satellitensteuerung plant Manöver zum Angleichen an das Orbit des Asteroiden. 2. Satellit führt Manöver zum Angleichen durch.	LTC065
CTC015	1. Satellit befindet sich auf dem gleichen Orbit wie der Asteroid.	1. Satellitensteuerung plant Phasing. 2. Satellit führt Phasing aus.	LTC066, LTC067

Tabelle 10.14.: Komponententestfälle der Satellitensteuerung

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC016	1. Satellit befindet sich in einem Orbit.	1. Satellitensteuerung holt Informationen des Orbits.	LTC060, LTC063
CTC017	1. Satellit befindet sich in einem Orbit um Kerbin.	1. Satellitensteuerung berechnet Warteorbit. 2. Satellitensteuerung plant Manöver in diesen zu wechseln.	LTC060, LTC063
CTC018	1. Satellit und Asteroid sind gleichzeitig bei gleicher Apoapsis.	1. Satellit startet finales Annäherungsmanöver.	LTC068
Komponententestfälle des Staging			
LTC060, LTC063 CTC019	1. Satellit führt Manöver durch. 2. Ein Tank geht leer.	1. Leerer Tank wird abgeworfen. 2. Manöver wird zu Ende durchgeführt.	
CTC020	1. Satellit startet.	1. Satellit wirft sukzessive leere Tanks ab.	LTC017
Komponententestfälle des Fieldbus			
CTC021	1. Satellit befindet sich in einem Orbit.	1. Satellit fragt nach Informationen über den Orbit. 2. Über den Fieldbus werden angefragte Informationen geliefert.	LTC069
CTC022	1. Satellit plant Manöver.	1. Satellit schickt Ausrichtungs- oder Beschleunigungssignale über den Fieldbus. 2. Manöver wird ausgeführt.	LTC069

Tabelle 10.14.: Komponententestfälle der Satellitensteuerung

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC023	1. Satellitensysteme starten.	1. Satellit wählt Master für Fieldbus.	LTC069
CTC024	1. Master fällt aus.	1. Satellit wählt neuen Master.	LTC070
CTC025	1. Satellit fragt Daten an über den Fieldbus.	1. Satellit erhält angefragte Daten zurück. 2. Satellit akzeptiert oder verwirft Daten nach Prüfen der Checksumme	LTC024

Komponententests der Fehlerinjektion

Im Folgenden werden die bei dem Ausführen der Komponententests der Fehlerinjektion festgestellten Fehler aufgelistet. In diesem Abschnitt werden die Komponententestfälle der Fehlerinjektion (Tabelle 10.15) festgelegt und beschrieben. Dabei werden die Komponententestfälle in Komponententestfälle der Host-PC-Simulator-Kommunikation, Komponententestfälle der Host-PC-Nutzer-Kommunikation, Komponententestfälle der Host-PC-Komponenten, Komponententestfälle der Host-PC-Satellitensteuerung-Kommunikation, Komponententestfälle der Host-PC-Mikrocontroller-Kommunikation, Komponententestfälle der Kommunikation zwischen mehreren Mikrocontrollern, Komponententestfälle der Mikrocontroller-Komponenten, Komponententestfälle der Hardware-Software-Kommunikation und Komponententestfälle der FPGA-Komponenten unterteilt. Komponententestfälle zu denen es momentan keine Referenz gibt werden mit „/“ in der Referenzspalte markiert.

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
Komponententestfälle der Host-PC-Simulator-Kommunikation			
CTC026	Prüfen, ob das Logprotokoll von der Logprotokoll-Komponente an den Simulator gesendet wird.	Der Simulator erhält das Logprotokoll.	LTC091, LTC109
CTC027	Prüfen, ob der Simulator den Zustand der Fehlerinjektion vom Controller erhält.	Der Simulator erhält den Fehlerinjektionszustand und dadurch kann das die Vorspulen freigegeben oder blockiert werden.	/
Komponententestfälle der Host-PC-Nutzer-Kommunikation			

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC028	Prüfen, ob die Nutzerschnittstelle vom Nutzer Befehle erhalten kann.	Die Nutzerschnittstelle erhält Befehle vom Nutzer und kann diese weiter verarbeiten.	LTC092, LTC096, LTC101, LTC102
Komponententestfälle der Host-PC-Komponenten			
CTC029	Prüfen, ob das Logprotokoll von der Logprotokoll-Komponente an das Monitoring gesendet wird.	Das Monitoring erhält das Logprotokoll und kann dieses weiter verarbeiten.	LTC091, LTC109
CTC030	Prüfen, ob die Steuerungs- und Umweltsignale vom Simulator-Satelliten-Kanal an das Monitoring gesendet werden.	Das Monitoring erhält die Steuerungs- und Umweltsignale und kann diese weiter verarbeiten.	LTC088, LTC092
CTC031	Prüfen, ob der Simulator-Satelliten-Kanal Fehlerdaten vom Fehlerinjektor erhält.	Der Simulator-Satelliten-Kanal erhält Fehlerdaten und kann diese weiterleiten.	LTC080, LTC081, LTC082, LTC085
CTC032	Prüfen, ob die Monitoring-Daten vom Monitoring an die Nutzerschnittstelle gesendet werden.	Die Nutzerschnittstelle erhält die Monitoring-Daten und kann diese anzeigen.	LTC092, LTC095, LTC096, LTC097
CTC033	Prüfen, ob das Monitoring Informationen über injizierte Fehler vom Controller erhält.	Das Monitoring erhält Informationen über injizierte Fehler und kann diese weiter verarbeiten.	LTC094, LTC095
CTC034	Prüfen, ob der Controller die Befehle zur Fehlerinjektion von der Nutzerschnittstelle erhält.	Der Controller erhält die Befehle zur Fehlerinjektion von der Nutzerschnittstelle und kann diese weiter verarbeiten.	LTC080, LTC081, LTC082, LTC083, LTC084
CTC035	Prüfen, ob der Controller Informationen über injizierte Fehler vom Fehlerinjektor erhält.	Der Controller erhält Informationen über injizierte Fehler und kann diese weiter verarbeiten.	LTC095
CTC036	Prüfen, ob der Fehlerinjektor Aufträge zur Fehlerinjektion vom Controller erhält.	Der Fehlerinjektor erhält Aufträge zur Fehlerinjektion und kann diese weiterverarbeiten.	LTC080, LTC081, LTC082, LTC083, LTC085
CTC037	Prüfen, ob der Fehlerinjektor Fehler aus der Fehlerbibliothek auslesen kann.	Der Fehlerinjektor kann Fehler auslesen.	LTC080, LTC081, LTC082, LTC083, LTC085

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC038	Prüfen, ob der Fehlerinjektor die Satellitenkomponenten laden und für die Fehlerinjektion festlegen kann.	Der Fehlerinjektor kann die Satellitenkomponenten laden und zur Fehlerinjektion festlegen.	LTC085, LTC086, LTC087
Komponententestfälle der Host-PC-Satellitensteuerung-Kommunikation			
CTC039	Prüfen, ob das Logprotokoll von der Satellitensteuerung an die Logprotokoll-Komponente gesendet wird.	Die Logprotokoll-Komponente erhält das Logprotokoll und kann dieses weiter verarbeiten.	LTC090, LTC093
CTC040	Prüfen, ob der Controller Informationen über das Laden eines Zustands von der Satellitensteuerung erhält.	Der Controller erhält Informationen darüber, wenn die Satellitensteuerung einen Zustand lädt.	LTC091
CTC041	Prüfen, ob die Satellitensteuerung eine Rückmeldung vom Controller nachdem Laden eines Zustands erhält.	Die Satellitensteuerung erhält eine Rückmeldung vom Controller nachdem ein Zustand geladen wurde.	LTC091
Komponententestfälle der Host-PC-Mikrocontroller-Kommunikation			
CTC042	Prüfen, ob Fehlerinjektionsbefehle von dem Fehlerinjektor auf dem Host-PC von der Kommunikationskomponente auf dem Mikrocontroller empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC089
CTC043	Prüfen, ob Fehlerdaten von der Kommunikationskomponente auf dem Mikrocontroller von dem Fehlerinjektor auf dem Host-PC empfangen werden.	Fehlerdaten werden empfangen.	LTC089
CTC044	Prüfen, ob manipulierte Umweltsignale von dem Simulator-Satelliten-Kanal auf dem Host-PC von dem Fehlerinjektor in die Ethernet-Kommunikation auf dem Mikrocontroller empfangen werden.	Manipulierte Umweltsignale werden empfangen.	LTC080, LTC088, LTC098, LTC099
CTC045	Prüfen, ob manipulierte Steuerungsbefehle von dem Fehlerinjektor in die Ethernet-Kommunikation auf dem Mikrocontroller von dem Simulator-Satelliten-Kanal auf dem Host-PC empfangen werden.	Manipulierte Steuerungsbefehle werden empfangen.	LTC081, LTC088, LTC098, LTC099

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
Komponententestfälle der Kommunikation zwischen mehreren Mikrocontrollern			
CTC046	Prüfen, ob Daten zur Kommunikation zwischen mehreren Mikrocontrollern emfangen und gesendet werden.	Daten zur Kommunikation werden empfangen und gesendet.	/
Mikrocontroller Komponenten			
CTC047	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Host-PC von dem Fehlerinjektor in die Ethernet-Kommunikation empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC080, LTC081, LTC084, LTC085, LTC098, LTC099
CTC048	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die Ethernet-Kommunikation von der Kommunikationskomponente zum Host-PC empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099
CTC049	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Host-PC von dem Fehlerinjektor in die Satellitensteuerung auf dem Mikrocontroller empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC050	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die Satellitensteuerung auf dem Mikrocontroller von der Kommunikationskomponente zum Host-PC empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099
CTC051	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Host-PC von dem Fehlerinjektor in die Hardware-Software-Kommunikation empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC052	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die Hardware-Software-Kommunikation von der Kommunikationskomponente zum Host-PC empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC053	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Host-PC zu der Kommunikationskomponente zum FPGA weitergeleitet werden.	Fehlerinjektionsbefehle werden weitergeleitet.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC054	Prüfen, ob Fehlerdaten von der Kommunikationskomponente zum FPGA zu der Kommunikationskomponente zum Host-PC weitergeleitet werden.	Fehlerdaten werden weitergeleitet.	LTC084, LTC085, LTC098, LTC099
CTC055	Prüfen, ob Aktivierungen von den drei Fehlerinjektoren und der FPGA-Kommunikationskomponente zu den jeweiligen anderen Komponenten gesendet und empfangen werden.	Alle 12 Kommunikationsmöglichkeiten von Aktivierungen werden gesendet und empfangen.	LTC080, LTC081, LTC083, LTC084, LTC085, LTC098, LTC099
Komponententestfälle der Hardware-Software-Kommunikation			
CTC056	Prüfen, ob Fehlerinjektionsbefehle von dem Kommunikationskomponente auf dem Mikrocontroller von der Kommunikationskomponente auf dem FPGA weitergeleitet werden.	Fehlerinjektionsbefehle werden weitergeleitet.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC057	Prüfen, ob Fehlerdaten von der Kommunikationskomponente auf dem FPGA von dem Kommunikationskomponente auf dem Mikrocontroller weitergeleitet werden.	Fehlerdaten werden weitergeleitet.	LTC084, LTC085, LTC098, LTC099
CTC058	Prüfen, ob manipulierte Software-Hardware-Daten von dem Fehlerinjektor in die SW-HW-Komminikation auf dem Mikrocontroller von dem von dem Fehlerinjektor in die SW-HW-Komminikation auf dem FPGA empfangen werden.	Manipulierte Software-Hardware-Daten werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC059	Prüfen, ob Hardware-Software-Daten von dem Fehlerinjektor in die HW-SW-Komminikation auf dem FPGA von der Satellitensteuerung auf dem Mikrocontroller empfangen werden.	Hardware-Software-Daten werden empfangen.	LTC084, LTC085, LTC098, LTC099
Komponententestfälle der FPGA-Komponenten			
CTC060	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Mikrocontroller von dem Fehlerinjektor in die SW-HW-Kommunikation empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC061	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die SW-HW-Kommunikation von der Kommunikationskomponente zum Mikrocontroller empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099
CTC062	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Mikrocontroller von dem Fehlerinjektor in die HW-SW-Kommunikation empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099
CTC063	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die HW-SW-Kommunikation von der Kommunikationskomponente zum Mikrocontroller empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099
CTC064	Prüfen, ob Fehlerinjektionsbefehle von der Kommunikationskomponente zum Mikrocontroller von dem Fehlerinjektor in die Satellitensteuerung auf dem FPGA empfangen werden.	Fehlerinjektionsbefehle werden empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099

Tabelle 10.15.: Komponententestfälle der Fehlerinjektion

Bez.	Testfall	Erwartetes Resultat	Ref.
CTC065	Prüfen, ob Fehlerdaten von dem Fehlerinjektor in die Satellitensteuerung auf dem FPGA von der Kommunikationskomponente zum Mikrocontroller empfangen werden.	Fehlerdaten werden empfangen.	LTC084, LTC085, LTC098, LTC099, LTC100
CTC066	Prüfen, ob Aktivierungen von den drei Fehlerinjektoren und der Mikrocontroller-Kommunikationskomponente zu den jeweiligen anderen Komponenten gesendet und empfangen werden.	Alle 12 Kommunikationsmöglichkeiten von Aktivierungen werden gesendet und empfangen.	LTC083, LTC084, LTC085, LTC098, LTC099

Teil III.

Spezifikation

11. Simulator

In diesem Kapitel wird die Spezifikation des Teilsystems Simulator vorgestellt. Zunächst wird die Simulatorschnittstelle erläutert. Anschließend wird der Satellit betrachtet und zum Abschluss das Szenario, in dem die Mission durchgeführt wird.

11.1. Schnittstelle zur Simulation

Für die erfolgreiche Durchführung des Projekts ist es notwendig, die Kommunikation zwischen mehreren Komponenten herzustellen. Im folgenden Kapitel soll die entwickelte Struktur des Netzwerks vorgestellt werden.

Eine der gestellten Anforderungen an das Projekt ist die Verwendung der Ethernet-Schnittstelle (siehe Kapitel 1.2). In tatsächlich durchgeführten Missionen, wie zum Beispiel der Rosetta Mission, werden im Gegensatz dazu Bussysteme wie SpaceWire verwendet [2]. Um den Anforderungen und dem Szenario gerecht zu werden, wird ein Kompromiss zwischen diesen beiden Welten eingegangen. Hierzu wird das Netzwerk auf zwei unterschiedlichen Ebenen betrachtet:

physikalisches Netzwerk Hierunter wird das Netzwerk verstanden das in dem Versuchsaufbau verwendet wird, um die Kommunikation der verwendeten Komponenten (Zedboard, sowie die Computer für Fehlerinjektion und Simulation) herzustellen.

logisches Netzwerk Hierunter wird das Netzwerk verstanden, wie es in dem Satelliten vorgefunden würde, für die die Satellitensteuerung entwickelt wird.

Für diese zwei Ebenen gelten unterschiedliche Gütekriterien. Das logische Netzwerk soll möglichst exakt der Netzwerkinfrastruktur entsprechen, wie sie in einem realen Satelliten vorgefunden wird. Auf der anderen Seite muss das physikalische Netzwerk der Vorgabe der Ethernet Schnittstelle genügen und darüber hinaus der Satellitensteuerung die gleiche Netzwerkschnittstelle liefern, wie sie in dem logisch Netzwerk vorliegt.

11.1.1. Physikalisches Netzwerk

Das angestrebte Netzwerk wird in der Abbildung 11.1 dargestellt. Es besteht aus den, für die Satellitensteuerung bereitgestellten, ZedBoards, die untereinander durch einen *managed switch* verbunden sind. Der Switch wurde vor allem gewählt, um die maximale Bandbreite der Ethernet-Schnittstelle nicht einzuschränken. Zeitgleich erlaubt er es, durch *port mirroring* den über ihn laufenden Datenverkehr selektiv an einen Beobachter weiter zu leiten. Diese Eigenschaft erleichtert das Debuggen der Netzwerkverbindung und gibt dem Monitoring der Fehlerinjektion die Möglichkeit, den Datenverkehr zu analysieren.

Der Host-PC der Fehlerinjektion ist mit zwei Netzwerkkarten ausgestattet. Dabei wird eine Verbindung für die Kommunikation mit den ZedBoards und der Simulation, sowie eine zum Abhören des Datenverkehrs auf dem Switch verwendet.

11.1.2. Logisches Netzwerk

Um den gegebenen Anforderungen des physikalischen Netzwerkes gerecht zu werden, gehen wir für das logische Netzwerk davon aus, dass ebenfalls eine Ethernet-Schnittstelle bereitgestellt wird. Für die Kommunikation mit Sensoren und Aktoren wird ein *Feldbus* verwendet

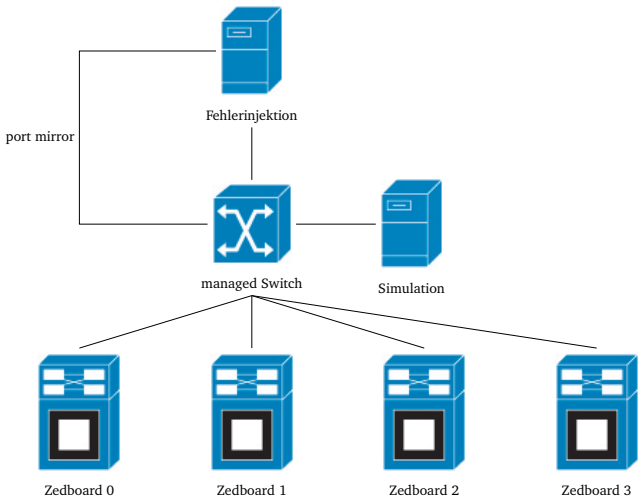


Abbildung 11.1.: Karte der physikalischen Netzwerkstruktur

der über eine *Bridge* erreichbar ist. Es wird davon ausgegangen, dass der Feldbus, sowie die Ethernet-Netzwerkstruktur, fehlertolerant ausgelegt sind, sodass kein Single Point Of Failure (SPOF) in der Netzwerkstruktur existiert.

Im logischen Netzwerk (siehe Abbildung 11.2) wird davon ausgegangen, dass Switches, Bridges sowie Logging-Server redundant ausgelegt wurden und sich in einer voll vermaschten Netzwerktopologie befinden. Durch die Limitierungen der Hardware können die ZedBoards lediglich mit einer Ethernet Verbindung in dieses Netzwerk eingebunden werden. Die Kommunikation zwischen Aktoren, Sensoren und Steuerung ist damit solange gesichert, wie ein Zedboard, sein angeschlossener Switch, wie auch der angesprochene Aktor, beziehungsweise Sensor funktionsfähig sind.

11.1.3. Adapter

Durch den verwendeten modellgetriebenen Entwicklungsprozess ist es notwendig nicht nur eine Verbindung zwischen der Simulation und dem Endprodukt, sondern auch dem Modell der Satellitensteuerung herzustellen. Es werden hierfür zwei Adapter bereitgestellt durch die, die jeweils benötigte Verbindung hergestellt wird.

Der bereitgestellte Funktionsumfang ist unabhängig von dem verwendeten Adapter, sodass sich lediglich das Interface ändert über das dieser Bereitgestellt wird.

UDP-Adapter

Wie im Kapitel 11.1.2 beschrieben, findet die Kommunikation der Satellitensteuerung über Ethernet und UDP statt. Der UDP-Adapter stellt diese Kommunikations-Schnittstelle mit der Simulation und ermöglicht damit im späteren Verlauf des Projektes das Testen der Satellitensteuerung auf den ZedBoards.

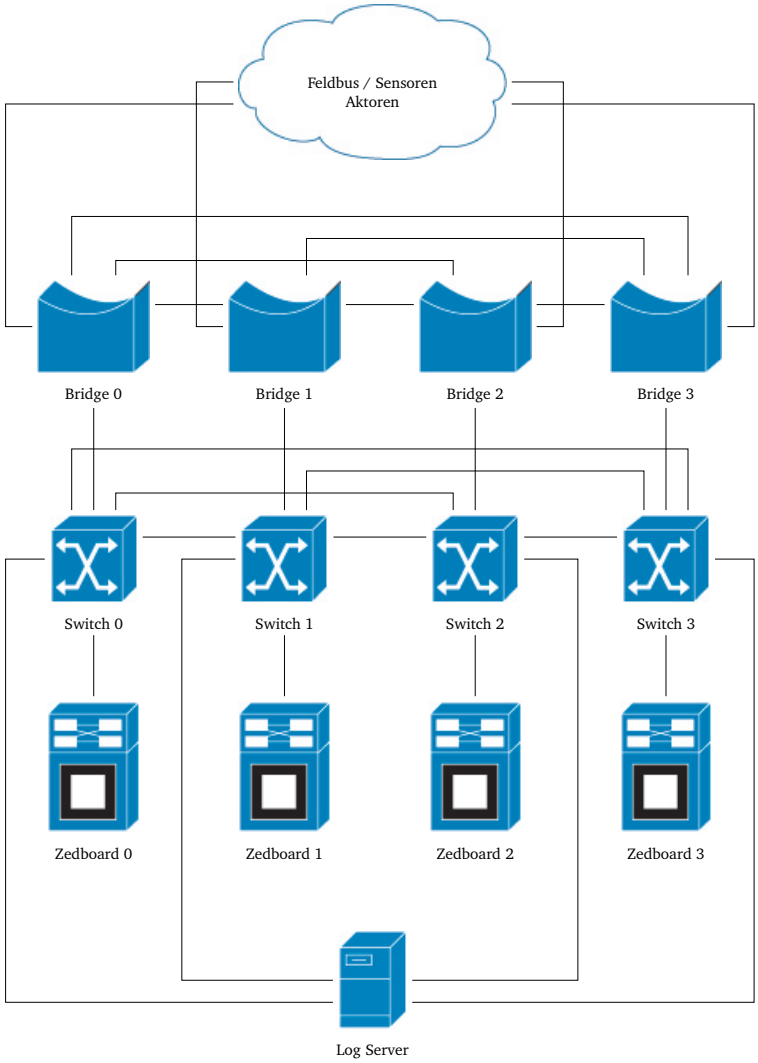


Abbildung 11.2.: Karte der logischen Netzwerkstruktur

SystemC-Adapter

Auf dem SystemC-Adapter kann auf zwei unterschiedlichen Ebenen zugegriffen werden, um einen geregelten Abstieg in der Kommunikationsverfeinerung zu ermöglichen. Es stehen folgende Möglichkeiten bereit:

Komponentenebene: Es ist möglich, mit jeder physikalisch im Satellit vorhandenen Komponente direkt zu kommunizieren.

Busebene: Es wird ein Bus simuliert, in dem angesprochene Komponenten durch eine mitgegebene Adresse referenziert werden.

Um die Unabhängigkeit der Satellitensteuerung von kRPC zu gewährleisten werden alle kRPC eigenen Daten- und Objekttypen durch eigene ersetzt. So kann der kRPC-Client nach dem Umstieg auf den UDP-Adapter auf Seiten der Satellitensteuerung ohne weiteren Aufwand aus dem Bauprozess entfernt werden.

Übersicht der implementierten Adapter

Im Allgemeinen werden alle als physikalisch im Satelliten vorhandene Komponenten in einer eigenen Adapter-Klasse gekapselt. Über diese lassen sich alle unterstützten Variablen, für eine darin gekapselte Satelliten-Komponente manipulieren beziehungsweise abfragen. Zu diesem Zweck stellen die Adapter eine Reihe von Ports bereit, durch die die notwendigen Funktionsaufrufe, die mittels kRPC durchgeführt werden, gekapselt.

Jede gekapselte Funktion X besitzt einen Kanal X_cmd durch den der Funktionsaufruf ausgelöst wird. Abhängig davon ob die gekapselte Funktion eine Rückgabe besitzt, ist der Funktion ein weiterer Kanal X_ret zugeordnet, über den der Rückgabe-Wert an den Auslöser des Funktionsaufrufs zurückgegeben wird.

In dieser Form stehen folgende Adapter-Klassen bereit:

- decoupler_adapter
- engine_adapter
- fairing_adapter
- launch_clamp_adapter
- light_adapter
- sas_adapter
- solar_panel_adapter

Neben diesen Bauteil-Orientierten Adaptern existieren folgende Adapter, die darüber hinausreichende Funktionalitäten implementieren:

ksp_adapter: In diesem Adapter werden alle für die Abdeckung des Funktionsumfangs notwendigen Adapter instantiiert und bereitgestellt. Sofern mehrere Instanzen eines Adapters benötigt werden, werden diese in einem sc_vector abgelegt. Darüber hinaus werden in diesem Adapter zwei weitere Ports bereitgestellt, durch das Verhalten der im Abschnitt 11.2 eingeführten Bridges simuliert wird.

control_adapter: Mit Hilfe dieses Adapters kann der Satellit gesteuert werden. Die Steuerung entspricht hier dem, was anderweitig auch mit der Tastatur möglich ist.

vessel_adapter: Dieser Adapter liefert Informationen über den Satelliten im Allgemeinen.

space_center_adapter: Alle Informationen, die nicht autonom von dem Satelliten selbst beschafft werden können, werden über diesen Adapter bereitgestellt. Man kann es als Kommunikation mit der Bodenstation verstehen, die beispielsweise Informationen über den Orbit von verschiedenen Himmelskörpern vorhält.

11.1.4. Verwendetes Protokoll

Für die Entwicklung des SystemC-Modells wurde zunächst auf das Plugin kRPC [44] zurückgegriffen, welches auf Basis von TCP/IP und einem RPC-Protokoll arbeitet. Im weiteren Projektverlauf wurde ein eigenes, UDP/IP basiertes Protokoll entwickelt.

Tabelle 11.1.: Eine Liste aller Adapter, die verwendet werden, und die zugehörigen Präfixe, die in der Adresse verwendet werden. *Asteroid* und *Satellite* weisen beide auf einen Vessel Adapter. Jedes Präfix wird als Hexadezimalzahl angegeben

Adapter	Präfix
Decoupler	0x01
Engine	0x02
Launch Clamp	0x03
Lights	0x04
Solar Panel	0x05
Satellite	0x06
Asteroid	0x07
Control	0x08
Space Center	0x09
SAS	0x0A
Fairing	0x0B

Adressierung

Um ein einfaches Routing und eine möglichst einfache Erweiterbarkeit zu gewährleisten wurde die Adresse in zwei Komponenten unterteilt von jeweils einem Byte Länge. Das erste Byte einer Adresse gibt den Typ des Adapters an, der durch die Adresse referenziert wird. Eine Liste dieser Präfixe ist in der Tabelle 11.1 zu finden. Das zweite Byte der Adresse gibt den Index an, in dem das zugehörige Objekt in der Adapter Datenstruktur gehalten wird. Die Position innerhalb dieser Datenstruktur ist abhängig davon, wo sich das verwaltete Objekt in dem kontrollierten *Vessel* befindet. Für Adapter, die nur ein einziges mal vorhanden sein können, wird immer der Index 0 angenommen. Wird beispielsweise versucht, das dritte *Light* zu *Adress* zu Adressieren so kann dieses über die Adresse 0x0403 erreicht werden.

Durch diese dynamische Adressierung ist es möglich ohne weitere eingriffe den gesteuerten Satelliten um neue Komponenten zu erweitern. Ebenso ist es möglich durch die Implementierung neuer Adapterklassen den unterstützten Funktionsumfang zu erweitern.

11.2. Der Satellit in KSP

Der Satellit muss, zum Erreichen des Missionsziels, in der Lage sein, an einen Asteroiden anzudocken. Dafür wird ein Greifarm verwendet, welcher dem Spiel durch die Mod „Astroid Day“ hinzugefügt wurde. Des Weiteren muss der Satellit in der Lage sein, nach dem Andocken an den Asteroiden diesen von seiner aktuellen Umlaufbahn zu bringen. Dafür wird eine starke Antriebsdüse benötigt, die genug Kraft aufbringen kann, um die Aufgabe zu erfüllen.

Zum Steuern der Satelliten in der Nähe des Asteroiden sind mehrere Schubdüsen (engl. Thruster) angebracht. Mit ihnen sind feinere Manöver möglich, welche nötig sind, um sich an den Asteroiden anzunähern. Für andere Manöver, welche den Satelliten ausrichten sollen, werden die Gyroskope innerhalb des Satelliten verwendet. Diese werden mit Strom betrieben. Damit der Vorrat an elektrischer Energie nicht aufgebraucht werden kann, wurden zwei Solarpanels an dem Satelliten angebracht, welche ständig in Richtung Kerbol auszurichten sind. Falls es notwendig ist, für längere Zeit im Schatten eines Objektes zu Manövrieren, wurde eine größere, wieder-aufladbare Batterie unter dem Satelliten angebracht. Diese wird voll aufgeladen während die Solarpanels ausgefahren sind.

Für den Weg in die Umlaufbahn von Kerbol wird eine Trägerrakete benötigt. Diese besteht im Grundsatz aus vier großen Tanks und einer Antriebsdüse. Am oberen Ende der Trägerrakete wurde eine beschützende Hülle angebracht in der sich der eigentliche Satellit befindet. Diese wird so angepasst, dass er möglichst aerodynamisch ist und dabei möglichst wenig wiegt. Die Hülle wird benötigt, damit die Einzelteile welche an den Satelliten angebracht wurden nicht beim Start in der Atmosphäre verglühen oder durch einige Teile ein größerer Windwiderstand auf einer Seite der Rakete entsteht was die Rakete instabil machen könnte. Am unteren Ende der Trägerrakete befindet sich die zentrale Antriebsdüse, sowie sechs weitere Antriebsdüsen vom gleichen Typ, welche mit einem seperaten Tank verbunden sind. Diese sechs Tanks, inklusive Antriebsdüsen, lassen sich zu jeder Zeit von der Hauptrakete (zentrale Antriebsdüse mit direkt verbundenen Tanks) trennen. Der Aufbau der Rakete ist in Abbildung 11.3 zu sehen

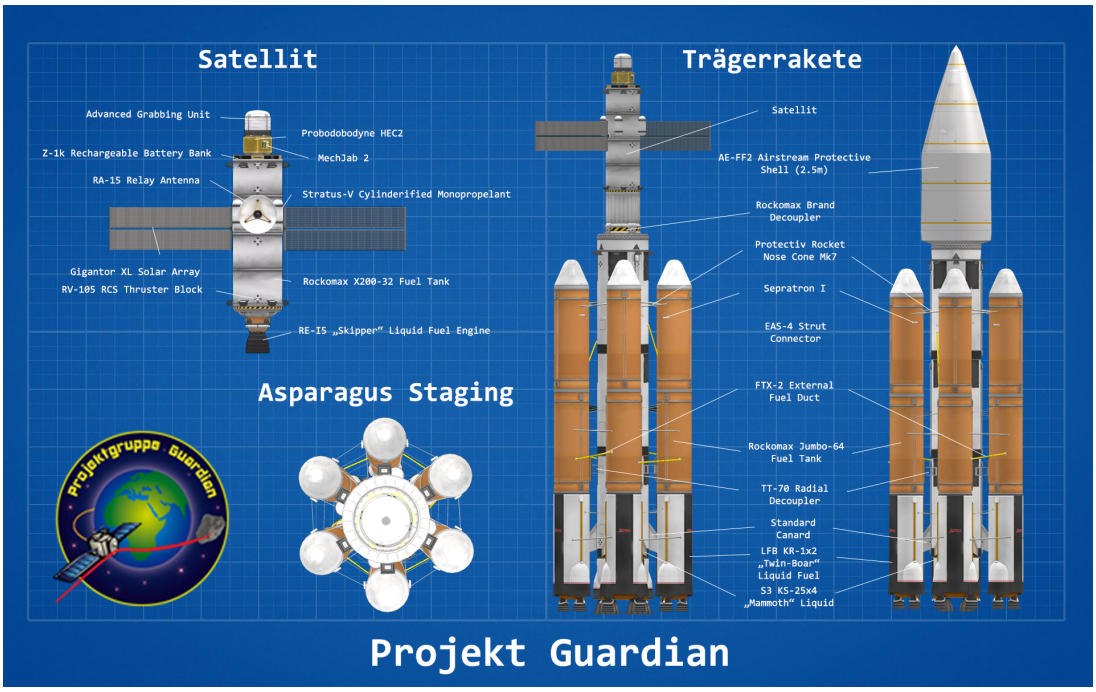


Abbildung 11.3.: Blaupause zum Satelliten und der Trägerrakete der Mission Guardian.

Um mit so wenig Treibstoff wie möglich in den Orbit zu kommen, sind die Tanks durch Treibstoffleitungen miteinander verbunden. Das Prinzip des verwendeten Stagings nennt sich Asparagus-Staging. Dabei wird zunächst der Treibstoff aus zwei sich gegenüberliegenden Tanks verbraucht, um alle Antriebsdüsen zu betreiben. Sobald die beiden Tanks leer sind, werden diese von der Hauptrakete getrennt. Somit wird beim Aufsteigen Gewicht gespart und erst beim Abtrennen der letzten zwei Tanks der Kraftstoff der Hauptrakete verwendet. Dadurch wird die Energie, die benötigt wird um den Orbit zu verlassen, mit jeder abgetrennten Stage geringer und minimiert den Treibstoffverbrauch insgesamt. An dieser Stelle ist noch zu erwähnen, dass das Asparagus-Staging momentan noch eine Theorie ist und aktiv für die Raumfahrt erforscht wird.

11.3. Das Missionsszenario

Das Szenario beschreibt die auszuführenden Schritte zur Durchführung der Mission. Diese ist in Phasen aufgeteilt und beschreibt die Aufgabe, einen auf Kollisionskurs mit Kerbin befindlichen Asteroiden vom Kurs abzubringen. Dazu soll ein unbemannter Satellit in einen Orbit, ähnlich zu dem des Asteroiden, gebracht werden. Es folgt nun eine phasenweise Beschreibung des Missionsszenarios. Dafür muss die Vorbedingung gelten, dass sich die Trägerrakete, inklusive des Satelliten, auf dem Launchpad befindet und startbereit ist.

Phase 0: Startvorbereitung

Die Mission der Projektgruppe ist in mehrere Phasen unterteilt, welche nacheinander durchgeführt werden. Die Trägerrakete befindet sich auf dem Launchpad und ist startbereit. Die Satellitensteuerung hat den optimalen Zeitpunkt für den Start berechnet, um einen möglichst ressourcensparenden Kurs zum Asteroiden fliegen zu können. Die Asteroiden der Mission werden im Simulator während des gesamten Szenarios getrackt.

- Schub auf 100% und Stability Augmentation System (SAS) aktiviert.

Phase 1: Flug bis 10km

Beim Flug bis auf $10km$ sollte die Trägerrakete senkrecht bei einer maximalen Geschwindigkeit von $280m/s$ steigen, um den Widerstand der Atmosphäre zu minimieren. Bei diesem Vorgang werden die nicht mehr benötigten Stages abgeworfen, um das Gewicht zu verringern. Bei einer Höhe von ca. $10km$ wird die Trägerrakete um 30° nach Westen geneigt. Bei diesem Manöver muss darauf geachtet werden, dass die Trägerrakete nicht anfängt zu rotieren, um unnötige Korrekturen zu vermeiden.

- Senkrecht steigen
- Maximale Steiggeschwindigkeit beträgt $280m/s$, Throttle ist dementsprechend anzupassen.
- Stage abwerfen wenn Tank leer
- Bei einer Höhe von ca. 10km um 30° Richtung W neigen
- Rotationen der Rakete vermeiden
- Stage abwerfen wenn Tank leer

Phase 2: Flug bis 36km

Bei einer Höhe von 36km wird die obere Abschirmung abgesprengt, um Gewicht einzusparen und die derzeit verwendete Stage läuft auf 100% Schub. Zur Stromversorgung müssen in dieser Phase die Solarpanel ausgefahren werden (Solarpanels nur ausfahren, wenn nicht erneut in die Atmosphäre eingetreten wird).

- Stage für die obere Abschirmung absprengen
- Schub auf 100%
- Solarpanel ausfahren
- Stage abwerfen wenn Tank leer

Phase 3: Flug bis Apoapsis bei ca. 100km

Beim Erreichen einer Apoapsis von circa 100km den Schub auf 0% schalten.

- Throttle auf 0%
- Stage abwerfen wenn Tank leer
- Manöver für verlassen der Umlaufbahn von Kerbin erstellen

Phase 4: Manöverpunkt erreicht

Der optimale Punkt für ein Manöver zum Verlassen des Orbits von Kerbin ist die Apoapsis. An dieser Stelle muss das Manöver den Satelliten in den Orbitn von Kerbol bringen.

- Manöver zum Verlassen der Umlaufbahn von Kerbin erstellen
- Stage abwerfen wenn Tank leer

Phase 5: Anpassung an den Zielorbit

Befindet sich der Satellit im Orbit um Kerbol, sollte dieser nach einiger Zeit die Bahn des Zielasteroiden kreuzen. Zu diesem Zeitpunkt muss der Orbit des Satelliten so angepasst werden, dass sich dieser anschließend auf dem gleichen Orbit wie der Asteroid befindet. Die Berechnungen für dieses Manöver können bereits auf der langen Reise zuvor durchgeführt werden, jedoch ist auch hier wieder eine sehr genaue zeitliche und korrekte Ausführung vonnöten.

- Manöver zum Angleichen der Flugbahn erstellen
- Stage abwerfen wenn Tank leer

Phase 6: Annäherung an das Ziel

Der Satellit sollte sich in dieser Phase bereits nah am Asteroiden befinden und nur noch kleine Korrekturen ausführen müssen. Wie einleitend beschrieben, gibt es in der Simulation das Problem, dass Asteroiden keine Gravitation besitzen. Daraus folgt das Problem, dass der Satellit in keinen Orbit um den Asteroiden gehen, sondern nur „neben“ diesem fliegen kann.

- Über Manöver an den Asteroiden annähern.
- Relative Geschwindigkeit zum Asteroiden niedrig halten.
- RCS einschalten
- Schub auf 0%
- Annähern mit dem Thruster
- Grabber Unit ausfahren

Phase 7: Greifen des Asteroiden

Der Greifer, welcher vorne am Satelliten montiert ist, muss so positioniert werden, dass er das erste ist was den Asteroiden berührt. Die Geschwindigkeit mit der der Satellit auf den Asteroiden trifft darf nicht größer als 5m/s sein, da ansonsten die darunter gelegenen Strukturen zerstört werden können. Der Greifer greift automatisch nach den Asteroiden, sobald diese in Kontakt kommen. Danach ist der Satellit mit dem Asteroiden verbunden.

- Mit der Grabber Unit vorraus und einer Maximalgeschwindigkeit von 5m/s gegen den Asteroiden manövrieren

- Asteroid wird gegriffen

Phase 8: Verschieben des Asteroiden

Zum Verschieben des Asteroiden muss ein Manöver erstellt werden, was möglichst kraftstoffsparend den Asteroiden von seiner jetzigen Umlaufbahn abbringt. Wurde ein Manöver erstellt muss der Schub langsam erhöht werden um ein auseinanderbrechen des Satelliten zu verhindern.

- RCS deaktivieren
- Manöver zum verschieben des Asteroiden erstellen
- Schub linear auf 100% erhöhen

Phasenübergreifende Operationen

Zu den phasenübergreifenden Operationen zählen Payloadoperationen, die ab der Phase 2 angewendet werden können. Ebenfalls kann der Satellit während der gesamten Mission Fehlertolenanzmechanismen anwenden.

12. Satellitensteuerung

Bei der Spezifikation der Satellitensteuerung werden verschiedene Konzepte vorgestellt, die auf die Einheiten der Satellitensteuerung zurückzuführen sind. Zunächst wird das Fortbewegungskonzept erläutert, anschließend die Konzepte zur Fehlertoleranz, zum Ressourcenmanagement, zur Orbitplanung und zur Kommunikation.

12.1. Fortbewegungskonzept

In diesem Abschnitt wird das Fortbewegungskonzept des Satelliten beschrieben. Es werden die Parameter, die bei der Konstruktion einer Rakete notwendig sind, eingeführt. Des Weiteren werden Abschätzungen über die benötigte Leistung zum Erfüllen der Mission gemacht. Die beschriebenen Formeln sind Vereinfachungen der Realität und beschreiben die Simulationsumgebung.

12.1.1. Kenngrößen bei der Konstruktionen von Raketen

Bei der Konstruktion des Antriebs für ein Raumfahrzeug gilt es drei Kenngrößen besonders zu beachten. In den weiteren Unterkapiteln werden diese Kenngrößen vorgestellt und ihre Bedeutung bei der Auslegung des Antriebs erklärt.

Thrust-to-Weight Ratio

Das Thrust-to-Weight Ratio beschreibt das Verhältnis von Schub zu Gewicht. Durch F_T wird der Schub der Antriebe beschrieben. Damit die Rakete an Höhe gewinnen kann, muss der Schub größer sein als die Raketenmasse (m), multipliziert mit der Gravitationsbeschleunigung (g). Sollte das Thrust-to-Weight Ratio kleiner oder gleich eins sein, wird die Rakete nicht von der Startrampe abheben können [39].

$$TWR = \frac{F_T}{m * g} \tag{12.1}$$

Spezifischer Impuls

Der spezifische Impuls beschreibt die Effizienz eines Antriebs. Der Wert gibt das Verhältnis von Schubkraft (F_T) zu Treibstoffverbrauch über die Zeit, hier mit dem Verlust der Masse über die Zeit ($\dot{m}$), wieder[39].

$$I_{sp} = \frac{F_T}{\dot{m}} \tag{12.2}$$

Delta-V

Mit dem Parameter ΔV wird die maximal mögliche Geschwindigkeitsänderung einer Rakete beschrieben. ΔV ist ein Wert, um die Leistungsfähigkeit einer Rakete zu bestimmen. Zur Berechnung wird die Raketengrundgleichung herangezogen. Hierfür wird der spezifische Impuls multipliziert mit dem natürlichen Logarithmus aus der Masse der Rakete mit vollen Treibstofftanks (m_{total}) zum Verhältnis der Masse der Rakete mit leeren Treibstofftanks (m_{dry}) [39].

$$\Delta v = I_{sp} * ln\left(\frac{m_{total}}{m_{dry}}\right) \tag{12.3}$$

12.1.2. Aufstieg in den Orbit von Kerbin

Für den Aufstieg in einen Orbit um Kerbin wird die Berechnung des ΔV in erster Näherung mit der Vis-Viva-Gleichung vorgenommen. Die Vis-Viva-Gleichung bestimmt die Geschwindigkeit von Körpern auf Keplerbahnen um einen Himmelskörper. Der Standardgravitationsparameter wird mit μ beschrieben, während r der Abstand des Körpers zum Gravitationszentrum ist. Die Länge der großen Halbachse wird mit a bezeichnet [39].

$$v^2 = \mu * \left(\frac{2}{r} - \frac{1}{a}\right) \tag{12.4}$$

In den folgenden Berechnungsschritten werden Verluste durch die Atmosphäre und Anziehung von Kerbin außer Acht gelassen. In der Berechnung wird angenommen, dass wir auf einem Orbit um den Mittelpunkt von Kerbin sind. Wir wollen diesen Orbit auf 80 km über der Oberfläche von Kerbin verschieben.

Die Geschwindigkeit des Satelliten beim Start berechnet sich mit der folgenden Gleichung. Der Planet Kerbin besitzt am Äquator einen Radius von 600 km und eine Rotationsperiode von 6 Stunden [88]. Durch folgende Formel kann somit die bereits vorhandene Geschwindigkeit der Rakete durch die Rotation des Planeten berechnet werden.

$$\frac{2\pi * r}{R} \tag{12.5}$$

$$\frac{2\pi * 600000m}{21600s} \approx 174,5\frac{m}{s} \tag{12.6}$$

Das Modell nimmt an, dass die Geschwindigkeit sich spontan auf einen gegebenen Wert einstellen kann. Unter Zuhilfenahme dieser Vereinfachung wird die Geschwindigkeit für einen elliptischen Orbit mit einer Apoapsis von 80 km und einer Periapsis auf der Oberfläche von Kerbin berechnet.

$$2500,8\frac{m}{s} = \sqrt{3,5316000 * 10^{12}\frac{m^3}{s^2} * \left(\frac{2}{600000m} - \frac{1}{640000m}\right)} \tag{12.7}$$

Vom Ergebnis kann die bereits gegebene Geschwindigkeit durch die Rotation von Kerbin abgezogen werden.

$$2500,8\frac{m}{s} - 174,5\frac{m}{s} = 2326,3\frac{m}{s} \tag{12.8}$$

Auf der Apoapsis erreicht der Satellit, somit die folgende Geschwindigkeit:

$$2206,6\frac{m}{s} = \sqrt{3,5316 * 10^{12}\frac{m^3}{s^2} * \left(\frac{2}{680000m} - \frac{1}{640000m}\right)} \tag{12.9}$$

Um auf einen kreisförmigen Orbit auf 80 km Höhe zu kommen, wird die Geschwindigkeit für diesen Orbit berechnet. Auf einem kreisförmigen Orbit vereinfacht sich die Vis-Viva-Gleichung wie folgt($r = a$).

$$v^2 = \frac{\mu}{r} \tag{12.10}$$

$$2278,9\frac{m}{s} = \sqrt{\frac{3,5316 * 10^{12}\frac{m^3}{s^2}}{680000m}} \tag{12.11}$$

Die Differenz zwischen den Ergebnissen der beiden Gleichungen ergibt das benötigte ΔV um in den kreisförmigen Orbit zu kommen.

$$2278,9\frac{m}{s} - 2206,6\frac{m}{s} = 72,3\frac{m}{s} \tag{12.12}$$

Somit wird ein ΔV von $2398,6\ \frac{m}{s}$ für einen zirkulären Orbit in 80 km Höhe um Kerbin benötigt.

$$72,3\frac{m}{s} + 2326,3\frac{m}{s} = 2398,6\frac{m}{s} \tag{12.13}$$

Für den Start aus der Atmosphäre von Kerbin wird der sogenannte Gravity Turn eingesetzt. Beim Gravity Turn wird versucht, die Energieverluste durch die Atmosphäre und die Schwerkraft so gering wie möglich zu halten. Wird die Rakete im rechten Winkel zur Oberfläche von Kerbin auf die Umlaufbahn geschossen, sind die Energieverluste durch die Schwerkraft maximal. Wird die Rakete jedoch auf einer Kurve in die Umlaufbahn geschossen, befindet sich die Rakete für einen längeren Zeitraum in der Atmosphäre und die Energieverluste durch den Luftwiderstand erhöhen sich. Es gilt eine optimale Flugkurve zu finden, um die Verluste minimal zu halten.

Die Kraft des Luftwiderstands berechnet sich aus der atmosphärischen Dichte (ρ), der Geschwindigkeit der Rakete (v), dem Widerstandskoeffizienten der Rakete (d) und der Querschnittsfläche der Flugrichtung (A) [87].

$$F_D = 0,5 * \rho * v^2 * d * A \tag{12.14}$$

Die atmosphärische Dichte ist abhängig vom atmosphärischen Druck (p) und der Temperatur (T). Jedoch ist der Einfluss der Temperaturänderungen für die Auslegung der Berechnung zu vernachlässigen [87].

$$\rho = \frac{p}{R * T} \tag{12.15}$$

Um einen energieeffizienten Aufstieg in den Orbit zu ermöglichen, ist die Geschwindigkeit der Rakete in den verschiedenen Flugphasen von Bedeutung. Dazu sollte die Rakete sich mit ähnlicher Geschwindigkeit, wie die der Grenzgeschwindigkeit bewegen. Die Grenzgeschwindigkeit beschreibt die Geschwindigkeit eines fallenden Objekts, wenn die Kräfte des Luftwiderstands und der Schwerkraft gleich sind.

$$F_G = m * \frac{GM}{r^2} \tag{12.16}$$

Die Schwerkraft berechnet sich aus der Masse der Rakete (m), der Gravitationskonstante (G), der Masse des Planeten (M) und dem Radius (r) zum Mittelpunkt des Planeten. Um die

optimale Geschwindigkeit zu bestimmen, wird die Geschwindigkeit gesucht bei der $F_D = F_G$ gilt.

Ohne die Parameter kann an dieser Stelle keine genaue Berechnung durchgeführt werden, jedoch werden Erfahrungen des benötigten ΔV in der KSP Community in ΔV-Maps festgehalten. Daher ist für einen Aufstieg in einen 80 km Orbit um Kerbin von einem ΔV-Bedarf von $3400\ \frac{m}{s}$ auszugehen.

12.1.3. Orbittransfer von Kerbin zu Kerbol

In dieser Phase der Mission müssen zwei Anpassungen vorgenommen werden. Zum einen muss die Inklination des Orbits an die des Orbits des Asteroiden angepasst werden. Zum anderen muss das Raumfahrzeug auf Fluchtgeschwindigkeit von Kerbin gebracht werden. Zum Anfang dieser Phase befindet sich das Fahrzeug in einer Höhe von ca. 80 km. Die Geschwindigkeit in diesem Orbit beträgt ca. 2278.9316 $\frac{m}{s}$.

$$\sqrt{\frac{3.5316 * 10^{12} m^3 s^{-2}}{680000m}} \approx 2278.9316 \frac{m}{s} \tag{12.17}$$

Die maximal nötige Änderung der Inklination liegt bei 90 Grad. Somit liegt das maximale ΔV-Budget zur Inklinationsänderung bei 3878.3019 m/s.

$$\Delta V = 2 * 2278.9316 \frac{m}{s} * sin(\frac{90^\circ}{2}) \approx 3878.3019 \frac{m}{s} \tag{12.18}$$

Des Weiteren werden ΔV-Werte von 943.964 m/s.

$$\Delta V_{escape} = 2278.9316 \frac{m}{s} * (\sqrt{2} - 1) \approx 943.964 \frac{m}{s} \tag{12.19}$$

Insgesamt liegt das ΔV-Budget für diese Phase bei ungefähr 3222.8956 m/s.

$$\Delta V_{gesamt} = 943.964 \frac{m}{s} + 2278.9316 \frac{m}{s} = 3222.8956 \frac{m}{s} \tag{12.20}$$

12.1.4. Veränderung des Asteroiden Orbits

Sobald der Asteroid vom Satelliten gegriffen wurde, werden sie als ein Objekt betrachtet. Durch die Vis-Viva-Gleichung kann daher das benötigte ΔV zur Veränderung des Asteroidenorbits berechnet werden. Dabei besteht eine Abhängigkeit zwischen der Masse des Asteroiden und der notwendigen Orbitanpassung.

12.1.5. Orientierung des Satelliten

Um die Orientierung eines Satelliten in der Simulationsumgebung zu verändern, gibt es zwei mögliche Umsetzungen. Eine Umsetzung ist das RCS, das aus kleinen Schubdüsen besteht. Durch das Ansteuern der Düsen lässt sich der Satellit verhältnismäßig schnell in die gewünschte Orientierung bringen. Der Nachteil des RCS ist der Verbrauch von Monopropellant-Treibstoff (siehe Kapitel 12.3).

Eine weitere Möglichkeit ist ein Reaktionsrad, das durch ein erzeugtes Drehmoment die Orientierung des Satelliten verändern kann. Reaktionsräder verbrauchen die Energieeinheit

Electric Charge. Dies ist in sofern günstig, da diese Energieform durch den Einsatz von Solarpaneelen gewonnen werden kann. Jedoch sollten Reaktionsräder möglichst nah am Schwerpunkt der Masse liegen, um einen optimalen Wirkungsgrad zu haben.

Für die Feinjustierung beim Greifen des Asteroiden ist ein RCS unabdingbar, daher ist es notwendig ein RCS am Satelliten zu verbauen.

12.2. Fehlertoleranzkonzept

In diesem Kapitel werden grundlegende Herangehensweisen und Ideen zur Implementierung von Adaptionsfähigkeiten vorgestellt.

12.2.1. Systemebene

Auf Systemebene muss überlegt werden, welche Komponenten zusätzlich zu den funktionalen Komponenten benötigt werden, um adaptives Verhalten implementieren zu können. Eine weitere Aufgabe ist die Kommunikation zwischen den ZedBoards fehlertolerant zu gestalten. Auch innerhalb der ZedBoards muss die Kommunikation zwischen den Komponenten fehlertolerant sein. Dazu müssen neben dem Satellitensteuerungssystem, wie es ohne Fehler funktionieren soll, auch Fehler und deren Auswirkungen modelliert werden.

Konzeptioneller Aufbau eines adaptiven System

Bei Fehlern oder anderen Ereignissen, die eine Adaption notwendig machen, sendet die jeweilige Komponente entsprechende Events. Auf diese Events reagiert über die Monitore der Controller, der über den Adapter die gebrauchte Adaption durchführt, siehe Abbildung 12.1. Dies kann eine Anpassung des Scheduling sein, Aktivierung passiver Reserve oder Zurücksetzen von Prozessen oder ganzen Komponenten. Zu beachten ist, dass auch in den Komponenten Fehlertoleranzmechanismen implementiert sein können, sodass Fehler zuerst innerhalb der Komponente diagnostiziert werden und in dieser bereits korrigiert werden. Dies ist jedoch nicht immer innerhalb der Komponente möglich, weshalb es diese Komponenten zur Adaptivität benötigt. Diese müssen auch selbst fehlertolerant aufgebaut sein und Möglichkeiten zur Adaption besitzen. Wenn fehlertolerant auf Alterung reagiert werden soll, dann muss das Scheduling veränderbar sein. Denn durch Alterung kann ein Schaltkreis langsamer werden, sodass durch unfertige Berechnungen Fehler auftreten. Dadurch benötigen betroffene Komponenten mehr Rechenzeit [94].

Kommunikation

In [3] wird ein formaler Ansatz beschrieben, der ein SystemC Transaction Level Model (TLM) um Fehlertoleranz ergänzt. Die beschriebenen Schritte sind Modellextraktion, Modellierung der Fehler, Ergänzen von Fehlertoleranz und Refinement des TLMs. In [3] werden diese Schritte beispielhaft durchlaufen. Um die Kommunikation zwischen den ZedBoards fehlertolerant zu entwerfen ist in [90] ein fehlertolerantes Kommunikationssystems beschrieben, das viele verschiedene Fehler toleriert und daran angelehnt können Fehlertoleranzmechanismen übernommen werden. Grundlegendes zur fehlertoleranten Netzwerktopologie findet sich zum Beispiel in [31].

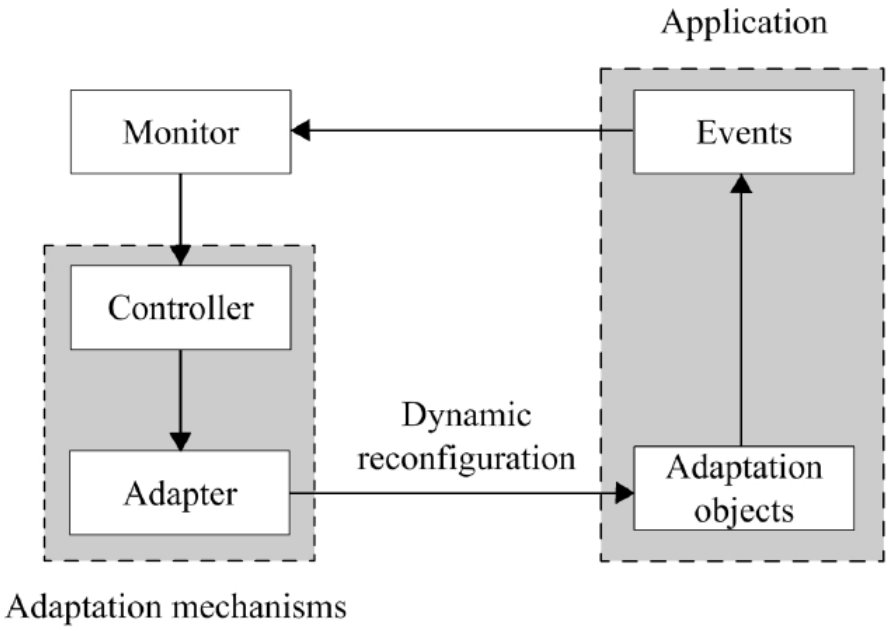


Abbildung 12.1.: Adaptives eingebettetes System [94]

12.2.2. Komponentenebene und Modulebene

Auf Komponenten und Modulebene gibt es sehr viele Ansätze Fehlertoleranz zu erreichen. Als Komponente werden hier alle Module, aus der diese besteht, zusammengefasst bezeichnet. Ein Modul der Orbitplanung wäre zum Beispiel die Berechnung einer bestimmten Größe. Ein Modul hat eine bestimmte Funktion innerhalb einer Komponente.

Monitoring von Komponenten: Um auf Komponenten Monitoring anzuwenden, können Watchdogs eingesetzt werden. Diese Überwachen die Komponente auf Aktivität. Watchdogs können auch auf Systemebene und Modulebene eingesetzt werden [31].

Acceptance tests: Innerhalb jede Komponente können Output, Input und Timing dieser kontrolliert werden, wie in [31] beschrieben. Das kann auch über Watchdogs stattfinden. Notwendig sind dazu auf jeden Fall Informationen über Output, Input und das Timing, also wann etwas zulässig ist.

Checkpointing: Bei Fehlern die nicht direkt durch Redundanz erkannt und korrigiert werden, werden Mechanismen zum Wiederherstellen eines funktionsfähigen Zustands benötigt. Dazu gehören Recovery Block Approach und Checkpointing [31].

Multiplexing (n-Modular): Eine Komponente kann für eine zu erfüllende Funktion mehrere gleichzeitig aktive Module haben. Aus diesen wird dann nach Mehrheitsentscheid der gültige Wert bestimmt. Zum Beispiel kann bei drei Modulen ein einzelner Fehler in einem Modul durch die beiden anderen Module toleriert werden [31]. Eine weitere Möglichkeit ist eine Fehlertoleranz durch Diversität, also zum Beispiel verschiedenen Implemen-

tierungen eines Algorithmus, sodass Programmierfehler nicht zum Ausfall aller Module führen [31].

Multiplicated: Anstatt mehrere Module gleichzeitig aktiv zu lassen, können die zusätzlichen Module zunächst passiv sein. Erst bei Erkennung eines Fehlers wird auf diese durch Rekonfigurierung zurückgegriffen [31].

Hybrider Ansatz: Es können beide zuvor genannten Möglichkeiten kombiniert werden [31].

Informationsredundanz: Durch Informationsredundanz können Fehler erkannt und bei genügend Redundanz auch korrigiert werden. Einfache Beispiele sind Parity Bits und Checksums. Es gibt noch mehr Möglichkeiten, zum Beispiel, in [31]. Codierungen können auch bei der Kommunikation auf Systemebene eingesetzt werden. Ein Beispiel zur Fehlererkennung über Monitoring wird in [22] gegeben. In [42] wird beispielhaft der Designprozess einer fehlertolerante Komponente dargestellt.

Fehlermodell

Hier wird dargestellt, wie Fehler modelliert werden können. Hiermit werden auch Fehler, die auf Systemebene auftreten, modelliert, da sich ein System in seine Komponenten aufteilt. Fehler in der Kommunikation können als Fehler der Komponenten modelliert werden.

Das generelle Vorgehen zur Modellierung der Fehler sieht zunächst die äußere Spezifikation der Komponenten vor, also welche Funktionen und Daten zwischen den Komponenten zur Verfügung gestellt beziehungsweise ausgetauscht werden. Diese Modellierung geschieht zunächst in der Annahme der Fehlerfreiheit. Anschließend werden die inneren Spezifikation der Komponenten auch unter Abwesenheit von Fehlern modelliert. Auch bei der inneren Spezifikation findet zunächst eine Unterteilung, hier in Module anstatt ganzer Komponenten, statt. Es müssen wie auf Komponentenebene die Abhängigkeiten zwischen den Modulen modelliert werden. Daraus und aus der Modellierung der Module ergibt sich eine *State Machine* der jeweiligen Komponenten. Anschließend können die Fehler durch Ergänzen von Fehlerzuständen und Übergängen zu diesen modelliert werden. Hierbei muss überlegt werden, welcher Fehler in welchen Zustand auftreten kann und welche Auswirkungen dieser hat, insbesondere welche Komponenten beziehungsweise Teile der Komponente betroffen sind. Darauf basieren können Fehlertoleranzmaßnahmen modelliert werden, die aus den ergänzten Fehlerzuständen Übergänge zu fehlerfreien Zuständen liefern. Dies können natürlich auch weitere fehlerfreie Zustände sein, die bei Abwesenheit von Fehlern nicht erreicht werden konnten. Dieses Vorgehen wird verfolgt, da es einfacher ist ein funktionierendes System um Fehlertoleranz zu erweitern, als von vornherein Fehlertoleranz mit zu implementieren, da es dadurch zunächst weniger komplex ist. Jedoch wird durchgehend darauf geachtet, dass Fehlertoleranz ergänzt werden kann.

Ein Fehler tritt in Form eines Fehlzustands, also eines unzulässigen Zustands einer Komponente, auf. Dieser kann entweder nur Auswirkungen innerhalb der Komponente haben oder auch zu Funktionsausfällen der Komponente führen. Dadurch können sich Fehler auch auf

andere Komponenten übertragen, obwohl die Fehlerursache innerhalb einer weiteren Komponente liegt [19]. Es gibt folgende Fehlerursachen [19]:

Entwurfsfehler: Hierzu gehören Implementierungsfehler und Spezifikationsfehler. Bei der Übertragung der Anforderungen in eine Spezifikation können Unvollständigkeiten, Widersprüche oder Abweichungen auftreten. Dies sind Spezifikationsfehler. Ein Implementierungsfehler ist zum Beipiel ein fehlerhaft implementierter Algorithmus. Zu den Entwurfsfehlern würden auch Dokumentationsfehler zählen, die zu falschem Gebrauch oder Wartung führen.

Betriebsfehler: Hierzu zählen störungsbedingte Fehler, wie zum Beispiel Fehler durch Strahlung im Weltall. Verschleißbedingte Fehler, wie zum Beispiel Alterung, und zufällige physikalische Fehler gehören auch zu Betriebsfehlern. Wartungs- und Bedienungsfehler würden auch hierzu zählen.

Darüber hinaus gibt es noch Herstellungsfehler bei der Herstellung der Hardware, diese werden hier jedoch nicht berücksichtigt.

Beim Fehlermodell wird die Dauer des Fehlers unterschieden. Dabei gibt es permanente Fehler, die dauerhaft auftreten, bis entsprechende Maßnahmen getroffen wurden. Die andere Art von Fehlern sind intermittierende Fehler, die nur vorübergehend auftreten. Hier kann die Fehlerursache permanent oder transient, also vorübergehend, sein [19].

Darauf aufbauend soll eine Fehlervorgabe erstellt werden, die angibt welche Fehler toleriert werden sollen. Diese muss auch enthalten, welche Fehler gleichzeitig toleriert werden sollen. Alle Fehler bei gleichzeitigem Auftreten zu tolerieren ist allgemein nicht möglich [19].

Formale Verifikation

Es besteht mit dem Tool STATE [77] die Möglichkeit aus SystemC einen *timed automaton* zu generieren, der mit dem Modelchecker UPPAAL[27] auf Eigenschaften überprüft werden kann [46]. Diese formale Verifikation ist im Hinblick auf Fehlertoleranz optional, da primär die Fehlerinjektion zum Einsatz kommt. Jedoch sollte das Modell ohne Fehlertoleranz auf gewisse Eigenschaften überprüft werden können.

12.2.3. Phasenplanung

Da der Satellit auf seiner Mission mehrere Phasen durchläuft und diese unterschiedliche Anforderungen an die Funktion des Satelliten haben, gibt es eine Phasenplanung die festlegt, in welcher Phase sich der Satellit befindet und in welche Phasen von dort aus gewechselt werden kann. Die Phase des Satelliten, in der dieser sich befindet, wird als Modus des Satelliten bezeichnet. Es wird auch mindestens einen Fehlerbehandlungsmodus geben, in den bei Erkennung von Fehlern gewechselt wird, um Fehlerbehandlungsmaßnahmen durchzuführen. Beim Zurückwechseln in den ursprünglichen Modus sollen dann durch Rekonfigurierung oder anderen Möglichkeiten zur Fehlerbehebung die Fehler behoben werden. Generell findet das Aktivieren von passiven Reserven, also zum Beispiel die Hinzunahme eines weiteren Teils des FPGAs, der vorher inaktiv war, sowie das Terminieren und Starten von Tasks und die Anpassung von Parametern von Tasks, wie zum Beispiel die Periode, nach der der Task wieder aktiv wird, nur

während eines Moduswechsels statt. Ein Beispiel für unterschiedliche Modi sind Modus beim Start, Modus im Fliegen eines festen Orbits und Modus zur Annäherung an den Asteroiden. In Abbildung 12.2 sind die verschiedenen Modi zu sehen. Es ist auch ein Zeitraffer-Modus abgebildet, der jedoch optional ist, da noch nicht entschieden ist, wie der Satellit gezielt in vorgegebene Zustände und damit Spielstände versetzt wird. [94]

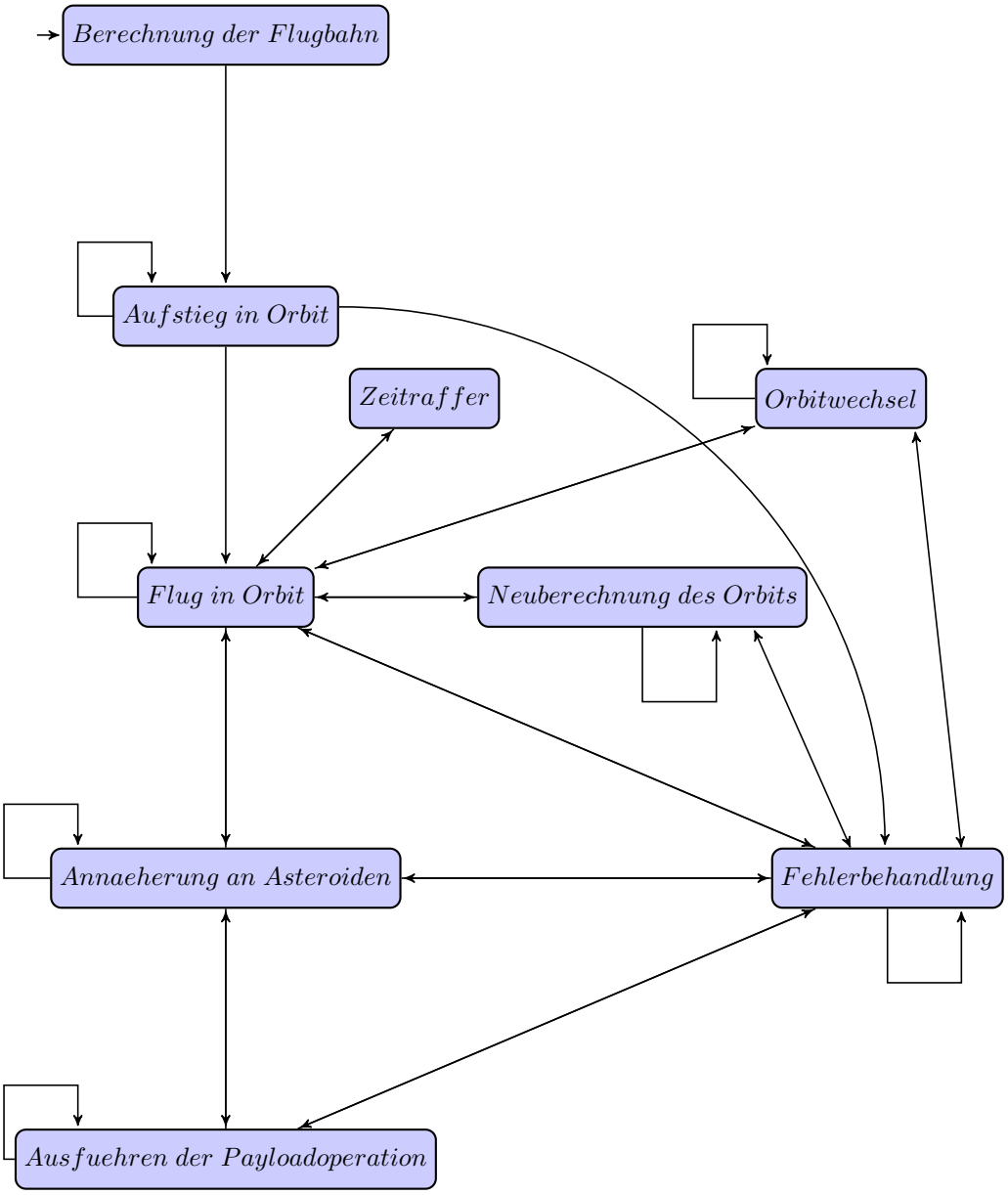


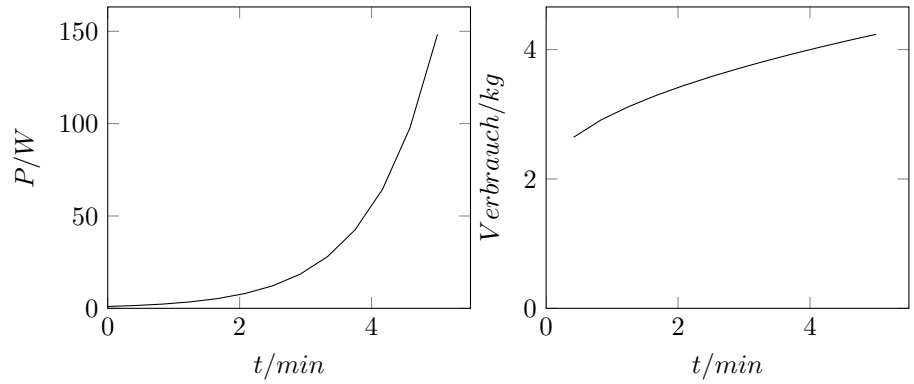
Abbildung 12.2.: Modi des Satelliten

12.3. Ressourcenmanagementkonzept

In diesem Kapitel wird das Ressourcenmanagement definiert. Es wird zuerst eine allgemeine Definition von Verbraucher und Erzeuger gegeben. Danach werden die Ressourcen im Detail besprochen.

12.3.1. Verbraucher

Ein Verbraucher setzt eine oder mehrere Ressourcen auf dem Satelliten um. Er lässt sich einschalten, ausschalten, drosseln und in Bereitschaft versetzen. Für jede Funktion besitzt der Verbraucher eine Funktion $W(t)$, die den Ressourcenverbrauch (oder die Belegung) über einen Zeitraum darstellt. Zwei beispielhafte Kennlinien zur Verbrauchsfunktion sind in Abbildung 12.3 dargestellt. Des Weiteren ist in Tabelle 12.2 eine Liste aller angenommenen Verbraucher dargestellt. Auch die Satellitensteuerung, bzw. die Hardware (ZedBoard) auf der diese ausgeführt wird, zählt mit zu diesen Ressourcen.



(a) Umsetzung von elektrischer Energie in Watt pro Minute in einem Einschaltvorgang.

(b) Verbrauch von Treibstoff in Kilogramm pro Minute in einem Einschaltvorgang.

Abbildung 12.3.: Beispielhafte Kennlinien von Verbrauchern.

Tabelle 12.1.: Auflistung möglicher Verbraucher auf dem Satelliten.

Verbraucher	Ressource(n)	Art
ZedBoard	Elektrische Leistung	Umsetzung
Antrieb	Elektrische Leistung, Treibstoff	Umsetzung/Verbrauch
Sensor	Elektrische Leistung	Umsetzung
Elektrischer Aktuator	Elektrische Leistung	Umsetzung

12.3.2. Erzeuger

Ein Erzeuger produziert eine physikalisch messbare Ressource. Erzeuger lassen sich – wie Verbraucher – einschalten, ausschalten und in Bereitschaft versetzen. Jede dieser Funktionen besitzt eine Produktionsfunktion $Prod(t) = -W(t)$, die in diesem vereinfachten Modell nur einen negativen Verbrauch besitzt und mit den Werten der Verbraucher verrechnet werden kann.

12.3.3. Speicher

Eine Sonderform von Erzeugern und Verbrauchern sind Speicher (Tanks, Batterien, ...). Diese können für einen Zeitraum Δt als ein Erzeuger oder ein Verbraucher agieren. Darüber hinaus besitzen sie einen Füllstand V_0 und eine maximale Kapazität V_{max}.

12.3.4. Ressource

Als Ressource werden alle Mittel auf dem Satelliten verstanden, die verbraucht, umgesetzt oder belegt werden müssen, damit der Satellit seine Mission erfüllen kann. Des Weiteren besitzt eine Ressource eine Einheit. Sie lässt sich damit quantifizieren. In diesem Dokument wird zwischen physikalischen und Computerressourcen unterschieden.

Die Ressourcen sind in Tabelle 12.2 dargestellt. In den folgenden Unterabschnitten werden sie nochmals detaillierter beschrieben. Es wird angenommen, dass eine höhere Auslastung einer Computerressource (Arbeitsspeicher, Prozessorauslastung, FPGA) zu einem höheren Verbrauch elektrischer Leistung führt.

Tabelle 12.2.: Auflistung möglicher Ressourcen auf dem Satelliten und deren Speicherform.

Ressource	Einheit	Speicherform
Elektrische Leistung	Watt	Batterie
Treibstoff	Liter oder Gramm	Tank
Arbeitsspeicher	Byte	-
Prozessorauslastung	CPU-Zeit pro Zeitintervall	-
FPGA-Belegung	Prozentual	-

Physikalische Ressourcen

Der Satellit (so wie auch die Trägerrakete) sind auf physikalische Ressourcen angewiesen. Diese Ressourcen werden zum Be- und Antreiben des Satelliten benötigt. KSP stellt die in Tabelle 12.3 genannten Ressourcen zur Verfügung. Diese besitzen Einschränkungen bezüglich Erneuerbarkeit und Lagerfähigkeit, daher ist eine Ressourcenplanung sehr wichtig. Nichterneuerbare Ressourcen müssen sich bereits zum Start an Bord des Raumfahrzeugs befinden, dazu sind primär die Ressourcen zum Antrieb (Liquid Fuel, Oxidizer, alternativ auch Solid Fuel oder Xenon Gas) sowie zur Lageänderung (Mono Propellant) zu zählen. Erneuerbare Ressourcen, wie Electric Charge, müssen für den kompletten Missionsverlauf in ausreichender Menge erzeugt werden.

Der Verbrauch von nichterneuerbaren Resourcen muss sorgfältig geplant werden. Diese stellen eine harte Lebenszeitbegrenzung für das Projekt dar, denn ohne Antrieb kann der Satellit nicht mehr agieren. Bei den erneuerbaren Ressourcen ist zu beachten, dass die Bedürfnisse der Verbraucher durch entsprechende Erzeuger gedeckt werden können. Üblicherweise werden dazu auf Satelliten Solarpaneele verwendet. Zudem müssen erneuerbare Energien in ausreichender Menge gelagert werden, um diese bei Nichtverfügbarkeit des Erzeugers weiterhin zur Verfügung stellen zu können.

Die erneuerbare Ressource *Electric Charge* ist besonders zu betrachten, da diese für den Betrieb des Satelliten notwendig ist. Sollte die Menge an vorhandener Electric Charge nicht ausreichend sein, um den minimalen Verbrauch der Erzeuger zu decken, ist der Satelliten nicht mehr funktionsfähig. Sollte der Fall eintreten, dass in absehbarer Zeit die Ressource nicht erneuert werden kann, muss der Satellit in einen Schlafzustand versetzt werden, bis der Erzeuger wieder ausreichende Mengen zur Verfügung stellen kann. Ist die Electric Charge aufgebraucht, ist der Satellit verloren.

Tabelle 12.3.: Liste der Ressourcen in Kerbal und ein Auszug ihrer Eigenschaften. Quelle: [61]

Ressource	Erneuerbar	Lagerfähig
Electric Charge	Ja	Ja
Mono Propellant	Nein	Ja
Liquid Fuel	Nein	Ja
Oxidizer	Nein	Ja
Solid Fuel	Nein	Ja
Xenon Gas	Nein	Ja

Computerressourcen

Ressourcen wie Arbeitsspeicher, Prozessorauslastung und FPGA werden nicht verbraucht, sondern belegt oder ausgelastet. Allerdings wird angenommen, dass eine höhere Auslastung einer Computerressourcen zu einer erhöhten Leistungsaufnahme von elektrischer Energie führt. Sie können aber nicht vollständig abgeschaltet werden, da die Steuerungssoftware diese Ressourcen benötigt, um die Mission des Satelliten zu erfüllen.

Des Weiteren darf die Satellitensteuerung keine der Computerressourcen vollständig belegen. Einerseits, weil es unbekannt ist, ob Wechselwirkungen mit anderen Tasks oder Alterungseffekten zu einer später erhöhten Auslastung der Ressourcen führen können. Andererseits befindet sich auf dem System, auf dem die Satellitensteuerung läuft, auch die Fehlerinjektion, welche auch Computerressourcen belegen kann. Um die Effekte und die Fehlerinjektion nicht explizit berücksichtigen zu müssen, wird in erster Näherung angenommen, dass die Satellitensteuerung nur 50% aller Computerressourcen belegen soll. Diese Grenze kann beim Fortschreiten des Projektes aber weiter angepasst werden.

Prozessor

Bei der Prozessorauslastung wird eine Zeitspanne Δt angenommen, in denen die Tasks T_1 bis T_n zeitlich eingeplant werden werden. Des Weiteren gäbe es einen Leerlaufprozess T_{leer}, der nur die verbleibende Zeit im Zeitintervall Δt darstellt. Ein beispielhaftes Scheduling ist in Abbildung 12.4 dargestellt. Der Prozess T_{leer} ist zu vier von acht Zeiteinheiten aktiv, das heißt der Prozessor ist zu 50% ausgelastet.

Es ist zwischen Tasks der Satellitensteuerung, des Betriebssystems und der Hardware zu unterscheiden. Ideal sollte das Betriebssystem laufen, wenn die Satellitensteuerung keine Prozessorzeit benötigt.

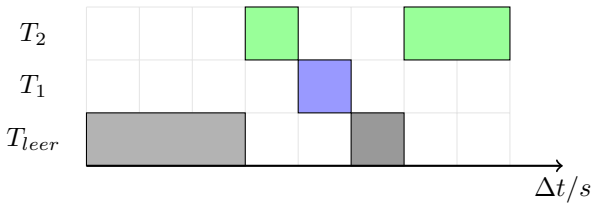


Abbildung 12.4.: Beispiel Prozessorauslastung.

Damit der Prozessor als weitere Ressource verwaltet werden kann, muss es möglich sein, dass jeder Task unterbrochen werden kann. Wie dies erfolgt (d.h. welcher Scheduling-Algorithmus benutzt wird), ist technische Implementierung und damit nicht Teil dieser Abstraktion.

Arbeitsspeicher

Der Arbeitsspeicher kann in statisch und dynamisch wachsenden Speicherverbrauch eingeteilt werden:

statisch: Hierzu zählen Programmcode, Konstanten, globale Variablen und First In First Out (FIFO)-Größen. Dieser Speicherverbrauch ist zur Übersetzungszeit bekannt und wird sich nicht ändern.

dynamisch: Hierzu zählen Heap und Task-Stacks. Ersteres wird vielleicht von einer Memory Management Unit (MMU) oder Memory Protection Unit (MPU) überwacht. Möglicherweise kann hier das Betriebssystem eingreifen. Letzteres wächst und schrumpft zur Laufzeit eines Tasks.

Der statische Verbrauch ergibt sich nach Übersetzung des Programms und wird konstant über die ganze Laufzeit sein. Der dynamische Verbrauch kann weiterhin in überwachbar und nicht überwachbar eingeteilt werden.

Die nötige Stackgröße der Software-tasks kann durch Function-Call-Graphs ermittelt werden, da jeder Funktionsaufruf eine feste Größe auf dem Stack belegen wird. Stacks können aber durch Rekursion beliebig groß wachsen. Falls Rekursion verwendet wird, muss der Speicherbedarf eines Tasks durch Heuristiken und Wertebereiche für Eingabedaten vorhersehbar gemacht werden.

FPGA-Belegung

Die Satellitensteuerung wird keine dynamische (partielle) Rekonfiguration besitzen. Es wird angenommen, dass die Kosten (Zeit, Leistung, ...) für die Rekonfiguration und den anfallenden Speicherverbrauch durch die Speicherung der entsprechenden Bitstreams zu aufwendig ist. Die FPGA-Belegung ist damit konstant. Die FPGA-Logik kann aber auf verschiedenen Abstraktionseben im Entwicklungsprozess in Hinsicht auf Stromverbrauch verbessert werden (vgl. [8][69]).

Ressourcenverteilung

Jede Ressource besitzt ein eigenes Netzwerk aus Leitungen (Kabel, Rohre, ...). Dieses Netzwerk ist der Satellitensteuerung bekannt. Ein Erforschungsalgorithmus, der erkennt, welche Erzeuger mit welchen Verbrauchern verbunden sind, wird nicht vorgesehen.

Leitung

Bei physischen Ressourcen wird davon ausgegangen, dass der Zustand (gesperrt/offen) einer Leitung abgerufen werden kann. Eine Spezialisierung davon stellen Tore dar, bei denen man den Leitungszustand setzen kann. Diese stellen damit die Grundlage zum *routen* der Ressource durch das Netzwerk dar. In Abbildung 12.5 ist ein Netzwerk zwischen zwei Tanks und drei Triebwerken dargestellt. Verliert der Satellit ein Triebwerk oder lässt sich ein Triebwerk nicht mehr benutzen, können die Tore dafür sorgen, dass kein Treibstoff mehr geliefert wird. Prinzipiell können Tore beliebig viele Segmente sperren und öffnen.

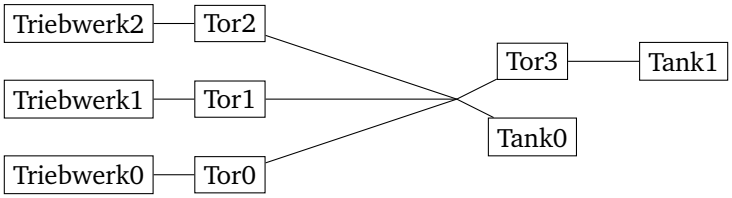


Abbildung 12.5.: Ressourcennetzwerk von Treibstoff

Leitungszustände

Es wird davon ausgegangen, dass diese Tore im Notfall selbstständig sperren. Das bedeutet, dass das Satellitensteuerungssystem bei Verlust eines Triebwerkes nicht explizit den Befehl zum Sperren gibt. Der Zustand jedes Tores kann gesetzt und abgefragt werden. Die Zustände sind in Abbildung 12.6 dargestellt.

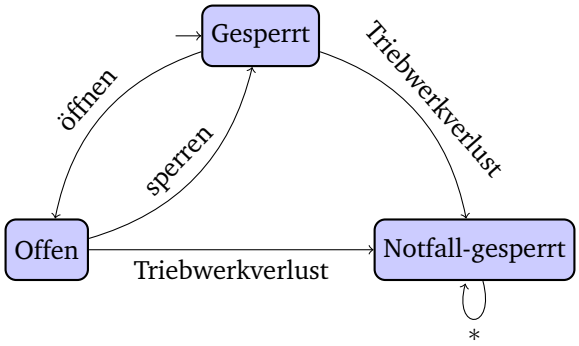


Abbildung 12.6.: Zustandsautomat eines Tores.

Ressourcennetzwerk als Adjazenzmatrix

Jede Ressource besitzt damit ein oder mehrere Netzwerke. Die Satellitensteuerung muss wissen, wie sie die Ressource von den Erzeugern an die Verbraucher verteilen kann und ist damit primär vom Zustand der Tore und Leitungen abhängig. Das Netzwerk aus Abbildung 12.5 ist in Abbildung 12.7 als Adjazenzmatrix dargestellt. Es wird der Zustand der Tore übernommen. In der Adjazenzmatrix ist noch ein viertes Triebwerk und ein dritter Tank gelistet. Diese Komponenten benutzen dieselbe Ressource, sind aber nicht Teil des Netzwerkes in Abbildung 12.5. Nicht-verbundene Komponenten wurden mit - markiert.

	Tank0	Tank1	Tank2
Triebwerk0	Tor0	Tor3 & Tor0	-
Triebwerk1	Tor1	Tor3 & Tor1	-
Triebwerk2	Tor2	Tor3 & Tor2	-
Triebwerk3	-	-	Offen

Abbildung 12.7.: Adjazenzmatrix

12.4. Orbitplanungskonzept

In diesem Abschnitt wird ein Verfahren zur Bestimmung aller nötigen Geschwindigkeits- und Orientierungsänderungen vorgestellt, die zum Erfüllen der Mission notwendig sind.

12.4.1. Darstellung des Orbits

Zu Anfang der Berechnung befindet sich der Satellit in einem stabilen kreisförmigen, niedrigen Orbit um den Planeten Kerbin. Für diesen Ausgangsorbit sind alle Parameter, die Kepler-Elemente zur Beschreibung eines Orbits, bekannt. Dies gilt insbesondere für den Orbitradius r_{LEO}, die Inklination i_{LEO} und die aktuelle Anomalie Θ_{LEO}.

Der abzufangende Asteroid befindet sich in einem stabilen Orbit um Kerbol. Des Weiteren ist sichergestellt, dass zu keinem Zeitpunkt der nachfolgend beschriebenen Ausführung die Flugbahn des Asteroiden eine SOI eines Planeten, mit Ausnahme der von Kerbin, kreuzt. Über die Flugbahn des Asteroiden sind alle orbitalen Parameter bekannt, insbesondere der Abstand der Apoapsis $r_{Ap,Ast}$, der Abstand der Periapsis $r_{Pe,Ast}$, das Argument der Periapsis ω_{Ast}, die Länge des aufsteigenden Knotens Ω_{Ast}, die Inklination i_{Ast} und die aktuelle mittlere Anomalie Θ_{Ast} der Epoche.

Außerdem müssen die Kepler-Elemente des Orbits von Kerbin bekannt sein.

12.4.2. Patched Conic Annäherung

Zur Vereinfachung der Berechnung eines n-Körper Problems, kann es in eine Aneinanderreihung von 2-Körper Problemen zerlegt werden. Somit ergibt sich eine Flugbahn die mehrere zusammengefügte Kegelschnitte beinhaltet.

Ein interplanetarer Flug besteht somit aus einer hyperbolischen Fluchtbahn von der Erde, die von einer elliptischen Bahn um die Sonne fortgeführt wird und in einer hyperbolischen Einfangbahn um den Zielplaneten endet. Für unsere Mission einen Asteroiden einzufangen, entfällt die letzte Phase.

Hierfür werden folgende Manöver benötigt:

- Apoapsis- und Periapsisänderung
- Inklinationsänderung
- Änderung der Länge des aufsteigenden Knotens

Insbesondere müssen zusätzliche Berechnungen für das Austrittsmanöver von Kerbin zu Kerbol und das letzte Manöver, der Phasenanpassung, gemacht werden.

12.4.3. Zeitpunktberechnung

Bei bekannter wahrer Anomalie zur Ausführung eines Manövers ist die Berechnung des Zeitpunkts, wann der Satellit diese wahre Anomalie hat, immer gleich. Diese wird im Folgenden dargestellt.

Zunächst muss die Umlaufzeit des Satellitenorbits wie folgt berechnet werden. Dabei ist a die große Halbachse und μ der Gravitationsparameter des umkreisten Objekts.

$$U = \sqrt{\frac{4\pi^2 a^3}{\mu}} \qquad \text{aus [13]} \tag{12.21}$$

Aus der bekannten wahren Anomalie wird dann die exzentrische Anomalie berechnet. Hier ist E die exzentrische Anomalie, Θ_{LEO} die wahre Anomalie des Satelliten und e die Exzentrizität des Satellitenorbits.

$$tan(E) = \frac{\sqrt{1-e^2} * sin(\Theta_{LEO})}{1 + e * cos(\Theta_{LEO})} \qquad \text{aus [13, S. 20]} \tag{12.22}$$

Anschließend wird über die Kepler-Gleichung die mittlere Anomalie bestimmt. Hier ist M die mittlere Anomalie, e wieder die Exzentrizität und E die exzentrische Anomalie.

$$M = E - e * sin(E) \qquad \text{aus [13, S. 21]} \tag{12.23}$$

Aus der mittleren Anomalie M kann nun mit der Umlaufzeit U, der Anomalie bei Epoche M_0 sowie der zugehörigen Epoche t_0 der Zeitpunkt t bestimmt werden, wann der Satellit die berechnete mittlere Anomalie hat.

$$M = M_0 + \frac{2 * \pi}{U} * (t - t_0) \qquad \text{aus [13, S. 20]} \tag{12.24}$$

12.4.4. Berechnung Entfernung zum umkreisten Objekt

Für die Vis-Viva Gleichung ist die Kenntnis der aktuellen Entfernung r zum umkreisten Objekt notwendig. Diese wird mit der großen Halbachse a, der Exzentrizität e und der wahren Anomalie Θ_{LEO} berechnet.

$$r = a * \frac{1 - e^2}{1 + e * cos(\Theta_{LEO})} \qquad \text{aus [23]} \tag{12.25}$$

12.4.5. Apoapsis- und Periapsisänderung

Für eine Apoapsis- bzw. Periapsisänderung müssen der Zeitpunkt und die benötigte Geschwindigkeitsänderung berechnet werden. Für die Ausrichtung gibt es nur zwei Möglichkeiten. Falls die Apoapsis bzw. Periapsis größer werden soll, so muss die Ausrichtung prograde sein, andernfalls retrograde. Diese sind die Ausrichtungen im orbitales Referenzsystem.

Über die Vis-Viva Gleichung wird mit Hilfe der großen Halbachse und der Entfernung zum umkreisten Objekt die aktuelle Geschwindigkeit berechnet. Durch Berechnung der benötigten Geschwindigkeit mit Hilfe der zu erreichenden großen Halbachse kann durch die Berechnung der Differenz der beiden Geschwindigkeiten der benötigte Δv-Wert festgestellt werden.

$$v = \sqrt{\mu * \left(\frac{2}{r} - \frac{1}{a}\right)} \qquad \text{aus [12]} \tag{12.26}$$

12.4.6. Inklinationsänderung

Für die Berechnung einer Inklinationsänderung werden Zeitpunkt und die benötigte Geschwindigkeitsänderung, sowie Ausrichtung benötigt. Eine Inklinationsänderung findet immer am

aufsteigenden Knoten oder absteigenden Knoten statt. Bei der Ausrichtung gibt es zwei Möglichkeiten. Eine Möglichkeit ist, den Satelliten genau normal oder genau antinormal im orbitalen Referenzsystem auszurichten. Hierbei werden jedoch neben der Inklination auch andere Kepler Elemente verändert, da der Betrag der Geschwindigkeit am Manöverpunkt verändert wird und damit Apoapsis bzw. Periapsis oder auch die Exzentrizität verändert werden. Dies ist für eine genaue Annäherung an einen Asteroiden unerwünscht. Daher haben wir uns folgende Berechnung für eine Inklinationsänderung überlegt. Die Ausrichtung wird nicht genau normal oder antinormal sein, sondern so, dass alle anderen Kepler Elemente unverändert bleiben.

Über die Vis-Viva Gleichung wird die Geschwindigkeit zum Zeitpunkt des Manövers berechnet. Da die Geschwindigkeit nach dem Manöver gleich der Geschwindigkeit vor dem Manöver sein soll, kann Betrag und Ausrichtung der Geschwindigkeitsänderung mit Hilfe des Kosinussatzes nach Abbildung 12.8 berechnet werden. Hier ist Δi_{LEO} die Inklinationsänderung.

$$\Delta v^2 = 2 * v_{LEO}^2 * (1 + cos(\Delta i_{LEO})) \tag{12.27}$$

Die Ausrichtung lässt sich auch mit Hilfe des Kosinussatzes berechnen. In Abbildung 12.8 ist ϕ der Winkel rechtsunten im Dreieck.

$$cos(\phi) = \frac{\Delta v}{2 * v_{LEO}} \tag{12.28}$$

Der Steuerkurs in Grad ergibt sich daraus wie folgt.

$$heading = 180 - \frac{180}{\pi} * \phi \tag{12.29}$$

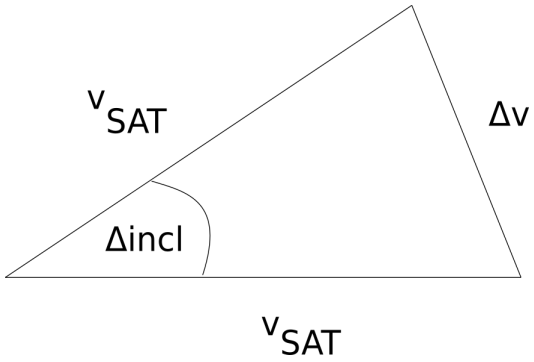


Abbildung 12.8.: Berechnung Inklinationsmaneuver

12.4.7. Änderung der Länge des aufsteigenden Knotens

Zum Ändern des aufsteigenden Knotens haben wir folgendes erarbeitet. Die Berechnungen für dieses Manöver sind denen der Inklinationsänderung gleich, bis auf eine vorausgehende Rechnung. Die Länge des aufsteigenden Knotens ist der zweite Winkel, wie die Ebene des Orbits zur Referenzebene steht. Dieser Winkel wird nicht wie die Inklination am aufsteigenden Knoten oder absteigenden Knoten geändert, sondern $\frac{\pi}{2}$ wahrer Anomalie später nach einem der beiden Knoten. Hier ist eine zusätzliche Berechnung notwendig.

Für eine Änderung der Länge des aufsteigenden Knotens um $\Delta\psi$ ist eine Inklinationsänderung an beschriebenem Punkt um $sin(i_{LEO}) * \Delta\psi$ nötig. Dies ist eine Näherung, die die Krümmung des Satellitenorbits nicht berücksichtigt, für kleine Änderungen aber sehr genau ist.

12.4.8. Austrittsmanöver

Das Zusammenstellen des Austrittsmanöver haben wir uns selber überlegt, nur einzelne Formeln zur Berechnung bestimmter Größen sind aus anderen Quellen übernommen. Zur Berechnung des Austrittsmanöver müssen folgende Größen berechnet werden:

Apoapsis des Orbits um Kerbol nach Austrittsmanöver: Die große Halbachse des Satellitenorbits um Kerbol soll so gewählt sein, dass nach Angleichung der Apoapsis Asteroid und Satellit gleichzeitig bei der Apoapsis sind.

Inklination des Satellitenorbits um die Erde vor dem Manöver: Aus der großen Halbachse ergibt sich benötigte Inklination, um diese zu erreichen.

Punkt, an dem die SOI von Kerbin verlassen wird

Echte Anomalie um Kerbin tangential zu Kerbol zu verlassen

Zur Bestimmung der Apoapsis ap_{LEO} nach dem Austrittsmanöver wird das Newton-Verfahren angewendet. Die Funktion deren Nullstelle bestimmt wird nun vorgestellt: Hier ist t_{AST} die Zeit, die der Asteroid bis zum Erreichen seiner Apoapsis braucht, pe_{KERB} die Periapsis Kerbins und ap_{AST} die Apoapsis des Asteroiden.

$$T(ap_{LEO}) = (2\pi\sqrt{\frac{1}{\mu}}(\frac{1}{2}\sqrt{\frac{(ap_{LEO}+pe_{KERB})^3}{8}} + \frac{\Delta anomaly}{2*\pi}\sqrt{(ap_{LEO})^3} + \frac{1}{2}\sqrt{\frac{(ap_{LEO}+ap_{AST})^3}{8}}) - t_{AST} \tag{12.30}$$

Die Funktion beschreibt die Zeit des Satelliten bis zur Apoapsis des Asteroiden abzüglich der Zeit des Asteroiden zu seiner Apoapsis. Die Flugbahn des Satelliten, wie in Abbildung 12.9 zu sehen, setzt sich zusammen aus einer halben Umlaufzeit mit pe_{KERB} und ap_{LEO}. Dann fliegt der Satellit in einem kreisförmigen Orbit mit Radius ap_{LEO}. Dieser muss $\Delta anomaly$ lang geflogen werden, um das Argument der Periapsis des Satellitenorbits mit dem Argument der Periapsis des Asteroidenorbits gleichzusetzen. Zuletzt fliegt der Satellit eine halbe Umlaufzeit mit Periapsis ap_{LEO} und Apoapsis ap_{AST}.

Aus pe_{KERB} und ap_{LEO} ergibt sich nun die große Halbachse, die der Satellit nach dem Austrittsmanöver haben soll. Daraus wird über die Vis-Viva Gleichung die Geschwindigkeit v_{LEO} berechnet, die der Satellit nach dem Verlassen der SOI Kerbins haben soll. Über die Vis-Viva Gleichung wird nun auch die Geschwindigkeit Kerbins v_{KERB} bestimmt. Aus den beiden Geschwindigkeiten und der Inklination i_{KERB} kann mit Hilfe des Kosinussatzes die Fluchtgeschwindigkeit v_∞ bestimmt werden.

$$v_\infty^2 = v_{KERB}^2 + v_{LEO}^2 - 2 * v_{KERB} * v_{LEO} * cos(i_{AST}) \tag{12.31}$$

Mit Hilfe des Kosinussatzes wird nun auch die Inklination des Satellitenorbits um Kerbin berechnet.

Da die Fluchtgeschwindigkeit v_∞ jetzt bekannt ist, kann die große Halbachse des hyperbelförmigen Fluchtorbits berechnet werden.

$$a_{hyperbel} = \frac{\mu}{v_\infty^2} \qquad \text{aus [85]} \tag{12.32}$$

Mit der Vis-Viva Gleichung wird daraus das benötigte Δv des Austrittsmanövers berechnet. Aus den nun folgenden Rechnungen ergibt sich die Anomalie, wann das Austrittsmanöver stattfinden muss. Daraus wird die Exzentrizität der Hyperbel mit der Periapsis pe_{LEO} des Satellitenorbits um Kerbin berechnet.

$$e_{hyperbel} = -\frac{pe_{LEO}}{a_{hyperbel}} \qquad \text{aus [85]} \tag{12.33}$$

Jetzt kann die Anomalie ϕ, die der Satellit beim Austreten aus der SOI Kerbins hat, berechnet werden.

$$cos(\phi) = \frac{a_{hyperbel} * (1 - e_{hyperbel}^2) - SOI_{KERB}}{SOI_{KERB} * e_{hyperbel}} \qquad \text{aus [85]} \tag{12.34}$$

Der Flugbahnwinkel *FPA* wird wie folgt berechnet.

$$tan(FPA) = \frac{e_{hyperbel} * sin(\phi)}{e_{hyperbel} * cos(\phi) + 1} \qquad \text{aus [85]} \tag{12.35}$$

Die Anomalie des Satelliten Θ_{LEO} für das Austrittsmanöver lässt sich nach unserer Ausarbeitung wie folgt berechnen.

$$\Theta_{LEO} = 2\pi - \omega_{LEO} - \phi + FPA \tag{12.36}$$

Hierbei ist $2\pi - \phi$ die wahre Anomalie, an der der Satellit beim aufsteigenden Knoten ist. Damit der Satellit Kerbin horizontal zu Kerbol verlässt, muss das Manöver zunächst um die wahre Anomalie, bis der Satellit das Ende der SOI erreicht, früher ausgeführt werden. In der Formel ist dies $wahreAnomalie$. Da die Flugbahn am Ende der SOI jedoch nicht horizontal zu Kerbin ist, sondern weiter nach außen von Kerbin weg gerichtet ist, muss dieser Winkel wieder dazu gerechnet werden. Dieser wird Winkel ist genau der *FPA*.

12.4.9. Phasenanpassung

Die folgenden Überlegungen zur Phasenanpassung haben wir selber geführt. Zur Berechnung einer Phasenanpassung wird zunächst die zeitliche Differenz zwischen dem Satelliten und dem Asteroiden an deren Apoapsis berechnet. Voraussetzung ist, dass die Apoapsis des Satelliten mit der Apoapsis des Asteroiden übereinstimmt. Dies geschieht mit Hilfe der in Unterkapitel 12.4.3 Rechnung. Anschließend wird die benötigte große Halbachse wie folgt berechnet. Die Formel zur Berechnung der Umlaufzeit umgestellt ergibt:

$$a^3 = \frac{U^2 * \mu}{4 * \pi^2} \tag{12.37}$$

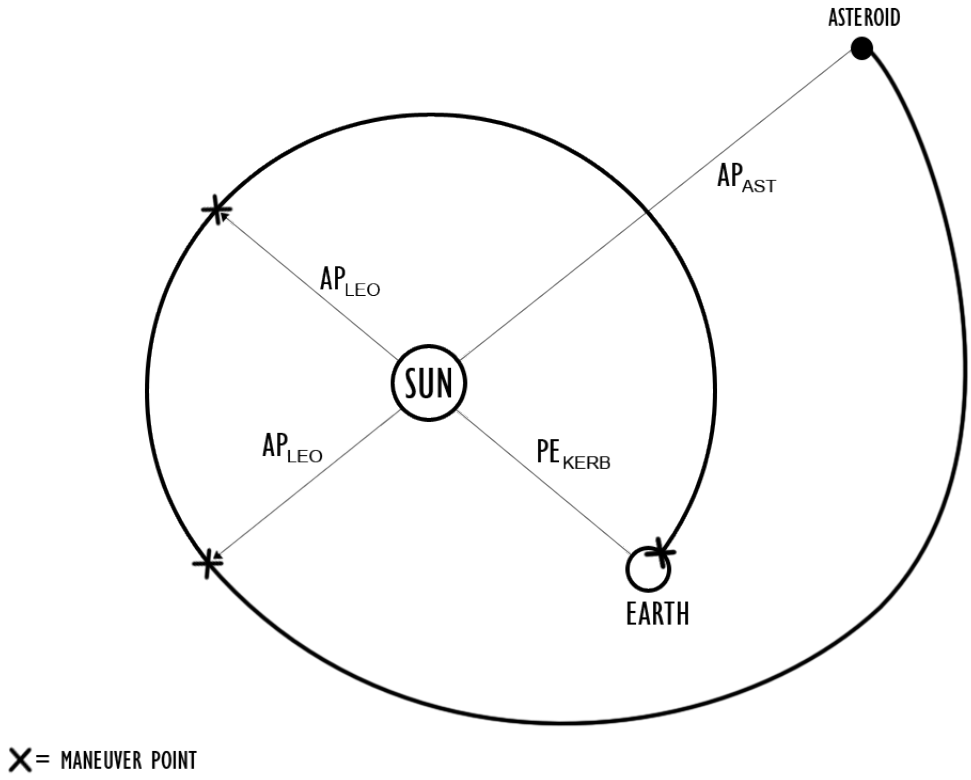


Abbildung 12.9.: Flugbahn Austrittsmaneuver und folgende Maneuver

12.4.10. Finale Annäherung

Am Anfang dieser Phase sollten sich sowohl Satellit als auch Asteroid auf der Apoapsis befinden. Der Abstand zwischen den beiden Objekten sollte nur noch wenige Kilometer betragen. Der restliche Abstand muss nun über gezieltes, kurzzeitiges Einsetzen der Antriebe und des RCS-Systems in Richtung des Asteroiden überwunden werden.

12.5. Kommunikationskonzept

In diesem Kapitel werden die Kommunikationsschnittstellen beschrieben. Es wurden zwei Arten identifiziert:

- die Kommunikation zu den Satellitenkomponenten (im folgenden als *Feldbus* bezeichnet.
- Kommunikation zwischen den Satellitensteuerungen der ZedBoards untereinander.

12.5.1. Feldbus

Der logische Aufbau des Feldbusses ist in Abbildung 12.10 dargestellt. Die Satellitensteuerungssysteme bekommen die Bezeichner S_0 bis S_n. Diese Systeme sind physisch voneinander getrennt. Jedes System stellt hierbei ein ZedBoard dar. Die Satellitenkomponenten, die über diesen Feldbus gesteuert werden, sind zusammengefasst unter dem Block *Peripherie*. Auch wenn nach diesem Aufbau alle Systeme mit der Peripherie verbunden sind, soll es nur einen

Feldbus-Master geben, der auch Befehle an die Peripherie senden darf. Nachrichten über diesen Feldbus werden prinzipiell gebroadcastet. Das heißt, dass auch die anderen Systeme die Befehle des Masters und die Antworten der Peripherie „mitlesen“ werden.

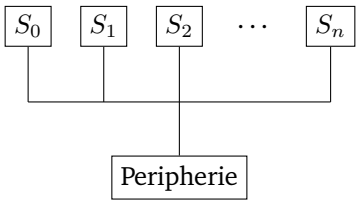


Abbildung 12.10.: Zugang zum Feldbus.

Nach diesem Aufbau gibt es nur eine physische Verbindung zu den Satellitenkomponenten. Diese Verbindung stellt damit einen Single Point Of Failure dar und kann bei Ausfall bei allen Komponenten zu Fehlverhalten führen. In diesem Aufbau wird aber davon ausgegangen, dass die Verbindung zu den Satellitenkomponenten nicht getrennt werden kann.

12.5.2. Kommunikation der Satellitensteuerungssysteme untereinander

Unabhängig vom Feldbus wird die Steuerung des Satelliten auf mehreren ZedBoards redundant laufen. Es wird dabei angenommen, dass alle Systeme dieselben Eingabedaten bekommen und unabhängig voneinander dieselben Ausgabedaten produzieren werden. Damit die Satellitensteuerungssysteme Fehler bemerken und immer einen Konsens finden können, müssen sich diese untereinander austauschen können. Dies wird durch einen Leader koordiniert. Der Leader unterscheidet sich vom Feldbus-Master in dem Sinne, dass er Systemzustandsübergänge mit den anderen Redundanzsystemen abspricht, während der Master exklusiven Zugriff auf die Satellitenperipherie bekommt[1].

Daraus ergeben sich folgende Szenarien:

Leader Election: Es wird ein Leader gewählt. Dies tritt ein, falls bisher kein Leader existiert (Systemstart), falls der Leader ausfällt oder sich fehlerhaft verhält oder falls ein anderes System Leader werden will.

Field Bus Master Election: Es wird ein Master für den Feldbus gewählt. Die Auslöser hierzu sind deckungsgleich mit dem der Leader Election.

State Change: Koordinierte Transistion in einen anderen Zustand. Dies tritt ein, wenn sich der Zustand der Satellitensteuerung ändern soll. Der Leader initiiert einen solchen Zustandswechsel. Falls die Redundanzsysteme diesem Wechsel zustimmen, wird gewechselt. Falls hierbei kein Konsens gefunden wird, müssen die fehlerhaften Systeme sich selbst untersuchen und gegebenenfalls ausgeschaltet werden.

Data Exchange: Der Koordinator fordert verarbeitete Missionsdaten von den Redundanzsystemen an. Falls verschiedene Systemkomponenten auf verschiedenen Redundanzsystemen ausfallen, können diese über ein „Data Exchange“ abgeglichen werden.

[1] Der Leader kann als Kapitän und der Bus Master als Steuermann einer Schiffsbesetzung angesehen werden.

Die beiden Election Szenarien können prinzipiell auch zusammengefasst werden, sodass der Leader auch Field Bus Master ist.

Diese Szenarien sind übliche Aufgaben von verteilten Systemen. Dabei können die eingesetzten Algorithmen Einfluss auf die Netzwerktopologie der Kommunikation der Systeme untereinander und Mindestanzahl der Redundanzsysteme haben.

Beispielsweise verlangt das Byzantine Generals Problem [35] zur Erkennung eines fehlerhaften Systems in verteilten Systemen mindestens vier Systeme. Des Weiteren benutzt auch das Projekt *Flying Laptop* vier redundante Systeme [25].

Eine alternative Fehlererkennung stellt Triple Modular Redundancy (TMR) dar, die drei redundante Steuersysteme und einen Voter benötigt. Erweiterungen von TMR können aber noch mehr redundante Systeme und Voter voraussetzen [38].

Allerdings ist es nicht möglich und sinnvoll, aufgrund von Budget, Leistungsaufnahme und Gewicht, beliebig viele ZedBoards einzusetzen. Aus diesem Grund hat sich die Projektgruppe für den Einsatz von vier ZedBoards entschieden.

Des Weiteren wird dieses Kommunikationsnetzwerk als zuverlässig angesehen. Einerseits befinden sich die Systeme physisch sehr nah aneinander, weswegen Übertragungsfehler und -abbrüche unwahrscheinlich sind, andererseits ist Fehlererkennung und -behebung bis zu einem gewissen Grad auf +niedrigeren Übertragungsschichten möglich.

13. Fehlerinjektion

In diesem Kapitel wird das Teilsystem Fehlerinjektion spezifiziert. Dazu wird zunächst das zu Grunde liegende Fehlermodell vorgestellt. Anschließend werden die Grundarchitektur mit ihren Einheiten und möglichen Injektionspunkten vorgestellt, sowie ein Überblick über Fehlerklassen und Aktivierungsmuster gegeben. Abschließend wird die Steuerung der Fehlerinjektion erläutert.

13.1. Fehlermodell

Das Fehlermodell in Abbildung 13.1 beschreibt mögliche Fehlerquellen und die daraus entstehenden Fehlerwirkungen. Daraus lassen sich Hardwarefehlertypen ableiten. Softwarefehler werden hier nicht weiter betrachtet, da diese auch aus Hardwarefehlern resultieren können.

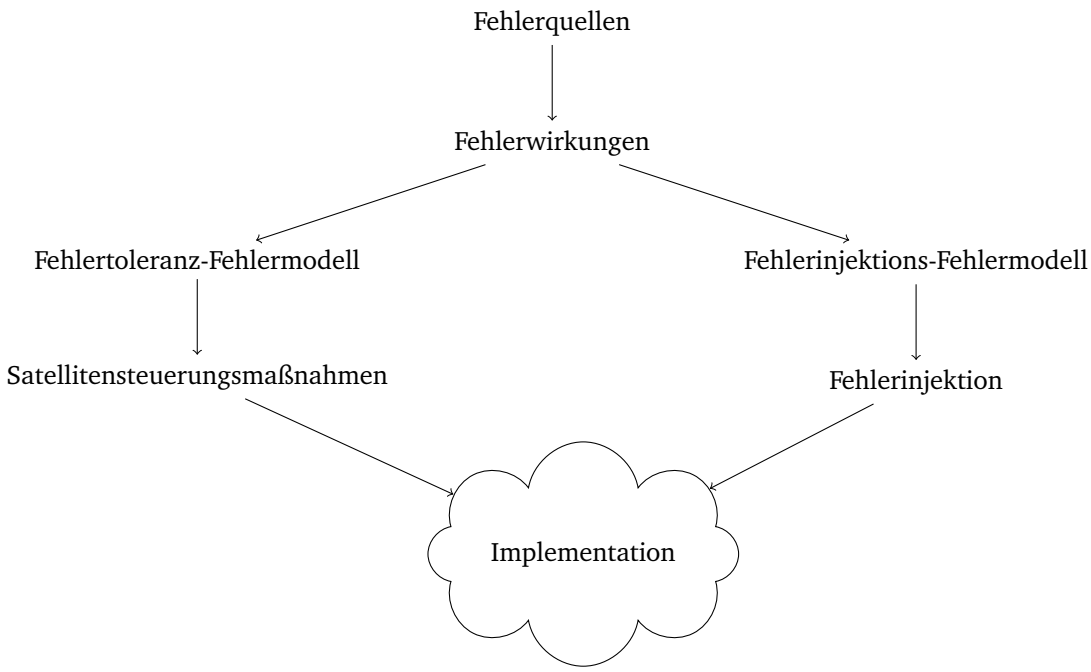


Abbildung 13.1.: Fehlermodelldiagramm

Basierend auf dem Buch „A Researcher's Guide to: Space Environmental Effects“ [40] werden als Fehlerquellen die folgenden Umwelteinflüsse betrachtet, welche im Weiteren näher beschrieben werden:

- Vakuum
- Atomarer Sauerstoff
- Ultraviolette Strahlung (UV-Strahlung)
- Partikelstrahlung
- Plasma
- extreme Temperaturen & Temperaturzyklen

- Einschläge von (Mikro-)Meteoriten

Eine weitere Fehlerquelle ist die Alterung, welche zum Auftreten von Alterungseffekten führt. Da die Implementierung von Alterung und Alterungseffekten zu aufwändig werden könnte, wird die Alterung als Fehlerquelle nicht weiter betrachtet. Es ist jedoch möglich, dass Alterungseffekte zum Beispiel durch *Delay* (Fehler F006) simuliert werden.

Im Folgenden werden die Fehlerwirkungen, der zuvor aufgeführten Fehlerquellen, aufgeführt. Dabei basieren die Fehlerwirkungen zu einem Großteil auf dem Buch *A Researcher's Guide to: Space Environmental Effects* [40] und auf der Präsentation *Radiation Effects on Electronics 101* [32], wobei letztere vor Allem die Fehlerwirkung von Partikelstrahlung, aber auch die Fehlerwirkungen von Plasma, UV-Strahlung sowie von Meteoriteneinschlägen behandelt.

Zusätzlich werden den Fehlerquellen auch die Hardware-Fehlertypen, welche am Ende dieses Abschnittes in der Tabelle 13.1 aufgelistet werden, zugeordnet. Diese Zuordnung findet nach logischem Ermessen des Autors statt.

Vakuum: Durch das Vakuum, beziehungsweise den extremen Unterdruck im All, kann es zum Ausgasen kommen. Das heißt, dass eingeschlossene Gase aus festen Stoffen austreten. Diese Gase können auf dem Satelliten kondensieren und so einen Film auf sensiblen Teilen bilden, wodurch diese korrodieren oder in ihrer Funktionalität eingeschränkt werden können. Zusätzlich können die ausgegasten Stoffe auch entzündbar sein, was zu Bränden oder Explosionen führen kann. Diese Fehlerwirkungen basieren auf dem Kapitel *Outgassing - Gaseous Materials Leading to Failure* [91] aus dem Buch *Handbook of Materials Falure analysis with case Studies from Aerospace and Automotive Industries*.

Daraus lässt sich schlussfolgern, dass durch Vakuum beziehungsweise durch ausgegaste Stoffe und deren Ablagerung die Fehler *Beschädigte Hardware* (Fehler F007), *Bridging* (Fehler F005) und *Schwankender Wert* (Fehler F002) entstehen können.

Atomarer Sauerstoff: Durch atomaren Sauerstoff kann es dazu kommen, dass Teile des Satelliten korrodieren, wodurch deren Funktionalität eingeschränkt wird oder der Stromkreislauf beschädigt wird.

Daraus lässt sich schlussfolgern, dass durch atomaren Sauerstoff und der daraus resultierenden Korrosion die Fehler *Beschädigte Hardware* (Fehler F007), *Zu geringe / hohe Stromzufuhr* (Fehler F008) und *Schwankender Wert* (Fehler F002) entstehen können.

UV-Strahlung UV-Strahlung kann bei einem Satelliten dazuführen, dass Oberflächen des Satelliten erodieren. Dadurch verschlechtern sich optische, elektrische und thermale Eigenschaften. Weiter kann durch Erosion die strukturelle Integrität verringert werden.

Daraus lässt sich schlussfolgern, dass durch UV-Strahlung die Fehler *Beschädigte Hardware* (Fehler F007) und *Delay* (Fehler F006) entstehen können.

Partikelstrahlung: Partikelstrahlung wie zum Beispiel Alphastrahlung kann mehrere Fehlerwirkungen bei einem Satelliten hervorrufen. Diese entstehen zum einen durch ionisierende und nicht-ionisierende Dosen und zum anderen aus Single-Event-Effekten. Die ionisierenden und nicht-ionisierenden Dosen können zur Degradierung von Mikroelektronik, von optischen Komponenten und von Solarzellen führen. Die Single-Event-Effekte können zur Verfälschung von Daten, Rauschen bei Aufnahmen, Systemabstürzen und

zur Beschädigung des Stromkreislaufs führen. Weiter kann es durch Partikelstrahlung zu Oberflächenerosion und zu ungewollten Aufladungen kommen. Die daraus entstehenden Fehler werden in den Abschnitten der UV-Strahlung (Erosion) und des Plasmas (ungewollte Aufladungen) genauer beschrieben.

Daraus lässt sich schlussfolgern, dass durch Partikelstrahlung die Fehler *Beschädigte Hardware* (Fehler F007), *Bit-Flip* (Fehler F003), *Stuck-At* (Fehler F001), *Schwankender Wert* (Fehler F002),*Zu geringe / hohe Stromzufuhr* (Fehler F008), *Delay* (Fehler F006) und *Logikfehler* (Fehler F004) entstehen können.

Plasma: Die Einwirkung von Plasma kann bei einem Satelliten dazu führen, dass es zu ungewollten Aufladungen kommt, was neben physikalischen Schäden auch zum Entstehen von Bias oder Impulsen bei Messwerten sowie zu Stromverlust führen kann.

Daraus lässt sich schlussfolgern, dass durch Plasma die Fehler *Beschädigte Hardware* (Fehler F007), *Bit-Flip* (Fehler F003), *Schwankender Wert* (Fehler F002) und *Zu geringe / hohe Stromzufuhr* (Fehler F008) entstehen können.

Extreme Temperaturen & Temperaturschwankungen: Temperaturschwankungen können bei einem Satelliten dazu führen, dass Komponenten durch Brüche funktionsunfähig werden, oder das kleinere Risse entstehen, welche dazu führen, dass eigentlich geschützte Stellen durch Einwirkung von atomarem Sauerstoff korrodieren. Weiter kann es durch extrem hohe Temperaturen auch zu Kurzschlüssen kommen, wenn sich zum Beispiel zwei Pins auf der Hardware derart verformen, dass sie Kontakt haben.

Daraus lässt sich schlussfolgern, dass durch extreme Temperaturen und Temperaturschwankungen die Fehler *Beschädigte Hardware* (Fehler F007) und *Bridging* (Fehler F005) entstehen können.

Meteoriteneinschläge: Durch Einschläge von vielen, teilweise sehr kleinen Meteoriten oder Weltraumschrott auf einen Satelliten können einzelne Komponenten beschädigt werden, wodurch die Funktionalität des Satelliten eingeschränkt oder der Satellit komplett zerstört wird. Außerdem kann es zu Oberflächenerosionen kommen.

Daraus lässt sich schlussfolgern, dass durch Meteoriteneinschläge die Fehler *Beschädigte Hardware* (Fehler F007) und *Bridging* (Fehler F005) entstehen können.

Im Folgenden werden neben den schon beschriebenen Hardware-Fehlern (Tabelle 13.1) auch mögliche Software-Fehler (Tabelle 13.2) aufgeführt und kurz beschrieben. Dabei basieren die Hardware-Fehler auf dem Paper *Fault Injection Techniques and Tools* [24] und die Software-Fehler auf dem Buch *Funktionsprüfung der Steuerungssoffware intelligenter technischer Produkte* [10].

Tabelle 13.1.: Hardware-Fehler

Bez.	Fehlertyp	Beschreibung
F001	Stuck-at-0/1/X	Der Wert eines Bits hängt entweder auf 0, 1 oder X fest und kann nicht mehr verändert werden. Dies kann durch Partikelstrahlung und damit zusammenhängenden ungewollten Aufladungen auftreten.
F002	Schwankender Wert	Der Wert eines Bits wechselt ungewollt zwischen 0 und 1 über einen längeren Zeitraum hin und her. Dies kann durch Vakuum, atomaren Sauerstoff, Partikelstrahlung bedingte Korrosion und Erosion, sowie durch Plasma und Partikelstrahlung bedingte Pulse auftreten.
F003	Bit-Flip	Der Wert eines Bits wurde ungewollt einmalig von 0 auf 1 oder von 1 auf 0 gesetzt, jedoch kann der Wert weiterhin verändert werden. Dies kann durch von Plasma oder Partikelstrahlung bedingte ungewollte Aufladungen auftreten.
F004	Logikfehler	Es wird eine falsche Logikoperation auf dem Logikgatter ausgeführt. Dies kann durch Partikelstrahlung bedingte Beschädigungen am Stromkreislauf auftreten.
F005	Bridging	Zwei Pins werden ungewollt zusammengeführt, wodurch ein falscher Wert entsteht. Dies kann durch vakuumsbedingte Ablagerungen von ausgegasten Stoffen und durch Hitze und (Mikro-)Meteoriteneinschlägen bedingten Verformungen auftreten.
F006	Delay	Logikoperationen oder Änderungen von Bitwerten werden Zeitverzögert durchgeführt. Dies kann durch Erosion, welche auf Grund von UV- und Partikelstrahlung auftritt, auftreten.
F007	Beschädigte Hardware	Die Hardware ist physisch beschädigt, wodurch die Funktionalität eingeschränkt wird oder komplett ausfällt. Dies kann durch alle Umwelteinflüsse entstehen.
F008	Zu geringe / hohe Stromzufuhr	Durch eine zu geringe / hohe Stromzufuhr wird die Funktionalität der Hardware eingeschränkt. Dies kann durch atomaren Sauerstoff, Plasma und Partikelstrahlung und den daraus bedingten Beschädigungen des Stromkreislaufs auftreten.

Tabelle 13.2.: Software-Fehler

Bez.	Fehlertyp	Beschreibung
Operationsfehler		
F009	Falsche Operation	Es wird entweder eine Operation mit einem falschen Wert, eine falsche Operation, zu viele Operationen oder zu wenig Operationen ausgeführt.
F010	Falsches lesen / schreiben	Ein falscher Datenwert wird gelesen oder geschrieben.
F011	Speicherüberlauf	Durch einen zu vollen Speicher werden Daten, die noch benötigt werden, überschrieben.
Zeitfehler		
F012	Timing von Operationen	Eine Operation wird zu früh oder zu spät ausgeführt.
F013	Timing von lesen / schreiben	Ein Datenwert wird zu früh, zu spät oder nicht gelesen / geschrieben.
F014	Zu langes Berechnen	Die Berechnung eines Datenwertes dauert zu lange.
Sonstige Software-Fehler		
F015	Source-Code-Veränderung	Der Source-Code wird durch äußere Einflüsse ungewollt verändert.
F016	Datenverlust	Daten gehen bei der Kommunikation von Komponenten verloren.

13.2. Die Grundarchitektur

Die Grundarchitektur des Teilsystems Fehlerinjektion besteht aus fünf Teilen:

- Die *Fehlerinjektionskomponente* auf dem Host-PC zur Steuerung der Fehlerinjektionen.
- Die *Nutzerschnittstelle* (UI) mit der Komponente zum Monitoring der Fehlerinjektionen auf dem Host-PC.
- Die drei *Verbindungen*:
  - Host-PC $\leftrightarrow$ ZedBoard zur Übertragung von Fehlerinjektionen und Zustandsladebefehlen.
  - Host-PC $\leftrightarrow$ Simulator zur Übermittlung von Fehlerdaten.
  - Mikrocontroller $\leftrightarrow$ FPGA zur Hardware-Software-Synchronisation.
- Der *Mikrocontroller* auf dem ZedBoard zur Injektion von Fehlern.
- Das *FPGA* auf dem ZedBoard zur Injektion von Fehlern.

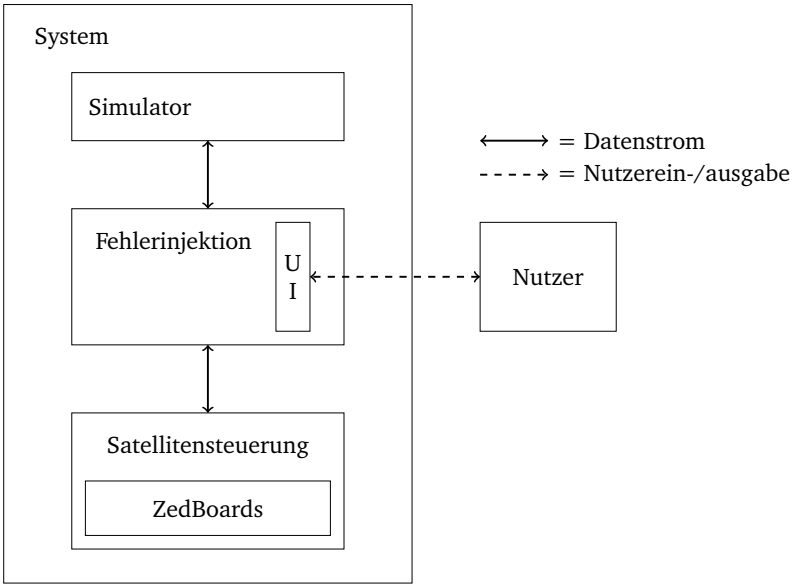


Abbildung 13.2.: Grundarchitektur der Fehlerinjektion

13.3. Abbildung der Anforderungen auf die Grundarchitektur

In diesem Abschnitt werden die Anforderungen aus 5.1.3 auf die zuvor vorgestellte Grundarchitektur abgebildet. Der Übersichtlichkeit halber geschieht dies in Tabellenform.

Tabelle 13.3.: Abbildung der Anforderungen auf die Grundarchitektur der Fehlerinjektion

Bez.	**Anforderung**	**Fehlerinjektionskomponente**
Funktionale Anforderungen		
FA030	Umweltsignale die von der Simulation an die Satellitensteuerung gesendet werden, müssen manipuliert werden können.	Host-PC, Verbindungen, Mikrocontroller
FA031	Zur Manipulation von simulierten Komponenten, wie Triebwerken, müssen die gesendeten Steuerungsbefehle von der Satellitensteuerung an die Simulation manipuliert werden können.	Host-PC, Verbindungen, Mikrocontroller
FA032	Fehler zur Manipulation des Satellitenzustands müssen von dem Benutzer an die Satellitensteuerung gesendet werden.	Host-PC, Verbindungen
FA033	Fehler zur Manipulation des Satellitenzustands müssen von der Satellitensteuerung empfangen werden.	Verbindungen, Mikrocontroller, FPGA
FA034	Empfangene Fehler zur Manipulation des Satellitenzustands müssen injiziert werden.	Mikrocontroller, FPGA

Tabelle 13.3.: Abbildung der Anforderungen auf die Grundarchitektur der Fehlerinjektion

Bez.	Anforderung	Fehlerinjektionskomponente
FA035	Es müssen Fehler in einzelne Satellitenkomponenten injiziert werden können.	Mikrocontroller, FPGA
FA036	Die Satellitenkomponenten müssen der Fehlerinjektion bekannt sein.	Host-PC, Verbindungen, Mikrocontroller, FPGA
FA038	Informationen über die Satellitenkomponenten müssen vorhanden sein.	Host-PC, Verbindungen
FA039	Die Kommunikation zwischen dem Simulator und der Satellitensteuerung muss zwischengespeichert oder abgefangen werden.	Host-PC, Verbindungen
FA040	Es muss ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellit bestehen.	Verbindungen
FA041	Ein an die Fehlerinjektionskomponente auf dem Host-PC zu sendenden Satellitenlog soll existieren.	Satellitensteuerung, Host-PC, Verbindungen
FA042	Es kann eine Systemzustandsüberwachung der Satellitensteuerung für den Fehlerinjektionszugriff erstellt werden.	Host-PC
FA043	Die Maßnahmen der Satellitensteuerung sollen dem Anwender zur Verfügung gestellt werden.	Nutzerschnittstelle (im speziellen das Monitoring)
FA044	Es soll ein Timestamp zu jedem Log-Eintrag existieren.	Verbindungen
FA045	Eine Speichereinheit aller injizierten Fehler muss vorhanden sein.	Host-PC
FA046	Auf die Speichereinheit von allen injizierten Fehlern muss zugegriffen werden können.	Host-PC
FA047	Eine UI zur Anzeige aller injizierten Fehler soll zur Verfügung gestellt werden.	Nutzerschnittstelle
FA048	Durch die Kommandozeile muss auf die Speichereinheit von allen injizierten Fehlern zugegriffen werden können.	Nutzerschnittstelle (im speziellen das Monitoring)
FA049	Der Fehlerinjektionszugriff muss auf Speicherblöcken oder durch speziell definierte Komponenten geschehen.	Mikrocontroller, FPGA
FA050	Eingabebefehle der Fehlerinjektion müssen dem Anwender zur Verfügung gestellt werden.	Nutzerschnittstelle
??	Eine GUI zur Steuerung der Fehlerinjektionen kann zur Verfügung gestellt werden.	Nutzerschnittstelle
Nicht-funktionale Anforderungen		

Tabelle 13.3.: Abbildung der Anforderungen auf die Grundarchitektur der Fehlerinjektion

Bez.	Anforderung	Fehlerinjektionskomponente
NA009	Die Nutzerschnittstelle der Fehlerinjektion soll einfach bedienbar sein.	Nutzerschnittstelle
NA010	Es sollen Fehler in unter einer Sekunde injiziert werden können.	Alle
NA011	Es soll möglichst wenig Platz auf dem FPGA gebraucht werden.	FPGA
NA012	Die Satellitenkomponenten müssen auf mögliche Fehlerquellen analysiert werden.	Mikrocontroller, FPGA
NA013	Die Auswahl an injizierbaren Fehlern muss definiert werden.	Mikrocontroller, FPGA
NA014	Die Kommunikation zwischen der Fehlerinjektion und dem Simulator und der Satellitensteuerung soll nicht länger als 1 s Verzögerung verursachen.	Alle
NA015	Die Verzögerung der Kommunikation soll möglichst konstant sein.	Alle
NA016	Der Satellitensystemzustand soll übersichtlich angezeigt werden.	Nutzerschnittstelle (im speziellen das Monitoring)
NA017	Ein Log-Eintrag soll für den Anwender direkt verständlich sein.	Nutzerschnittstelle (im speziellen das Monitoring)
NA018	Die injizierten Fehler sollen übersichtlich angezeigt werden.	Nutzerschnittstelle (im speziellen das Monitoring)

13.4. Die Aufgabenverteilung auf der Grundarchitektur

In diesem Abschnitt werden die Aufgaben der fünf Fehlerinjektionskomponenten beschrieben. Die Komponenten sind die Fehlerinjektionskomponenten auf dem Host-PC, auf dem FPGA, auf dem Mikrocontroller, sowie den Verbindungen zu den anderen Komponenten und die Nutzerschnittstelle mit der Monitoringkomponente.

13.4.1. Die Fehlerinjektionskomponente auf dem Host-PC

Die Fehlerinjektionskomponente auf dem Host-PC sitzt zwischen dem Simulator und der Satellitensteuerung auf dem ZedBoard. Die Komponente besitzt als Hauptaufgabe die Steuerung und Überwachung der Fehlerinjektion. Dazu gehört die Verarbeitung von Nutzeranfragen, das Versenden von Fehlerinjektionsanweisungen an den Mikrocontroller und Simulator, sowie das Loggen von Fehlern. Um Fehler injizieren zu können, müssen die möglichen Fehler der Satellitensteuerung dem Host-PC bekannt sein. Dies geschieht während der Initialisierung des Systems, wo der Mikrocontroller dem Host-PC mitteilt, welche Fehler injizierbar sind. Daneben besitzt die Fehlerinjektionskomponente auf dem Host-PC die Aufgabe der Weiterleitung beziehungsweise der einfachen Manipulation der Signale zwischen Simulator und Satellitensteuerung. Auch Teil dieser Komponente ist die Monitoringkomponente zur Überwachung von

Daten und zum Annehmen von Nutzeranfragen. Die Monitoringkomponente erfüllt den R-Anteil des FARM-Modells. Am Ende dieses Abschnitts wird die Monitoringkomponente genauer betrachtet.

Neben der Aufgabe der Fehlerinjektion soll der Nutzer über die Nutzerschnittstelle angeben können, ob die Simulation vorspulen darf. Dazu wird die Fehlerinjektion in den Zustand *aktiv* oder *inaktiv* gesetzt und dem Simulator den Zustand über die Host-PC - Simulator-Verbindung mitgeteilt.

Neben dem Vorspulen soll der Nutzer einen Missionsladestand laden können. Dazu muss neben dem Laden eines Simulatorstandes auf dem Simulator das ZedBoard in den gewünschten Zustand gebracht werden. Hierzu soll der Nutzer über die Nutzerschnittstelle einen Zustand auswählen und dem ZedBoard den Zustand über die Verbindung zum Mikrocontroller mitteilen. Der Zustand selbst kann durch Fehlerinjektionen beziehungsweise Initialisierungsmethoden auf dem ZedBoard geladen werden. Wie genau das Laden abläuft, bleibt den Komponenten auf dem ZedBoard überlassen und wird erst im Modulentwurf geklärt.

13.4.2. Die Fehlerinjektionskomponente auf dem Mikrocontroller

Die Fehlerinjektionskomponente auf dem Mikrocontroller soll Fehler auf dem Mikrocontroller und in den Signalen zum Host-PC, zu den Mikrocontrollern der anderen ZedBoards und zum FPGA injizieren. Da eine direkte Kommunikation des Host-PCs mit dem FPGA nicht vorgesehen wird, dient diese Komponente auch als Übermittler der Kommunikation von Host-PC und FPGA. Die Komponente soll auch Zustandsladebefehle vom Host-PC empfangen und verarbeiten.

13.4.3. Die Fehlerinjektionskomponente auf dem FPGA

Die Fehlerinjektionskomponente auf dem FPGA soll Fehler auf dem ZedBoard und in den Signalen zum Mikrocontroller injizieren. Die Komponente soll auch vom Mikrocontroller weitergeleitete Zustandsladebefehle empfangen und verarbeiten.

13.4.4. Die Verbindungen

Zur Kommunikation zwischen Simulator, Host-PC, der Fehlerinjektion auf dem Mikrocontroller und der Fehlerinjektion auf dem FPGA werden Verbindungen definiert. Von dem Host-PC zum Mikrocontroller der ZedBoards sollen Ethernet-Verbindungen zum Datenaustausch benutzt werden. Diese Verbindungen sollen die gleichen sein, die zur Kommunikation zwischen Simulator und Satellitensteuerung und zwischen den ZedBoards untereinander benutzt werden. Dadurch kann man die Fehlerinjektion als ein *man in the middle* ansehen. Die Verbindungen werden für fünf Zwecke eingesetzt:

- Zur Übertragung der *Umweltsignale* des Simulators zur Satellitensteuerung auf dem ZedBoard.
- Zur Übertragung der *Steuerungssignale* der Satellitenkomponenten von der Satellitensteuerung zum Simulator.

- Zur Übertragung des *Satellitensteuerungs-Logs* von der Satellitensteuerung zum Simulator, sowie zum Monitoring der Satellitenmaßnahmen gegen Fehler auf dem Host-PC.
- Zur Übertragung von *Fehlerinjektionsdaten* zur Injektion von Fehlern auf dem ZedBoard, von Ladebefehlen, sowie zum Monitoring der Fehler auf dem Host-PC.
- Zur Übertragung der *Satellitensteuerungsdaten* zwischen den ZedBoards. Hier soll die Fehlerinjektion in den zu sendenden und zu empfangenen Komponenten sitzen.

Für die Übertragung der vier Signale vom Host-PC zum Mikrocontroller soll nur ein Kabel verwendet werden, sodass auf Mikrocontroller- und auf Host-PC-Seite eine Kommunikationskomponente zur Zusammenfügung und Teilung von Datenpaketen gebraucht wird. In Abbildung 13.3 werden die Kommunikationssignale zur Datenübertragung gezeigt.

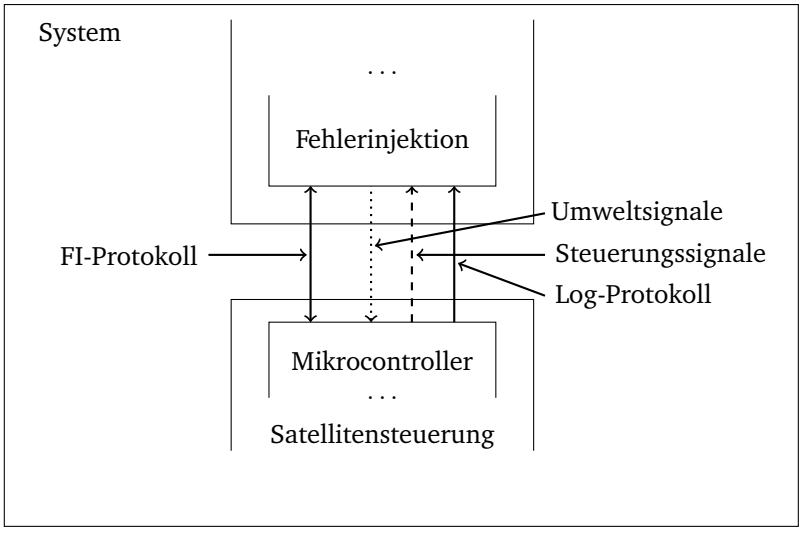


Abbildung 13.3.: Ethernetverbindung: Host-PC – Mikrocontroller

Die zweite Verbindung (siehe Abbildung 13.4) wird zur Kommunikation zwischen der Fehlerinjektion auf dem Host-PC und den Simulator benötigt. Neben den schon aus der ersten Verbindung bekannten Datensignalen, ist das Fehlerinjektionsprotokoll bei dieser Verbindung nur zur Übertragung des Fehlerinjektionszustands des Host-PCs, sowie der Übertragung der zu injizierenden Fehler zuständig. Diese Daten werden übertragen, damit der Simulator das Vorspulen der Mission steuern kann.

Die dritte und letzte Verbindung (siehe Abbildung 13.5) wird zur Kommunikation zwischen der Fehlerinjektion auf dem FPGA und der Fehlerinjektion auf dem Mikrocontroller gebraucht. Auch diese Verbindung ähnelt der Kommunikation zwischen der Satellitensteuerung auf dem Mikrocontroller und dem FPGA. Zur Übertragung der Daten werden zwei Wege benötigt:

- Die Übertragung von *Hardware-zu-Software-Signalen*. Dazu werden Register von der Hardware beschrieben, die von der Software ausgelesen werden können.
- Die Übertragung von *Software-zu-Hardware-Signalen*. Dazu werden Register von der Software beschrieben, die von der Hardware ausgelesen werden können.

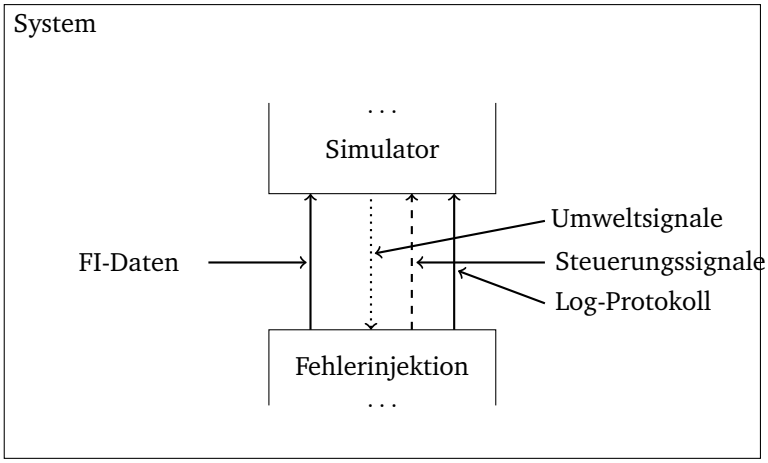


Abbildung 13.4.: Verbindung: Host-PC – Simulator

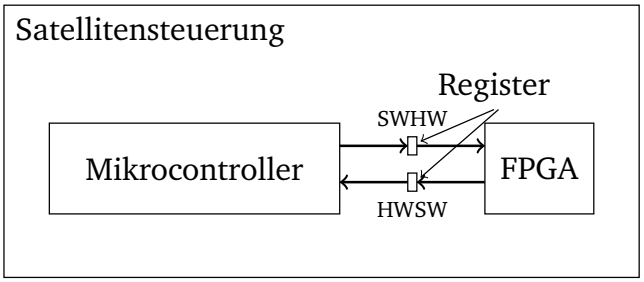


Abbildung 13.5.: Hardware-Software-Kommunikation

13.4.5. Die Nutzerschnittstelle

Das User Interface, also die Nutzerschnittstelle, stellt dem Benutzer Funktionen zum Injizieren von Fehlern und der Systemüberwachung zur Verfügung. Ihr Aufbau ist zweigeteilt, zum einen gibt es die Steuerungskomponente, zum anderen die Monitoring-Komponente.

Steuerung

In der Steuerungskomponente kann der Benutzer Fehler auswählen und diese aktivieren. Sie werden daraufhin ins System injiziert. Es lassen sich dabei verschiedene Methoden unterscheiden, Fehler zu injizieren.

1. **Fehlerinjektion aktivieren / deaktivieren:** Um Fehler injizieren zu können muss die Fehlerinjektion aktiviert sein. In bestimmten Fällen können keine Fehler injiziert werden, z. B. wenn das Spiel gerade vorgespult wird.

2. **Ein- und Ausschalten von Fehlern:** Ein Fehler kann ein- bzw. ausgeschaltet werden. Als Beispiel kann hier ein Sensor dienen, der seine Funktion durch das injizieren eines solchen Fehlers verliert.

3. **Fehler mit mehreren Optionen:** Ein Fehler kann in verschiedenen Stärken auftreten. Dabei kann sowohl eine feste Abweichung definiert werden, beim Temperatursensor beispielsweise um $5°$. Weiter kann ein Intervall festgelegt werden, innerhalb dessen zufällige Werte gewählt werden, um Rauschen zu verursachen.

4. **Fehler mit Parametern:** Einem Fehler kann ein Parameter übergeben werden. Als Beispiel dient erneut der Temperatursensor. Wird ein solcher Fehler injiziert, z. B. mit dem Parameter 20, so zeigt der Temperatursensor nur noch $20°$ an.

5. **Satellitenzustand laden:** Wenn in der Simulationsumgebung ein Spielstand geladen wird, so muss anschließend auch der Satellitensteuerungszustand angepasst werden. Dies geschieht, indem auch hier ein Zustand geladen wird.

6. **Fehlerexperimente injizieren:** Der Benutzer kann ein Skript zur automatisierten Fehlerinjektion erstellen und ausführen.

Ein Fehlerexperiment wird in Form eines Skriptes zur Verfügung gestellt, das sequenziell abgearbeitet werden soll. Man könnte ein Skript auch als Menge von Fehlerinjektionen modellieren. Dies hätte aber den Nachteil, dass man die genauen Zeitpunkte der Injektionen bei jeder Fehlerinjektion angeben muss, was bei der sequenziellen Modellierung implizit enthalten ist. Auch gehen wir davon aus, dass der Ersteller bei der sequenziellen Modellierung weniger schnell den Überblick verliert, als bei der Mengenmodellierung.

Monitoring

Die Fehlerinjektionskomponente auf dem Host-PC hat zur Aufgabe, die Fehlerinjektion zu steuern und zu überwachen. Dies geschieht über eine Nutzerschnittstelle *(User Interface)*. Diese wiederum beinhaltet eine Monitoringkomponente, mit deren Hilfe der Nutzer verschiedene Daten, wie z. B. die Geschwindigkeit, die Flughöhe, usw. abrufen kann. Vor allem aber soll diese Komponente dazu dienen, injizierte Fehler auf ihre Wirksamkeit zu prüfen. Die Monitoringkomponente muss die folgenden Eigenschaften besitzen:

- Anzeigen der Umweltdaten
- Anzeigen der Steuersignale
- Anzeigen des Satellitenzustands
- Anzeigen der zur Zeit injizierten Fehler
- Anzeigen der Auswirkungen der injizierten Fehler
- Anzeigen des Fehlerinjektionszustandes
- Unterscheiden zwischen eingehenden und ausgehenden Daten

Anzeigen der Fehlerinjektion meint, dass erkennbar ist, ob die Fehlerinjektion gerade aktiv ist. Dies kann beispielsweise durch verschiedene Farben umgesetzt werden.

Mit dem Unterschied zwischen eingehenden und ausgehenden Daten ist dabei gemeint, dass neben dem injizierten Fehler auch seine Auswirkungen zu erkennen sind. Als Beispiel wird ein injizierter Fehler in einem Temperatursensor betrachtet. Eingehend liefert dieser einen Wert von $20°$, durch den injizierten Fehler werden aber z. B. $0°$ angezeigt. Im Monitoring werden nun beide Werte angezeigt, wobei der zweite Wert markiert wird.

13.5. Fehlerklassen und injizierbare Fehler

Fehlertypen lassen sich aufgrund gemeinsamer Eigenschaften zu Fehlerklassen zusammenfassen. Die Fehlertypen aus Abschnitt 13.1 werden nun, nach Fehlerklassen geordnet, aufgelistet und einem injizierbaren Fehler zugeordnet, wenn der Fehlertyp nicht selbst den Fehler darstellt.

Eine Fehlerklasse ist eine Menge an Fehlertypen, welche auf Grund einer bestimmten Eigenschaft dieser zugeordnet werden. Dabei gibt es zu jeder Fehlerklasse eine Komplement-Fehlerklasse, welche genau die Fehlertypen enthält, welche in der anderen Fehlerklasse nicht vorhanden sind.

Mögliche Eigenschaften zur Einteilung sind Persistenz, also ob ein Fehler permanent oder transient ist oder der phänomenologische Grund, also ob ein Fehler durch Menschen (oder Aliens) verursacht wurde oder durch einen natürlichen Grund wie radioaktive Strahlung ausgelöst wird.

Hier findet jedoch eine Aufteilung der Fehlertypen nach der Dimension, also nach Software-Fehlern (Tabelle 13.5) und Hardware-Fehlern (Tabelle 13.4), statt.

Im Folgenden werden die Fehlertypen und die Art ihrer Injektion aufgeführt, welche durch die Fehlerinjektion injiziert werden sollen. Von den bereits in Abschnitt 13.1 aufgeführten Fehlern werden die zu geringe Stromzufuhr (Fehlertyp F008), sowie die Source-Code-Veränderung (Fehlertyp F015) nicht umgesetzt, da diese zum einen durch andere Fehlertypen dargestellt werden können oder, ähnlich wie eine vorsätzliche Manipulation, sehr unwahrscheinlich sind.

Neben dem Auftreten von einzelnen Fehlern ist auch das Auftreten von mehreren Fehlern gleichzeitig möglich, sofern diese umsetzbar sind. So kann es zum Beispiel sein, dass ein Bit Stuck-at-0 (Fehlertyp F001) ist und es wird eine falsche Operation ausgeführt. Nicht möglich ist jedoch, dass ein Bit Stuck-at-0 ist und gleichzeitig das selbe Bit einen schwankenden Wert hat. Außerdem ist es möglich, dass Operationen falsch ausgeführt werden (Fehlertyp F009), indem die gemessenen Werte mit einer festen Abweichung, einem Rauschen oder beidem belegt werden. Nicht möglich ist es jedoch, dass auf den gleichen Wert zwei Feste Abweichungen gelegt werden.

Das Auftreten von einzelnen Fehlern wird als *single-point fault* und das von mehreren Fehlern wird als *multi-point fault* bezeichnet.

Tabelle 13.4.: Injizierbare Hardware-Fehler

Bez.	**Fehlertyp**	**Injektion**
F001	Stuck-at-0/1/X	
F002	Schwankender Wert	
F003	Bit-Flip	
F004	Logikfehler	
F005	Bridging	Wird durch zwei Stuck-at-Fehler simuliert.
F006	Delay	
F007	Beschädigte Hardware	Wird durch Stuck-at-Fehler und schwankende Werte simuliert.

Tabelle 13.5.: Injizierbare Software-Fehler

Bez.	Fehlertyp	Injektion
Operationsfehler		
F009	Falsche Operation	
F010	Falsches lesen/ schreiben	Wird durch falsche Operation simuliert.
F011	Speicherüberlauf	Wird durch falsche Operation simuliert.
Zeitfehler		
F012	Timing von Operationen	
F013	Timing von lesen/ schreiben	Wird durch Timing von Operationen simuliert.
F014	Zu langes Berechnen	Wird durch Timing von Operationen simuliert.
Sonstige Software-Fehler		
F016	Datenverlust	

13.6. Injektionspunkte

Als Injektionspunkte werden die Punkte bezeichnet, an denen Fehler injiziert werden können. In diesem System wird es drei Injektionspunkte geben:

1. Fehlerinjektion auf dem Host-PC
2. Fehlerinjektion auf dem Mikrocontroller
3. Fehlerinjektion auf dem FPGA

Diese drei Injektionspunkte sollen nun weiter erläutert werden. In Abbildung 13.2 lässt sich erkennen, wo sich die einzelnen Fehlerinjektionspunkte befinden. Der erste Injektionspunkt ist auf dem Host-PC zu finden. Hier lassen sich bereits Umweltdaten und Steuersignale manipulieren. Die Fehlerinjektionskomponente dient außerdem der Fehlereingabe und dem Monitoring (vgl. Abschnitt **??**). Der zweite Injektionspunkt befindet sich auf dem ZedBoard, genauer auf dem FPGA. Hier können Fehler in die Satellitenkomponenten injiziert werden. Hervorzuheben ist, dass, auch wenn von einem ZedBoard die Rede ist, mehrere, redundante ZedBoards gemeint sind. Ähnlich verhält es sich mit dem dritten Injektionspunkt, der sich ebenfalls auf dem ZedBoard befindet, diesmal jedoch auf dem Mikrocontroller. Da dieser als Zwischenkomponente für die Kommunikation zwischen FPGA und dem Host-PC fungiert, können hier Fehler in die Signale eben dieser Komponenten injiziert werden.

13.7. Aktivierungsmuster

Dieser Abschnitt behandelt im ersten Teil eine Klassifizierung der Aktivierungsmuster mit denen die Fehler injiziert werden sollen. Im zweiten Teil wird behandelt, wie man einen Fehler von mehreren Aktivierungsmustern abhängig macht.

Die Aktivierungsmuster geben an, zu welchem Zeitpunkt ein Fehler injiziert werden soll. Sie beschreiben die Menge A des FARM Konzepts. Da das zu testende System die Satellitensteuerung ist, kann ein Fehler von den Zuständen der Hardware- und Softwarekomponenten der Satellitensteuerung, von der Hardware-Software-Kommunikation, von den Aktionen des Systems auf andere Satellitenkomponenten und von der relevanten Umgebung abhängig sein.

Mit Aktionen der Satellitensteuerung sind die Befehle an die anderen Komponenten des Satelliten gemeint, wie zum Beispiel Zündbefehle eines Triebwerks. Als Aktionen der relevanten Umgebung sind dann (vorverarbeitete) Messungen von Sensoren oder Zustände der Satellitenkomponenten gemeint. Die Hardware-Software-Kommunikation besteht aus den Eingangs- und Ausgangssignalen zwischen dem Mikrocontroller und dem FPGA.

Ein Fehler soll zudem auch aktiviert werden, wenn ein anderer Fehler aktiviert wird. Das ist notwendig, damit mehrere Fehler zum gleichen Zeitpunkt injiziert werden können. Es reicht theoretisch aus, mehrere Fehler mit dem gleichem Aktivierungsmuster zu injizieren, aber praktisch könnte es sein, dass sie nur sequenziell abgearbeitet werden. Dies stellt ein Problem dar, da in der Zeit die Satellitensteuerung seinen Zustand geändert haben kann.

Neben diesen Aktivierungsmustern kann ein Fehler noch von der abgelaufenen Zeit abhängig sein. Die Zeit ist als Aktivierungsmuster wichtig, da sie Prozesse betrachten kann, die in einer Modellierung des Fehlerinjektionssystemes zu detailliert oder zu anwendungsspezifisch wären. Ein Beispiel ist ein Bitflip einer Speicherzelle, der eine Sekunde später auftaucht als ein Bitflip einer anderen Speicherzelle. Dabei ist es dem Nutzer der Fehlerinjektion nicht wichtig, ob dieser Bitflip von einer radioaktiven Strahlung kam oder durch einen Hardwarefehler beim Schreiben der Speicherzelle sondern nur, dass der Speicherfehler nach einer bestimmten Zeit nach dem letzten Bitflip passiert. Für die Zeit braucht man zudem ein Einheitensystem. Nach dem Stand der Technik ist es nicht machbar, ein Ereignis genau nach 10 Pikosekunden auszuführen, somit macht eine so genaue Skalierung der Zeit keinen Sinn. Für das FPGA des ZedBoards braucht man mindestens eine Schrittweite der Zeit von 10 Nanosekunden, was einem FPGA-Takt entspricht. Ob dies für eine Fehlerinjektion realistisch ist, hängt allerdings stark von dem modellierten System ab und lässt sich jetzt noch nicht klären. Für die Betrachtung der Zeit auf Softwareebene ist auch eine Skalierung der Zeit auf 10 ns Schrittweite zu wenig, da eine Injektion durch eine Unterbrechungsroutine länger dauert und man nicht sagen kann, ob zusätzlich die Fehlerinjektion unterbrochen wird. Durch die oft verwendete Uhrentaktung von einer Millisekunde auf Betriebssystemebene macht eine Skalierung der Zeit bei Software auf Millisekundenebene Sinn. Hier muss man allerdings beachten, dass die Injektion auch unterbrochen werden kann, sodass eine genaue Injektion nicht anzunehmen ist. Der Nutzer sollte somit bei der Injektion die Zeit als Mindestzeit und als Aktivierungsmuster ansehen, anstatt als genaue Zeit. Dies sollte allerdings kein Problem für den Nutzer sein, da eine Injektion abhängig von der Zeit keinerlei Aussage über den Satellitenzustand gibt und somit der Zustand nur von zweitrangiger Bedeutung ist. Die gleiche Begründung lässt sich bei der Skalierung der Zeit auf satellitensteuerungsübergreifender Kommunikationsebene geben. Also für die Kommunikation der Steuerbefehle und Umweltsignale.

Somit lassen sich die Aktivierungsmuster in folgende Kategorien unterteilen:

Kein Aktivierungsmuster: Natürlich sollte ein Aktivierungsmuster optional sein.

Zeit: in Takten auf Hardwareebene und in Millisekunden auf Software- und Kommunikationsebene.

Fehler: Fehler als Aktivierungsmuster

Aktionen der relevanten Umgebung: Umweltsignalwerte

Aktionen der Satellitensteuerung: Steuerungsbefehle

Zustand der Hardwarekomponenten der Satellitensteuerung: FPGA

Zustand der Softwarekomponenten der Satellitensteuerung: Mikrocontroller

Hardware-Software-Kommunikation FPGA $\rightarrow$ Mikrocontroller und
FPGA $\leftarrow$ Mikrocontroller

Nun ist es interessant, wie man einen Fehler von mehreren Aktivierungsmustern abhängig macht. Als Beispiel könnte man sich vorstellen, dass ein Bitflip im Speicher des FPGAs nach Eintreffen des Batteriezustandes (Umweltsignal) und nach dem Wechseln des Mikrocontrollers in den Stromsparmodus (Zustand der Softwarekomponente) injiziert werden soll. Das ist mit der bisherigen Klassifizierung noch nicht möglich. Es fehlen Verknüfungsoperatoren auf den Aktivierungsmustern. Es schien für dieses Projekt sinnvoll, die folgenden Operatoren für Aktivierungsmuster A, B und $n \in \mathbb{N}$ aus Tabelle 13.6 zu wählen. Der Operator A^n wird dabei nur als Abkürzung von $(A \cdot A \cdot \ldots \cdot A)$ verstanden. Als Auswertungsreihenfolge der Operatoren wird die Reihenfolge aus Tabelle 13.7 gewählt.

Syntax	Semantik
$A \wedge B$	A ist zum *gleichen* Zeitpunkt wie B aktiviert.
$A \vee B$	A oder B ist aktiviert.
$\neg A$	A ist nicht aktiviert.
$A \cdot B$	B wird gleichzeitig oder nach A aktiviert.
A^n	A wird n mal aktiviert.

Tabelle 13.6.: Aktivierungsmuster-Operatoren

Priorität	Symbol
1	$\neg$
2	$\cdot$
3	$\wedge$
4	$\vee$

Tabelle 13.7.: Auswertungsreihenfolge der Operatoren

Der Begriff $A \wedge B$ ist für den Zustand der Satellitensteuerung eindeutig durch die Werte der Speicherzellen definiert, während es für die Aktivierungsmuster *Zeit* , *Aktionen der relevanten Umgebung* und *Aktionen der Satellitensteuerung* nicht definiert ist. Folgende Definitionen werden genutzt:

- Ein *Umweltsignalwert* A ist genau dann mit einem anderen aktivierten Aktivierungsmuster B aktiviert, solange kein neuer Umweltsignalwert eingetroffen ist.
- Ein *Steuerbefehl* A ist genau dann mit einem anderen aktivierten Aktivierungsmuster B aktiviert, solange kein neuer Steuerbefehl gesendet wurde.
- Für die Zeit als Aktivierungsmuster A bedeutet das, dass zu genau dem Takt oder genau in der Millisekunde das Aktivierungsmuster B aktiv ist. Meistens ist es eher gewollt, das ein Fehler nach einer gewissen Zeit und dann nach Auftreten eines Aktivierungsmuster B injiziert wird. Dafür wird die Verknüpfung $A \cdot B$ genommen.

Syntax	Semantik
T	Der Wertebereich der Wartezeiten in Millisekunden oder ZedBoard-Takten.
aF	Die Menge der gleichzeitig zu injizierenden Fehler.
U	Die Menge der Umweltsignale, die vom Simulator zum Mikrocontroller gesendet werden.
S	Die Menge der Steuersignale, die vom Simulator zum Mikrocontroller gesendet werden.
M	Die Menge der Zustandssignale von dem Mikrocontroller.
$FPGA$	Die Menge der Zustandssignale von dem FPGA.
$SWHW$	Die Menge der Signale, die zur Synchronisation von Hardware und Software vom Mikrocontroller beschrieben werden.
$HWSW$	Die Menge der Signale, die zur Synchronisation von Hardware und Software vom FPGA beschrieben werden.

Tabelle 13.8.: Aktivierungsmuster-Tabelle

$\langle S \rangle \quad \rightarrow \varepsilon \mid \langle Exp \rangle$

$\langle Exp \rangle \rightarrow \langle Aktiv \rangle \mid (\langle Exp \rangle)$
$\quad \mid \quad \neg \langle Exp \rangle \mid \langle Exp \rangle \langle N \rangle$
$\quad \mid \quad \langle Exp \rangle \wedge \langle Exp \rangle \mid \langle Exp \rangle \vee \langle Exp \rangle \mid \langle Exp \rangle \cdot \langle Exp \rangle$

$\langle N \rangle \quad \rightarrow {}^{1} \mid {}^{2} \mid {}^{3} \mid \ldots$

Abbildung 13.6.: Grammatik zur Erzeugung von Aktivierungsmustern

Für das Konstruieren von Aktivierungsmustern kann am besten eine Sprache benutzt werden. Zum Definieren der Sprache muss erst das Alphabet bestimmt werden. Das Alphabet Σ besteht aus den Mengen der Aktivierungsmuster $Aktiv$ (Tabelle 13.8), den Operatoren OP und den $(,)$ Symbolen zum Schachteln von Aktivierungsmustern.

$$Aktiv := T \cup aF \cup U \cup S \cup M \cup FPGA \cup SWHW \cup HWSW$$

$$OP := \{\wedge, \vee, \neg, \cdot\} \cup \{{}^{1}, {}^{2}, {}^{3}, {}^{4}, \ldots\}$$

$$\Sigma := Aktiv \cup OP \cup \{(,)\}$$

Die Sprache wird durch die EBNF-Grammatik in Abbildung 13.6 erzeugt. Das Nichtterminal $\langle Aktiv \rangle$ besteht aus Elementen der oben definierten Menge $Aktiv$. Eine Einführung von prädikatenlogischen Operatoren, wie $\exists$, erschien nicht sinnvoll, da Wertabfragen auf mehreren Speicherzellen auch durch mehrere Verknüpfungen von $\vee$ modelliert werden können und als nicht häufig eingesetzte Funktion geschätzt wurde.

13.8. Steuerung von Fehlerinjektionen

Die Steuerung einer Fehlerinjektion hat als Aufgabe die Vorbereitung eines bestimmten Fehlers auf seine Injektion, die Aktivierung der Fehlerinjektion und das Freigeben der Fehlerinjektionsressourcen.

Für das Freigeben werden die zusätzlichen Eigenschaften der Häufigkeit und der Fehlerdauer gebraucht, sodass diese vor der Beschreibung der Phasen definiert werden. Zuletzt wird in

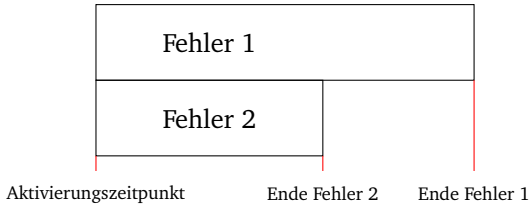


Abbildung 13.7.: Zwei Fehler mit unterschiedlichen Dauern

diesem Abschnitt auf die Abbildung der Fehler und Aktivierungsmuster auf dessen Injektionspunkte eingegangen.

Neben der Einteilung in Fehlerklassen müssen Fehler zur Steuerung die Eigenschaften *Häufigkeit*, *Fehlerdauer* und *Reaktivierungsanzahl* besitzen können. Mit Häufigkeit ist gemeint, wie oft ein Fehler injiziert werden soll. Mit der Fehlerdauer soll der Nutzer angeben können, wie lange ein bestimmter Fehler in der Kommunikation anhält. Die Fehlerdauer ist nicht als Eigenschaft für einmal auftauchende Fehler, wie zum Beispiel Fehler auf Speicherzellen, vorhanden. Die Reaktivierungsanzahl gibt an, wie oft die Fehlerinjektion mitsamt Aktivierung wiederholt werden soll, damit man sich das wiederholte Übersenden von Befehlen spart.

Man fragt sich gegebenenfalls, warum die Fehlereigenschaften nicht als Teil der Aktivierungsmuster definiert werden, sodass ein Aktivierungsmuster kein Zeitpunkt, sondern ein Zeitintervall ist. Dies hätte den Nachteil, dass sich ein Fehler nur ein Aktivierungsmuster mit einem anderen Fehler teilen könnte, wenn sie die gleiche Häufigkeit und Fehlerdauer besitzen. Die beiden Fehler in Abbildung 13.7 könnten sich kein Aktivierungsmuster teilen.

Der Ablauf der Fehlerinjektion lässt sich grob in drei Phasen unterteilen:

Die **Injektionsvorbereitung** bereitet die Fehlerinjektion direkt nach dem Erhalt des Injektionsbefehles von dem Nutzer vor. Dazu wird der Fehler und dessen Zustand (siehe Fehlerklassen **??**) auf dem Host-PC abgespeichert, um den Nutzer über die Fehlerinjektion zu informieren. Die genauen Fehlerinjektionszeiten werden dem Simulator gesendet, um das Vorspulen des Simulators zu steuern. Danach wird der Injektionspunkt anhand der Nutzerinformationen bestimmt. Dies kann als Injektionspunkt der Host-PC, der Mikrocontroller oder das FPGA sein. Nach Bestimmung des Ortes müssen die relevanten Daten zur Vorbereitung über das Fehlerinjektionsprotokoll an den Injektionspunkt gesendet werden. Für das FPGA muss hier das Fehlerinjektionsprotokoll vom Host-PC zum Mikrocontroller und dann vom Mikrocontroller zum FPGA gewählt werden. Wie im Detail die Vorbereitung am Injektionspunkt, wie Speicherbelegung und Aktivierungslistener, abgehandelt wird, wird erst im System- und Modulentwurf behandelt. Den Erfolg oder Misserfolg der Vorbereitung der Fehlerinjektion wird an den Host-PC zurückgeliefert. Ein Misserfolg ist dabei eine bereits verwendete Fehlerinjektionsleitung eines Fehlers. Zum Beispiel soll ein Sensorsignal nicht gleichzeitig von einer Fehlerinjektion den Wert x zugewiesen bekommen und von einer anderen Fehlerinjektion den Wert y. Durch diese Einschränkung wird die Anzahl der möglichen injizierbaren Fehler nicht kleiner, da ein Signal immer nur einen Wert haben kann. Der Vorteil ist hier aber, dass der Aufwand bei der Implementierung gesenkt wird. Die Rückmeldung wird vom Host-PC verarbeitet und entsprechend dem Nutzer angezeigt.

Die **Aktivierung** der Fehler entsteht durch das Auslösen der Aktivierungsmuster. Wie im Abschnitt **??** beschrieben, können die Aktivierungsmuster von diversen Signalen und Zuständen von Mikrocontroller und FPGA zusammengesetzt sein. Daher müssen bei einer Aktivierung mit Abhängigkeiten von Mikrocontroller und FPGA sowohl Informationen von Mikrocontroller und FPGA zeitig eintreffen, damit in der Zeit nichts Weiteres passiert. Dies wird durch Unterbrechungen auf dem Mikrocontroller gelöst. Für das FPGA wird kein weiteres Konzept benötigt, da es durch die Parallelität und Taktung deutlich schneller ist als der Mikrocontroller. Das Einzige, was bei einer Injektion auf dem FPGA dem Nutzer bewusst sein muss, ist, dass das FPGA nicht zur Fehlerinjektion unterbrochen werden kann. Je nach Dauer der Fehleraktivierung kann die Injektion verzögert sein. Diese Dauer sollte, wenn überhaupt, jedoch die Anzahl der möglichen Fehlerinjektionen nicht stark verringern.

Das **Freigeben der Fehlerinjektionsressourcen** wird nach Beendigung der Fehlerinjektion durchgeführt. Das Ende einer Fehlerinjektion ist erreicht, wenn der injizierte Fehler seine angegebene Häufigkeit oder Zeitdauer überschritten hat oder die Fehlerinjektion vom Benutzer beendet wurde. Falls der Fehler keine Häufigkeit oder Zeitdauer besitzt, ist das Ende sofort nach dem Injizieren erreicht. Der Host-PC und der Nutzer werden über das Freigeben der Fehlerinjektionsressourcen informiert.

Nun wird das Festlegen des Injektionspunktes aus der ersten Phase genauer betrachtet. Als Festlegung der Injektionspunkte wird folgende Abbildung gewählt:

$$Map_{F \times A \to I}(f, a) : Fehlermenge \times Aktivierungsmuster \to Injektionspunkte$$

Für ein Tupel $f \times a$ kann somit der Injektionspunkt ermittelt werden. Bevor auf die genaue Abbildung eingegangen wird, wird der Definitionsbereich und Wertebereich betrachtet. Für den Definitionsbereich $Fehlermenge$ ist ein Fehler ein Element aus der in 13.1 definierten Fehlermenge (siehe auch Tabelle 13.9 für die Bedeutungen der Mengen). Dadurch kann ein Fehler aus der Menge der Umweltsignale des Simulators, aus der Menge der Steuersignale zum Simulator, aus der Menge der Zustände der Hardware oder Software oder aus der Menge der Hardware-Software-Kommunikation sein. Wie im Detail der Fehler aussieht, ist für die Abbildung unwichtig.

$$F := U \cup S \cup M \cup FPGA \cup SWHW \cup HWSW \tag{13.1}$$

Das Gleiche wird für die Aktivierungsmuster gemacht. Dazu wird die Kategorien der Aktivierungsmuster aus Abschnitt **??** genommen. Die Bedeutung der Mengen wird in Tabelle 13.8 beschrieben. Die folgende Menge $Aktiv$ beschreibt die Aktivierungsmuster:

$$Aktiv := T \cup aF \cup U \cup S \cup M \cup FPGA \cup SWHW \cup HWSW \tag{13.2}$$

Die Menge der Aktivierungsmuster ist dann durch folgende Menge definiert:

$$A := \text{Menge der Wörter, welche aus der in Abschnitt **??** Sprache erzeugt werden} \tag{13.3}$$

Bez.	Beschreibung
U	Die Menge der Umweltsignale, die vom Simulator zum Mikrocontroller gesendet werden.
S	Die Menge der Steuersignale, die vom Mikrocontroller zum Simulator gesendet werden.
M	Die Menge der Zustandsfehler vom Mikrocontroller. Beispielsweise Korruption von Speicherzellen.
$FPGA$	Die Menge der Zustandsfehler vom FPGA.
$SWHW$	Die Menge der Signale, die zur Synchronisation von Hardware und Software vom Mikrocontroller beschrieben werden.
$HWSW$	Die Menge der Signale, die zur Synchronisation von Hardware und Software vom FPGA beschrieben werden.

Tabelle 13.9.: Fehlermengen-Tabelle

Kombination der Injektionspunkte
Fehlerinjektion auf dem Host-PC
Fehlerinjektion auf dem Mikrocontroller
Fehlerinjektion auf den FPGAs
Fehlerinjektion auf dem Mikrocontroller und auf dem FPGA

Tabelle 13.10.: Tabelle der Kombination der Injektionspunkte

Der Wertebereich wird in die Punkte Host-PC, Mikrocontroller, FPGA und Mikrocontroller $\wedge$ FPGA unterteilt. Es ist anzumerken, dass die Injektionspunkte der Fehler, die zeitgleich injiziert werden sollen, zusammen nicht aus der Potenzmenge der Injektionspunkte bestehen. Ein Fehler in den Umweltsignalen oder Steuersignalen soll auf dem Mikrocontroller injiziert werden, anstatt auf dem Host-PC. Das wird gemacht, damit die Signalverzögerung nicht zu hoch ist. Somit kommt als einzige Kombination aus Injektionspunkten der Mikrocontroller $\wedge$ FPGA als Teil der Potenzmenge von den ersten drei Injektionspunkten hinzu (siehe Tabelle 13.10).

Bevor auf die Abbildung eingegangen wird, ist es wichtig zu Wissen, wo die Aktivierungsmuster überprüft werden. Während die Zustände, Hardware und Software, nur direkt auf der Hardware und Software überprüft werden können, kann man die Aktionen der relevanten Umgebung und die Aktionen der Satellitensteuerung auf dem Mikrocontroller und auf dem Host-PC überprüfen. Für ein Umgebungssignal aus den Aktionen der relevanten Umgebung kann es je nach Fehler interessant sein, ob das Umgebungssignal zur gleichen Zeit ankommt, wie der Fehler injiziert werden soll. Daher muss der Zeitpunkt des Eintreffens des Umweltsignals auf dem Mikrocontroller genutzt werden und somit auch von dem Mikrocontroller aus überprüft werden. In anderen Fällen ist der Injektionszeitpunkt nicht so wichtig und kann auch auf dem Host-PC überprüft werden. Dadurch muss keine Unterbrechung auf dem Mikrocontroller ausgeführt werden und vereinfacht die Kommunikation für die Injektion. Es lässt sich sagen, dass je nach Aktion der relevanten Umgebung anders entschieden werden muss. Die selbe Begründung lässt sich für Steuerbefehle geben. Für die Hardware-Software-Kommunikation kann man die Aktivierungsmuster und Fehler sowohl auf Hardwareseite als auch auf Softwareseite messen und injizieren. Während es auf Softwareseite Sinn macht, die Kommunikationsrichtung Software zu Hardware zu manipulieren, um zum Beispiel Fehler für lange Zeiten zu manipulieren, müssen auch auf Hardwareseite die Signale manipuliert werden können. Dies muss gemacht werden, wenn die Signale zeitgenau zu dem Zustand der Hardware injiziert werden

sollen. Für die Rückrichtung Hardware zu Software könnte man auch auf beiden Seiten Fehler injizieren. Hier macht die Manipulation auf Softwareseite wenig Sinn, denn die Weitergabe der Injektionsdaten zu der Hardwareseite erzeugt den gleichen Effekt und nimmt wahrscheinlich weniger Zeit des Mikrocontrollers ein. Somit kann ein Hardware-zu-Software-Signal genauso wie der Zustand des FPGAs behandelt werden.

Die folgende Abbildung erschien sinnvoll:

$$
Map_{F\times A\to I}(f,a) = \begin{cases}
Host-PC & \text{wenn } f = U \wedge a \text{ nur abhängig von } U \\
 & \vee f = S \wedge a \text{ nur abhängig von } S \\
Mikrocontroller & \text{wenn } FPGA \notin f \wedge HWSW \notin f \wedge \\
 & (\ U \in f \wedge a \text{ nicht nur abhängig von } U \\
 & \ \vee S \in f \wedge a \text{ nicht nur abhängig von } U \\
 & \ \vee U \in f \wedge S \in f \\
 & \ \vee M \in f \\
 & \ \vee SWHW \in f \wedge a \text{ nicht abhängig von} \\
 & \quad FPGA \wedge a \text{ nicht abhängig von } HWSW\ ) \\
FPGAs & \text{wenn } U \notin f \wedge S \notin f \wedge M \notin f \wedge \\
 & (\ FPGA \in f \\
 & \ \vee HWSW \in f \\
 & \ \vee SWHW \in f \wedge (a \text{ abhängig von } FPGA \\
 & \quad \vee a \text{ abhängig von } HWSW\ )\ )
\end{cases}
\tag{13.4}
$$

Der Host-PC soll genau dann die Fehler injizieren, wenn ein Umweltsignal manipuliert werden soll, das maximal von der Zeit und von weiteren Umweltsignalen abhängig ist oder wenn ein Steuersignal manipuliert werden soll, das maximal von der Zeit und von weiteren Steuersignale abhängig ist. Mit abhängig ist hier gemeint, dass das Aktivierungsmuster nur aus Umweltsignalen und aus Fehleraktivierungen besteht, die auch nur aus Umweltsignalen bestehen. Der Mikrocontroller ist der Injektionspunkt, wenn Umweltsignale und Steuersignale nicht mit dem Host-PC manipuliert werden sollen, der Zustand des Mikrocontrollers manipuliert werden soll oder wenn ein Software-zu-Hardware-Signal manipuliert werden soll, das nicht von dem Zustand des FPGAs oder von einem Hardware-zu-Software-Signal abhängt. Das FPGA ist der Injektionspunkt, wenn der Zustand des FPGAs oder ein Hardware-zu-Software-Signal manipuliert werden soll oder wenn ein Software-zu-Hardware-Signal manipuliert werden soll, das von dem Zustand des FPGAs oder von einem Hardware-zu-Software-Signal abhängt.

Es wird auf dem Mikrocontroller und dem FPGA injiziert, wenn mehrere unterschiedliche zu injizierende Signale manipuliert werden sollen, die einzeln auf dem Mikrocontroller oder FPGA injiziert würden. Ein Beispiel ist die „ gleichzeitige“ Manipulation des Zustandes von Mikrocontroller und FPGA.

Teil IV.

Entwurf und Implementierung

In diesem Abschnitt soll zunächst der Aufbau des Systems erläutert werden. Dieses besteht aus drei Teilsystemen, dem Simulator, der Satellitensteuerung und der Fehlerinjektion, wie in Abbildung 13.8 dargestellt.

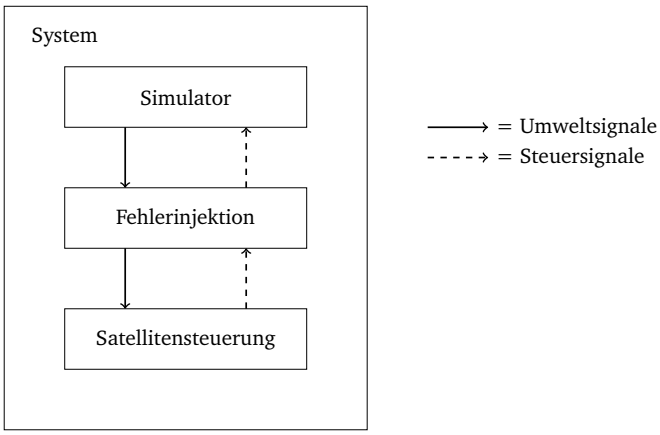


Abbildung 13.8.: Das System, bestehend aus drei Teilsystemen.

Der Spezifikation folgend, wird das Teilsystem Fehlerinjektion dabei verteilt angelegt; auf dem Host-PC befindet sich das User Interface (UI) zur Injektion von Fehlern, bzw. Fehlerexperimenten. Außerdem können an dieser Stelle durch die Fehlerinjektion die Umweltsignale und Steuerbefehle manipuliert, sowie Befehle an das Fehlerinjektionselement auf den ZedBoards geschickt werden, wo sich weitere Einheiten der Fehlerinjektion befinden, um Fehler auf der Hardware injizieren zu können. Abbildung 13.9 veranschaulicht diese Aufteilung.

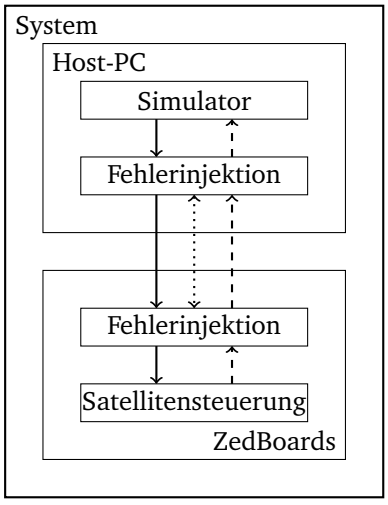


Abbildung 13.9.: Aufteilung der Fehlerinjektion im System.

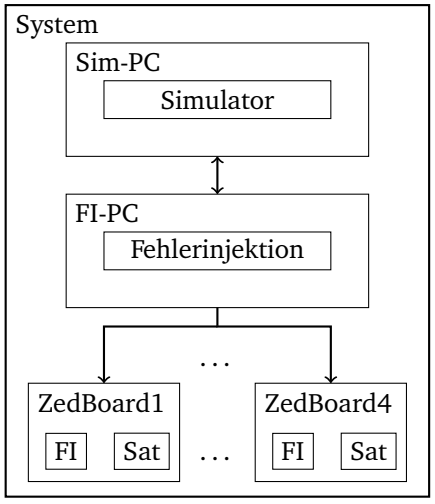


Abbildung 13.10.: Aufteilung des Systems auf die Hardware.

Durch die mangelnde Leistungsfähigkeit der Hardware, hier des Host-PC, hat sich die Projektgruppe dazu entschieden, die Teilsysteme bzw. Teilprojekte Simulator und Fehlerinjektion

voneinander zu trennen. Daraus ergibt sich die Aufteilung des Host-PC in den Simulations-PC (Sim-PC) und den Fehlerinjektions-PC (FI-PC). Abbildung 13.10 zeigt diese Aufteilung. Hier befindet sich das Teilsystem Simulator auf dem Sim-PC, während sich das Teilsystem Fehlerinjektion auf dem FI-PC befindet. Das Teilsystem Satellitensteuerung befindet sich, neben Teilen der Fehlerinjektion, auf den ZedBoards. Im Verlauf dieses Kapitels werden die Teilsysteme näher erläutert. Dazu wird jedes Teilsystem in Komponenten aufgeteilt, die wiederum aus Modulen bestehen.

Der folgende Abschnitt befasst sich mit dem Entwurf und der Implementierung der drei Teilsysteme. Zunächst wird der jeweilige Systementwurf vorgestellt, anschließend der Modulentwurf und abschließend die Implementierung. System- und Modulentwurf beinhalten außerdem Komponenten- und Modultests. Die Modultests werden für die Teilsysteme nur strukturell dargestellt. Die genaue Herangehensweise beim Testen kann dem Anhang entnommen werden.

14. Simulator

In diesem Kapitel werden System- und Modulentwurf, sowie die Implementierung des Teilsystems Simulator vorgestellt.

14.1. Systementwurf

In diesem Abschnitt wird der Systementwurf des Teilsystems Simulator dargestellt. Dies beinhaltet die Simulation auf dem Host-PC, die Verbindung zum ZedBoard sowie die Satellitensteuerung. Das Simulator-Plugin für den Prototyp I (Abschnitt B) weicht als Zwischenprodukt zum Endprodukt in seinen Komponenten ab. Die Komponente KSP ist die Simulationsumgebung, dargestellt durch das Spiel KSP. Als Basis für die Kommunikation wird das Plugin kRPC eingesetzt, welches eine Server- und Client-Seite besitzt. Der vorläufige Umfang ist in Abbildung 14.1 dargestellt. In der finalen Fassung kommuniziert das Simulator-Plugin ausschließlich über UDP und RAW Sockets. Dementsprechend fallen die Komponenten *Adapter*, *SystemC*, das IF *C++_IF* und der kRPC-Client weg. Die Anbindung von SystemC entfällt vollständig, weil diese nur für die Kommunikation mit der simulierten Satellitensteuerung benötigt wird. Das *UDP_IF* stellt die Bridge zwischen Ethernet und Feldbus dar.

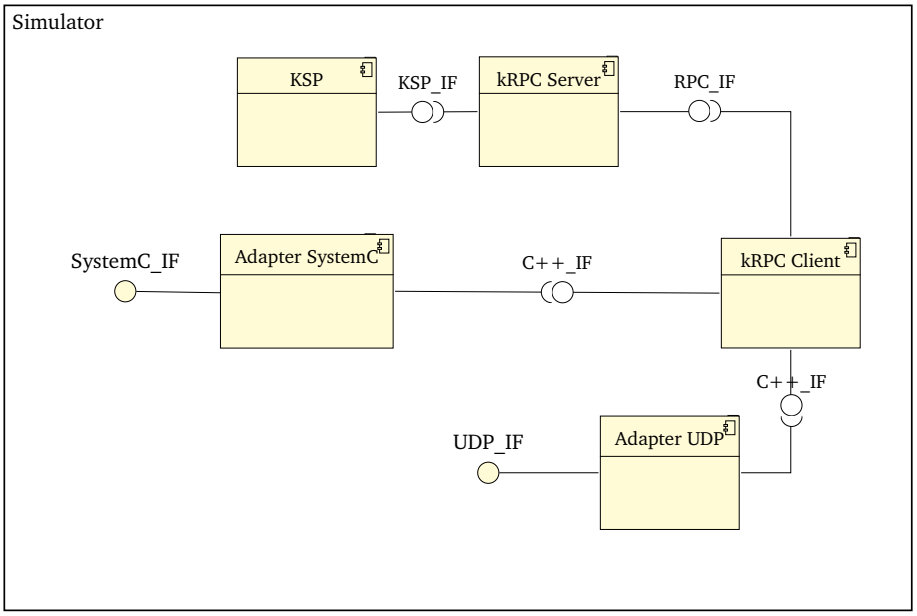


Abbildung 14.1.: Komponentendiagramm der Simulationskomponente auf dem Host-PC

Zur Kommunikation mit dem Simulator über den *kRPC-Server* wird der *kRPC-Client* verwendet, welcher sich über die Schnittstelle *RPC_IF* mit dem Host-PC auf dem Port 50.000 verbindet und mittels RPC-Calls kommuniziert. Diese Remoteprozeduraufrufe werden mit Protocol Buffer v3-Nachrichten realisiert. Bisher konnten die Protocol Buffer noch nicht unter Windows umgesetzt werden [17]. Aus diesem Grund wird für die Kommunikation derzeit ein GNU/Linux-System verwendet.

Zur Kommunikation mit dem ZedBoard werden die Schnittstelle *C++_IF* und die Komponente Adapter UDP verwendet, welche die Daten vom kRPC-Client umwandelt und über die Schnittstelle UDP_IF in Form von UDP- und RAW-Socket empfängt und weiterleitet. Des Weite-

ren ist über eine weitere *C++_IF*-Schnittstelle die Komponente *Adapter SystemC* angebunden. Diese verbindet die Simulation mit der Fehlerinjektion und der Satellitensteuerung. Über das *SystemC_IF* wird das Gesamtsystem der Satellitensteuerung angebunden, welches zu diesem Zeitpunkt in einer Simulationsumgebung verwendet wird und erst mit dem finalen System ebenfalls über die *UDP_IF* angebunden wird.

14.1.1. Komponententests

Das Teilsystem Simulator besteht aus fünf Komponenten. Es liegen keine direkten Komponententests vor, jedoch konnte im Betrieb der folgende Fehler festgestellt werden:

Kennzeichnung:	**Fault 001**
Bezeichnung:	**Segmentationfault im KSP Adapter**
Referenz:	Tabelle: 10.13
Beschreibung	Beim Anfragen der Mission Elapsed Time erfolgt ein Segmentation Fault.
Korrektes Verhalten:	Beim Anfragen der Mission Elapsed Time erfolgt kein Segmentation Fault.
Behoben:	**Beseitigt.**
Lösungsansatz:	Ein Fehler im Portmapping des KSP Adapters hat dazu geführt, dass die Anfrage nach der Mission Elapsed Time in den Kanal für das Anfragen von Orbitinformationen gelangt ist. Da der Handler für die Orbit Informationen ungeprüft versucht die Payload zu casten erfolgte ein Segmentation Fault. Das Portmapping wurde korrigiert und der Fehler somit behoben.

14.2. Modulentwurf

Im Modulentwurf werden die Komponenten aus Kapitel 11 in Module unterteilt und diese in Form von Klassen- und Sequenzdiagrammen erläutert.

14.2.1. Modulentwurf des logischen Modells

Im logischen Model wird das Simulator-PlugIn als Bindeglied zwischen der Simulation und der Satellitensteuerung verwendet. Es stellt die nötigen Funktionen bereit, welche es ermöglichen Befehle von der Satelitensteuerung an die Simulation zu senden. Im KSP Adapter werden die einzelnen Funktionen aller Adapter über First In, First Out-Queues mit der Simulation verbunden. Eingehende Pakete werden vom KSP Adapter entgegengenommen, interpretiert und je nach Inhalt des Paketes an die entsprechende First In, First Out weitergegeben. Über einen RPC-Call werden dann die Funktionen bei kRPC aufgerufen. Ist ein gültiger Befehl bei kRPC eingetroffen, wird von hier der entsprechende Befehl in der Simulation aufgerufen und ausgeführt. Anschließend wird dann, je nach Funktion, ein Objekt an den Sender zurückgeliefert. Dies kann in Form eines Richtungsvektors oder einer einzelnen Zahl geschehen. Am Adapter werden die Werte in einen Payload-Vektor verpackt und an den KSP Adapter gesendet. Dieser übergibt die erhaltenen Daten in Form eines Fieldbus Packets an die Satellitensteuerung (Abbildung D.31).

Am Beispiel des Engine Adapters soll der beschriebene Ablauf verdeutlicht werden. Dazu sollen die Triebwerke der Rakete von der Satelitensteuerung gestartet werden. Die Klasse En-

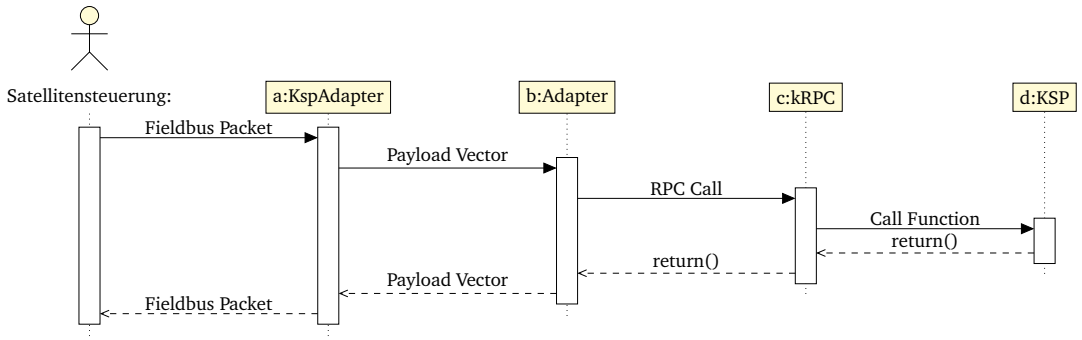


Abbildung 14.2.: Genereller Ablauf des KSP Adapters

gine Adapter hat die Möglichkeit die Triebwerke der Rakete zu starten, sowie Parameter über den aktuellen Zustand der Triebwerke abzufragen.

Müssen die Triebwerke der Rakete gestartet werden, wird ein entsprechender Befehl von der Satelitensteuerung an den KSP Adapter gesendet. Dieser entnimmt den Befehl aus dem Fieldbus Packet und leitet ihn in Form eines Payload Vektors an den Engine Adapter weiter. Hier wird dann der Funktionsaufruf ßet_active_handler()"mit dem Wert "trueän kRPC weitergeleitet. kRPC ruft daraufhin die Funktion ßet_active(true)äuf, was in der Simulation dazu führt, dass die Triebwerke der Rakete gestartet werden. Intern werden dann noch Rückgabewerte an kRPC und den Engine Adapter weitergegeben (Abbildung D.39).

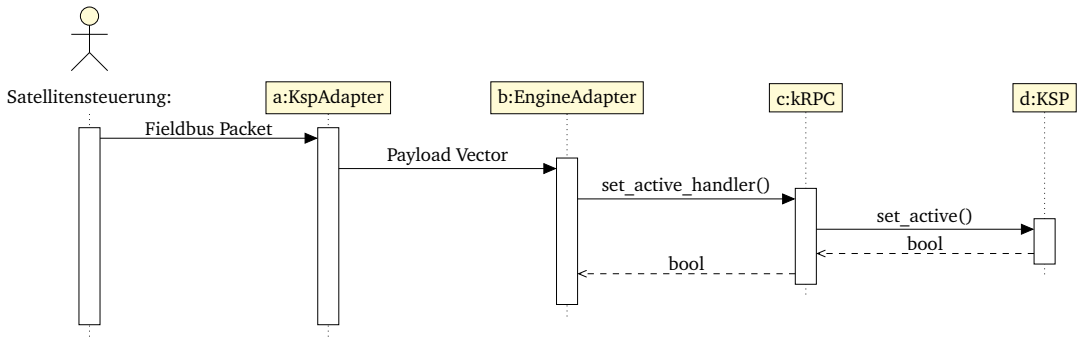


Abbildung 14.3.: Sequenzdiagramm

14.2.2. Modulentwurf des Simulator-PlugIns

Im folgenden Kapitel wird auf die Klassendiagramme und Sequenzdiagramme des Simulator-PlugIns eingegangen. Die grundlegende Struktur wird im Klassendiagramm in Abbildung 14.4 dargestellt.

Das Simulator-Plugin ist eine Erweiterung, welche die Steuerung einer Rakete in der Simulationsumgebung KSP ermöglicht. Mit Hilfe der KSP-API werden die Steuer- und Umgebungsvariablen über das PlugIn abgefragt und mittels UDP-Paketen an die Satellitensteuerung übertragen. Dieser Ablauf ist passiv und muss angestoßen werden, damit bespielsweise Statusinformationen übermittelt werden. Der Ablauf ist im Sequenzdiagramm in Abbildung 14.5 zu sehen.

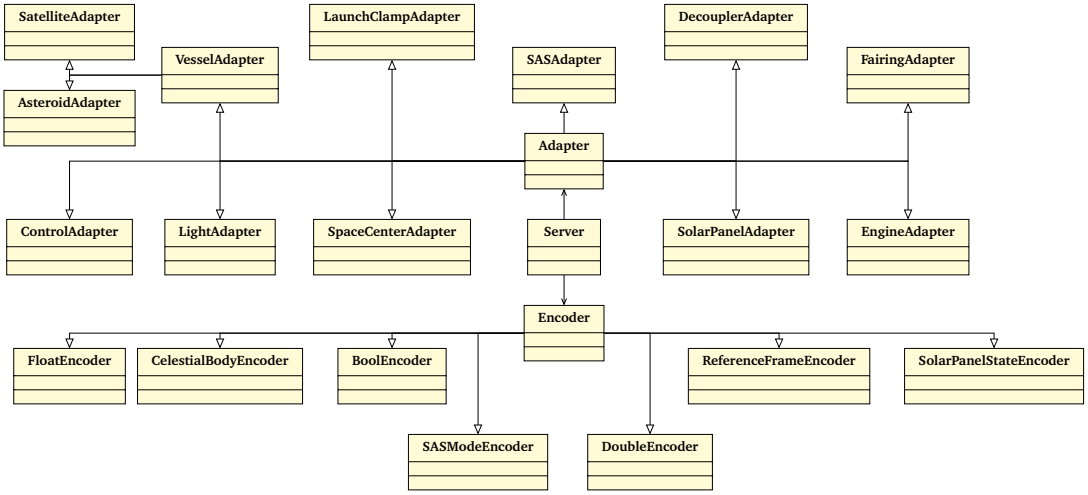


Abbildung 14.4.: Übersicht der Klassenbeziehungen der Teilgruppe Simulator

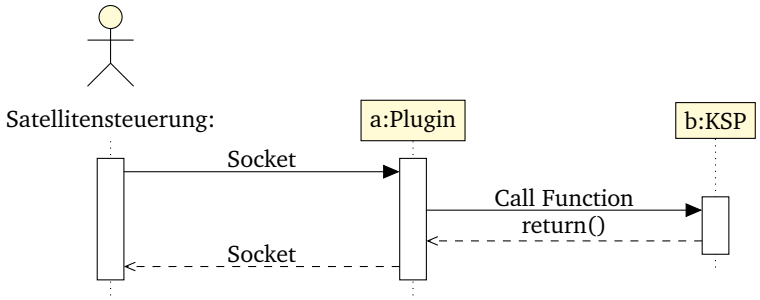


Abbildung 14.5.: PlugIn: Genereller Ablauf

Eingehende Pakete werden vom Server entgegengenommen, an den Encoder weitergegeben und dort deserialisiert. Das Ergebnis ist ein Datenobjekt in Form eines Fieldbuspackets, welches an den Adapter weitergeleitet wird. Dort wird es, je nach Inhalt, an die entsprechende Klasse weitergegeben. Diese sendet die jeweiligen Informationen, beziehungsweise holt diese über die KSP-API. Der genaue Ablauf ist im Sequenzdiagramm in Abbildung 14.6 zu sehen. Eine Ausnahme gilt für die Klasse *SpaceCenterAdapter*. Hier wird das Fieldbuspacket nicht direkt an die KSP-API gesendet, sondern bedient sich noch einzelner Funktionen aus dem kRPC-Plugin.

14.2.3. Modultests

Die Kernfunktion, die es zu für das Teilprojekt Simulator zu prüfen gilt ist, ob der Effekt, der durch eine gegebene Funktion erreicht werden soll, tatsächlich in KSP durchgesetzt wird. Als solches ist die Prüfung der Funktion nur zusammen mit KSP möglich.

Um im Allgemeinen die Funktion zu testen die von einem Adapter bereitgestellt wird, werden `Set` und `Get` Methoden gemeinsam getestet, falls vorhanden. Hierfür wird zunächst eine Eigenschaft über die `Set` Methode auf einen erwarteten Wert gesetzt und im Anschluss über die `Get` Methode geprüft, ob dieser Wert in KSP gesetzt wurde.

Durch das Setzen unterschiedlicher Werte kann sichergestellt werden, dass der ausgelesene Wert nicht zufällig mit dem erwarteten Wert übereinstimmt. Dadurch, dass durch die Adapter außerhalb von KSP keine Daten vorgehalten werden, kann garantiert werden, dass diese

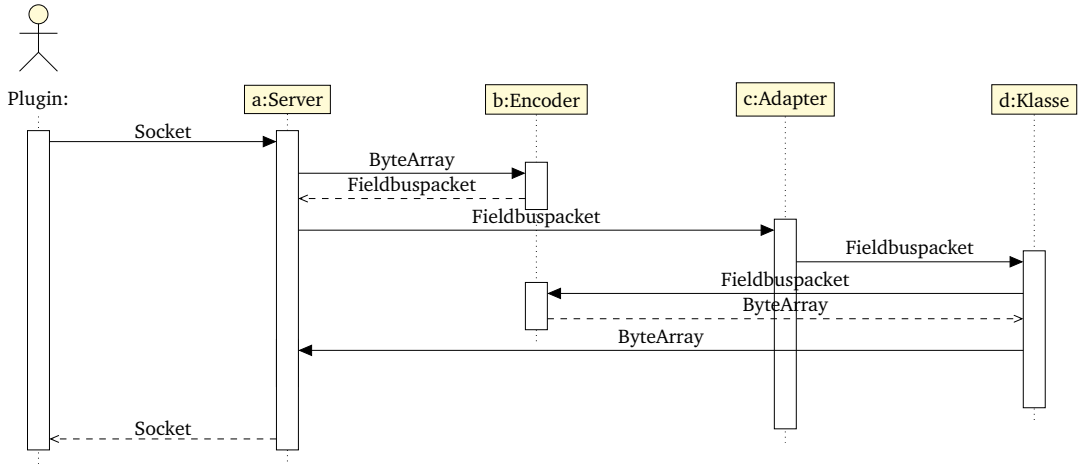


Abbildung 14.6.: PlugIn: Genereller Ablauf Plugin

erfolgreich in der Simulation gesetzt wurden, falls die erwarteten Daten ausgelesen werden konnten.

Zur vereinfachten Ausführung der Modultests wird das selbstentwickelte *SC_ASSERT* verwendet. Da es Fälle gibt, in denen keine `Get` Methode implementiert werden konnte, ist eine manuelle Prüfung in KSP nicht zu umgehen. Einer dieser Fälle ist die *Launch Clamp*, dessen API nur das Lösen der Klammer erlaubt, jedoch nicht das Abfragen, ob diese bereits gelöst wurde. Für das Testen des eigenen Plugins sowie des zugehörigen Clients wurden Ebenfalls die zum Test der Adapter angelegten Tests genutzt.

Modultests mittels SC_ASSERT

Zur Durchführung eines Modultests müssen in der Regel eine Reihe von Prüfungen von einer untersuchten Klasse bestanden werden. Die Erkenntnisse aus all diesen einzelnen Tests müssen aggregiert werden um ein Ergebnis für den vollständigen Test zu ermitteln. Um diese Aufgabe zu erleichtern wurde ein Makro mit dem Namen *SC_ASSERT* implementiert, welches zwei grundlegende Funktionen erfüllt:

- *Aggregation:* Es werden die gesammelten Prüfungen aus allen durchgeführten *SC_ASSERT* gesammelt und abhängig davon, ob alle Prüfungen bestanden wurden, wird von dem Prozess 0 zurückgegeben. Wurde eine der Prüfungen nicht bestanden wird -1 zurückgegeben.
- *Fehlermeldung:* Es wird eine Fehlermeldung ausgegeben in der die Zeile angegeben wird in dem sich das *SC_ASSERT* befindet, das fehlgeschlagen ist.

Implementiert wurde das *SC_ASSERT* durch ein Compiler Makro, in dem eine interne SystemC-Funktion aufgerufen wird. An die Funktion wird, neben einer zu übergebenden Bedingung, ebenfalls die aktuelle Zeile und der Name der Datei mitgegeben, an der das *SC_ASSERT* ausgeführt wurde. Die gesammelten Daten werden in der Variablen *test_state*, die sich in dem *sc_unit* Namespace befindet, aggregiert. Über die Funktion *result* im selben Namespace kann der passende Rückgabewert des Programms ermittelt werden.

Zur Durchführung der Modultests liegen zum jetzigen Zeitpunkt keine Ergebnisse vor. Daher müssen alle Tests als nicht durchgeführt bewertet werden. Für die folgenden Module wurden Testroutinen geschrieben, die dem Model im Ordner `model/test/unit/simulator` zu finden sind.

- base_adapter
- control_adapter
- decoupler_adapter
- engine_adapter
- fairing_adapter
- ksp_adapter
- launch_clamp_adapter
- light_adapter
- solar_panel
- space_center_adapter
- vessel_adapter

Es liegen außerdem Testbenches zu

- wait und
- fieldbus_packet

vor.

14.3. Implementierung

Dieser Abschnitt beschreibt die Implementierung des Teilsystems Simulator. Dabei werden das allgemeine Vorgehen näher erläutert und Probleme aufgezeigt, die während der Implementierung aufgetreten sind.

14.3.1. Zeitverständnis und Synchronisation

Integraler Bestandteil des Projektes ist es die Entworfene Steuerung mit Hilfe des Kerbal Space Program testen zu können. Da einzelne Durchläufe der Mission mehrere Jahre in Anspruch nehmen ist es notwendig die Zeit, soweit die Möglichkeit besteht, beschleunigen zu können. Diese Aufgabe setzt voraus, dass die Steuerung und die Simulation ihr Verständnis von Zeit synchronisieren und gemeinsam einen Sprung zu einem anderen Zeitpunkt absolvieren können. Da diese Zeitsprünge das Verhalten des Modells nicht beeinflussen darf, ist es notwendig, dass in dieser Zeit keinerlei Interaktion zwischen dem Modell der Satellitensteuerung und der Simulation stattfindet. Es ist daher notwendig zu ermitteln in welchen Zeitintervallen keine Interaktion stattfindet. Ein weiteres Problem stellen die unterschiedlichen Zeitmodelle dar, die genutzt werden.

Betrachtet man den vollständigen Satelliten als das zu modellierende System kann damit das erstellte Modell als eine Co-Simulation (siehe Kapitel 2.8) aufgefasst werden, bei der die Satellitensteuerung und der restliche Satellit in zwei unterschiedlichen Modellen dargestellt werden. Durch verschiedene Designentscheidungen können an dieser Stelle bereits einige Grundlegende Eigenschaften dieser Co-Simulation festgestellt werden:

- Durch das geschaffene Kommunikationsprotokoll zwischen Satellit und Steuerung existieren keine geteilten Daten die in den Modellen synchronisiert werden müssen.
- Da der Satellit sich vollkommen passiv verhält kann das identifizieren der Intervalle in denen die Modelle eigenständig arbeiten können direkt der Steuerung entnommen werden.

Abhängig davon, ob die Synchronisation zwischen dem SystemC basierten Modell der Satellitensteuerung oder den Zedboards der Satellitensteuerung und der Simulation vorgenommen werden soll, entstehen unterschiedliche Probleme. Im Folgenden werden die identifizierten Probleme und die implementierten Lösungen dieser dargestellt.

Zeitsynchronisation in SystemC

Zur Identifizierung der Zeitintervalle in denen die Modelle unabhängig arbeiten können wird der SystemC eigene Scheduler ausgenutzt (siehe Kapitel 2.7). Hierfür werden alle Aufrufe der *wait*-Funktion durch eine eigene Funktion mit dem gleichen Interface gekapselt. In dieser wird das eigentliche *wait* ausgeführt und im Anschluss die Synchronität zwischen den beiden Modellen wiederhergestellt.

Da das von KSP verwendete Zeitmodell (siehe Kapitel 2.3.4) nicht zulässt die Zeit zu pausieren muss hier die Annahme getroffen werden das die Intervalle in denen das SystemC-Modell voranschreitet immer größer sein müssen als die Zeit, die es zur Berechnung des jeweiligen Schrittes benötigt. Ist dies nicht der Fall ist es möglich, dass sich das Modell der Steuerung, im Verhältnis zu der Zeit in KSP, in der Vergangenheit befindet. Darüber hinaus kann dieses Zeitverhalten zu Inkonsistenzen führen. Da die Zeit, entgegen der Annahme von SystemC, kontinuierlich voranschreitet, werden Anfragen, die an KSP gemacht werden, nicht zum selben Zeitpunkt durchgeführt. Als Folge liefern wiederholte Anfragen nicht das selbe Ergebnis und auch Abfragen von physikalischen Größen, die von einander abhängig sind, können Inkonsistenzen aufweisen.

Zeitsynchronisation auf den Zedboards

Da auf dem Zedboard kein Scheduler wie unter SystemC zur Verfügung steht, der bereits die inaktiven Zeitabschnitte berechnet, ist dies für die Synchronisation der ZedBoards eigenständig zu erledigen. Da durch die angestrebte Redundanz mehrere ZedBoards und eine Fehlerinjektion verwendet werden, ist es darüber hinaus notwendig die Inaktivität aller beteiligten ZedBoards zu ermitteln. Im Folgenden werden alle ZedBoards und die Fehlerinjektion allgemein als *Teilnehmer* bezeichnet. Hierfür wurde in Anlehnung an die allgemeine Co-Simulation (siehe Kapitel 2.8) ein Protokoll entwickelt:

- *Registrierung:* Um die Inaktivität aller Teilnehmer zu identifizieren ist es notwendig zunächst zu wissen, welche Teilnehmer vorhanden sind. Hierfür muss jeder Teilnehmer

zunächst mitteilen, dass er für die Synchronisation berücksichtigt werden muss. Als Antwort erhält der Teilnehmer eine Bestätigung, zusammen mit der aktuellen Zeit des Kerbal Space Program, um eine Synchronisation der Zeit zu ermöglichen.

- *Wait:* Ist ein Teilnehmer Registriert kann er mit einem speziellen Paket mitteilen das er bis zu einem definierten Zeitpunkt inaktiv sein wird. Signalisiert ein Teilnehmer eine Inaktivität bis zu dem Zeitpunkt -1 wird dieser als wartend markiert, jedoch nicht geweckt. Als Antwort wird das Setzen des Zeitpunktes mit einem speziellen Paket quittiert in dem ebenfalls die aktuelle Zeit in KSP mitgegeben wird.
- *WakeUp:* Wurde der Zeitpunkt erreicht bis zu dem ein Teilnehmer seine Untätigkeit mitgeteilt hat, wird ein spezielles Wakeup-Signal mit der aktuellen KSP-Zeit gesendet. Dieses Paket wird so lange wiederholt übermittelt, bis der Teilnehmer sein erfolgreiches Erwachen mitteilt.
- *WakeUpReceived:* Mit diesem Paket signalisiert ein Teilnehmer, dass er das *WakeUp* erhalten hat und bereit ist, seine Aktivität fortzusetzen. Als Antwort erhält der Teilnehmer ein quittierendes Paket mit der aktuellen KSP-Zeit.
- *Interrupt:* Da es möglich ist, dass die Teilnehmer sich untereinander wecken, ist es über dieses Signal möglich, das eigene Warten abzubrechen. Es wird davon ausgegangen, dass dieses Signal nur gesendet wird, falls sich KSP in einem Zustand befindet, in dem es die Zeit nicht beschleunigt. Dies sollte der Fall sein, solange die Teilnehmer sich nicht innerhalb ihrer inaktiven Phase gegenseitig wecken. Als Antwort erhält der Teilnehmer ein quittierendes Paket mit der aktuellen Zeit.

Über dieses Protokoll ist es möglich die inaktiven Zeitintervalle zu ermitteln und die Zeit in KSP zu beschleunigen. Ebenso werden durch die in den Antworten enthaltene Zeitinformationen die Zeiten zwischen den Teilnehmern und dem Kerbal Space Program synchronisiert.

Mögliche Verbesserungen des Zeitverhaltens im Kerbal Space Program

Durch zusätzliche Plugins ist es möglich das Zeitverhalten von KSP zu manipulieren. Beispielsweise ist es über das Plugin *Time Control 2.0* möglich, die Physikalische Simulation zu verlangsamen [86]. Über die in dem Plugin verwendete Technik ist es ebenfalls möglich, die Zeit vollständig zu pausieren. Dies wurde beispielsweise in dem Plugin *Kerbal Alarm Clock* implementiert [81].

Unter der Annahme, dass diese Zeitkontrollfunktionen von außen zugreifbar gemacht werden können, ist es möglich, das Zeitverhalten des SystemC-Modells auf die Simulation zu übertragen. Über die verlangsamte Physiksimulation ist es möglich, die Auflösung der Simulation, auf Kosten der Ausführungszeit, zu erhöhen. Durch die zusätzliche Implementierung des Pausierens, könnte ebenfalls die Deltazeitsemantik der SystemC-Simulation auf den Simulator übertragen werden.

14.3.2. Serialisierung von Feldbuspaketen

Für die Kommunikation zwischen der Satellitensteuerung und dem Simulator wurden *Fieldbus_Packet*-Objekte definiert, die alle Informationen beinhalten, die für das Auslösen einer

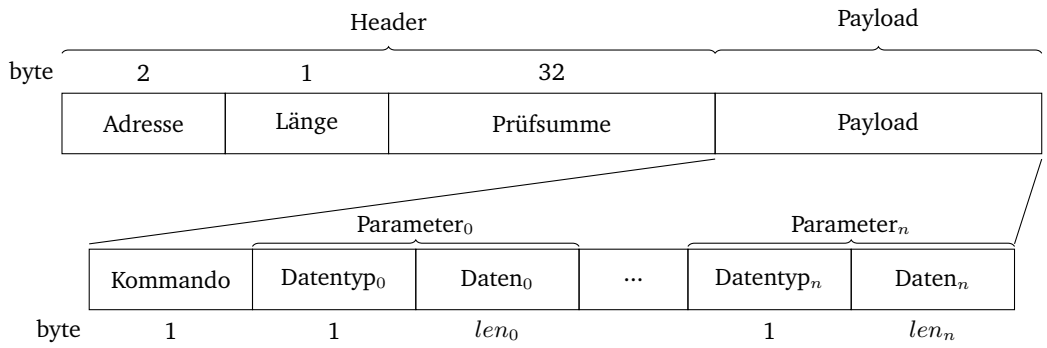


Abbildung 14.7.: Allgemeine Struktur eines serialisierten Fieldbus_Packet-Objekts.

Aktion benötigt werden oder alle Daten enthalten, die von der entsprechenden Seite angefragt werden. Während der Arbeit auf Basis des SystemC-Modells konnten diese Objekte direkt per Referenz an die verarbeitenden Adapter übergeben werden. Mit dem Aufbau eines eigenen Plugins ist dies jedoch nicht länger möglich. Stattdessen ist es nun notwendig, die Objekte in eine Serielle Byte-Form zu bringen, um diese über das Netzwerk an die verarbeitende Gegenstelle zu versenden. Dieser Prozess wurde bereits im Modulentwurf angedacht, jedoch wurde das exakte Protokoll nicht genauer spezifiziert. Dieses wurde im Rahmen der Implementierung entwickelt und wird im Folgenden genauer beschrieben.

Ein *Fieldbus_Packet* Objekt beinhaltet im Allgemeinen vier Informationen:

- Die Adresse des angesprochenen Busteilnehmers,
- einen eindeutigen Schlüssel, der das auszuführende Kommando spezifiziert,
- eine Prüfsumme, um den Transport des Pakets zu sichern und
- eine Payload, die aus einer undefinierten Anzahl von zu übertragenen Variablen unterschiedlicher Datentypen besteht.

Um eine möglichst einfache Erweiterbarkeit des Protokolls zu gewährleisten wurde bei dem Design darauf geachtet, dass die Pakete sich aus ihrer serialisierten Form direkt deserialisieren lassen und keine weiteren Metainformationen über die ausgelöste Aktion oder die adressierte Komponente notwendig sind. Ebenso wurde darauf geachtet, die Größe auf ein Minimum zu beschränken, um den entstehenden Netzwerkoverhead so gering wie möglich zu halten.

Struktur des serialisierten Objekts

Die Entwickelte Struktur lässt sich im Groben in zwei Bereiche unterteilen; angefangen mit dem *Header*, der allgemeine Informationen enthält und eine konstante Länge besitzt. Dem Header folgt ein Bereich mit variabler Länge, der für die zu übertragende Payload bestimmt ist. Eine Übersicht der Struktur des serialisierten Objekts und der Länge der einzelnen Komponenten kann der Abbildung 14.7 entnommen werden.

Der Header beinhaltet die Adresse des Objekts, das über den Feldbus erreicht werden soll; ein weiteres Byte, über das die Länge der folgenden Payload in Byte angegeben wird; sowie die Prüfsumme, die für die Transportsicherheit integriert wurde. Die Prüfsumme wird hierbei lediglich über die Payload gebildet.

Tabelle 14.1.: Eine Übersicht alle unterstützten Datentypen mit ihrer Datenlänge und dem Code, der den entsprechenden Datensatz markiert.

Datentyp	**Code**	**Datenlänge**
bool	0x01	1 Byte
float	0x10	4 Byte
double	0x11	8 Byte
Reference_Frame	0x02	1 Byte
Celestial_Body	0x03	1 Byte
Solar_Panel_State	0x04	1 Byte
SAS_Mode	0x05	1 Byte

Die Payload besteht aus mindestens einem Byte und maximal 255. Diese Limitierung entsteht durch die Wahl des einen Bytes zur Längencodierung im Header. Das erste Byte der Payload enthält immer den Wert des Kommandos aus dem *Fieldbus_Packet*-Objekt, das serialisiert wird. Abhängig von den im Paket hinterlegten Parameterobjekten, können weitere Bytes angehängt werden.

Serialisierung der übertragenen Datentypen

Durch den modellbasierten Ansatz konnte zu dem Zeitpunkt des Entwurfs des Protokolls bereits identifiziert werden, welche Datentypen von dem Protokoll unterstützt werden müssen. Eine tabellarische Aufführung aller unterstützten Datentypen kann der Tabelle 14.1 entnommen werden.

Da der Datentyp der mitgegebenen Parameter direkt aus dem Paket geschlossen werden kann, wurde für jeden Datentyp ein eindeutiger Code definiert (siehe Tabelle 14.1) der jedem serialisierten Parameter vorangestellt wird (siehe Abbildung 14.7). Auf Basis dieses Bytes wird geschlossen, wie viele Bytes der folgende Parameter beansprucht und damit der nächste Parameter beginnt.

Dieser Datenteil ist abhängig vom Datentyp unterschiedlich zu interpretieren. Handelt es sich um eines der Enums Reference_Frame, Celestial_Body, Solar_Panel_State oder SAS_Mode, wird in dem einen Byte das für die Darstellung genutzt wird, der Wert des Enums abgelegt. Im Falle eines *Bool* wird in dem Byte entweder der Wert 0xFF für *true* oder 0x00 für *false* abgelegt.

Bezüglich der Datentypen *float* sowie *double* werden die Daten im IEEE Format [43] abgelegt, wobei das hochwertigste Bit als erstes übertragen wird (Big-Endian).

15. Satellitensteuerung

In diesem Kapitel werden System- und Modulentwurf, sowie die Implementierung des Teilsystems Satellitensteuerung vorgestellt.

15.1. Systementwurf

In diesem Abschnitt wird der Systementwurf für das Teilsystem Satellitensteuerung dargestellt. Zunächst wird eine Übersicht über die Komponenten gegeben. Anschließend werden die Komponentenschnittstellen definiert und die Koordination von Feldbus und Mission erläutert.

15.1.1. Komponentenübersicht

Um die Aufgaben der Satellitensteuerung erfüllen zu können, wurden aus der Spezifikation die folgenden Komponenten abgeleitet, dargestellt in Abbildung 15.1.

- *MissionControl:* Kontrollinstanz, die die Phasenplanung (vgl. Abschnitt 12.2.3)vornimmt und den anderen Komponenten mitteilt, in welcher Phase der Mission sie sich gerade befinden. Sie koordiniert sich zu einer `leader`-Instanz. Über die Schnittstelle zum Operating System ist es der MissionControl möglich, Tasks neu zu starten.
- *ResourceManagement:* Diese Komponente ruft Umweltsignale von der FieldBus-Schnittstelle ab und sendet ggf. entsprechende Steuerbefehle an ebendiese Schnittstelle. Die Anpassungen aufgrund der Umweltsignale können phasenabhängig sein, weswegen sie zusätzlich eine Anbindung zur Phasenplanung besitzt. Des Weiteren benötigt sie ggf. Ressourceninformationen (Prozessor- und Arbeitsspeicherauslastung), die sie über eine Schnittstelle des Betriebssystems ausließt.
- *OrbitPlanning:* Diese Komponente setzt die Orbitplanung um und sendet die entsprechenden Maneuver an die MotionControl-Komponente. Dafür benötigt sie die aktuellen Umweltwerte und Lagewerte des Satelliten, welches sie über die FieldBus-Schnittstelle abruft.
- *MotionControl:* Diese Komponente setzt die Maneuver der Orbitplanung in Steuerbefehle für die Aktuatoren des Satelliten um und sendet diese an die FieldBus-Komponente.
- *FieldBus:* Die FeldBus-Komponente ruft Umweltsignale ab und kann Steuerbefehle senden. Sie koordiniert sich zur `leader`-Instanz im Kommunikationsnetzwerk.
- *LogServer:* Der Logserver nimmt Lognachrichten von allen Komponenten entgegen und speichert diese ab. Er ist vergleichbar mit einem Flugschreiber und dient der Fehlererkennung.

Die Komponente „Operating System“ in Abbildung 15.1 wird nicht entwickelt, sondern stellt der *MissionControl* nur Informationen über die Computerressourcen und eine Schnittstelle zur Kontrolle der Tasks und dem Ressourcenmanagement zur Verfügung.

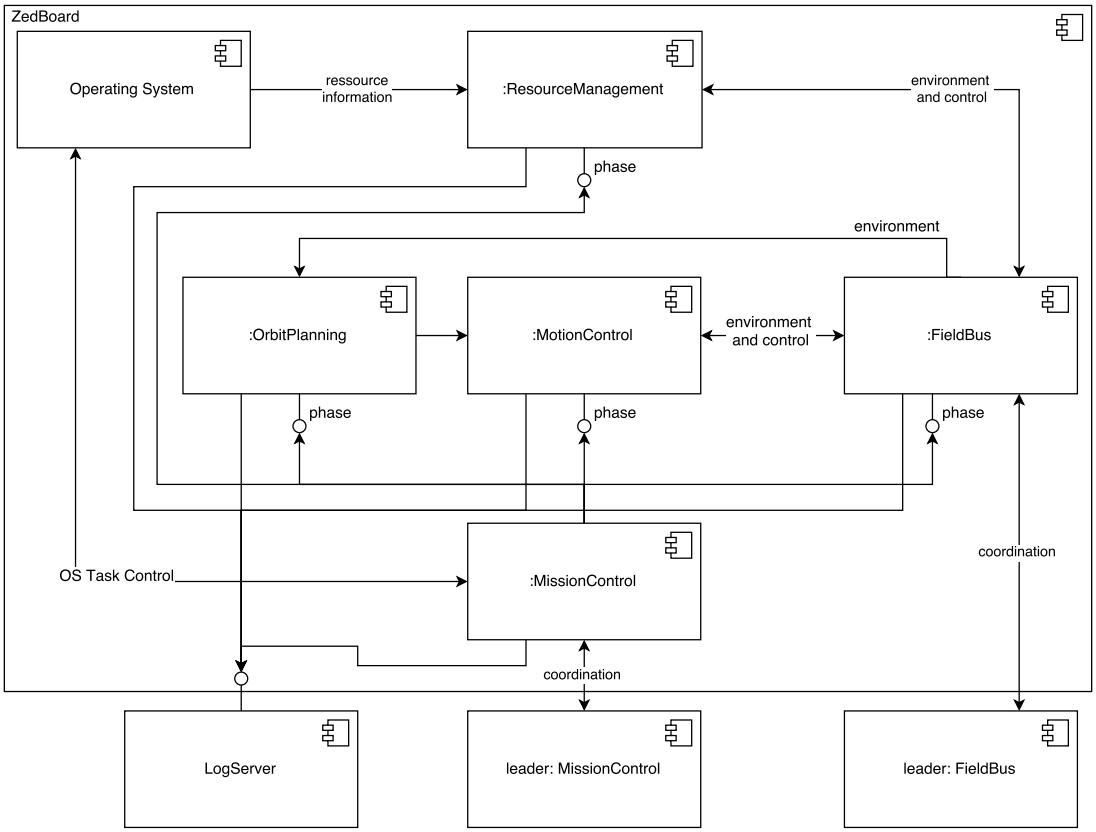


Abbildung 15.1.: Übersicht der Komponenten der Satellitensteuerung.

15.1.2. Allgemeine Komponentenschnittstelle

Jede Komponente (außer Logserver und MissionControl) sollen mindestens folgende Schnittstellen besitzen:

- *inout* state: Von außen muss es möglich sein, auf die internen Variablen jeder Komponente zuzugreifen. Dies geht über ein *state-interface*. Hier drin sind alle statusrelevanten Variablen zu kapseln.
- *in* phase: die aktuelle Phase. (erhalten von MissionControl)
- *out* log: Schnittstelle, um Lognachrichten zum Logserver zu senden.

15.1.3. Feldbuskoordination

In Abbildung 15.2 ist die Koordinierung der Feldbusschnittstelle mit anderen Feldbusschnittstellen dargestellt. Sie besitzt zwei Schnittstellen:

- *Election:* Über diese Schnittstelle findet die Wahl des Leaders statt.
- *BusCmdVoting:* Über diese Schnittstelle wird die Ausführung eines Steuerbefehls ausgehandelt.

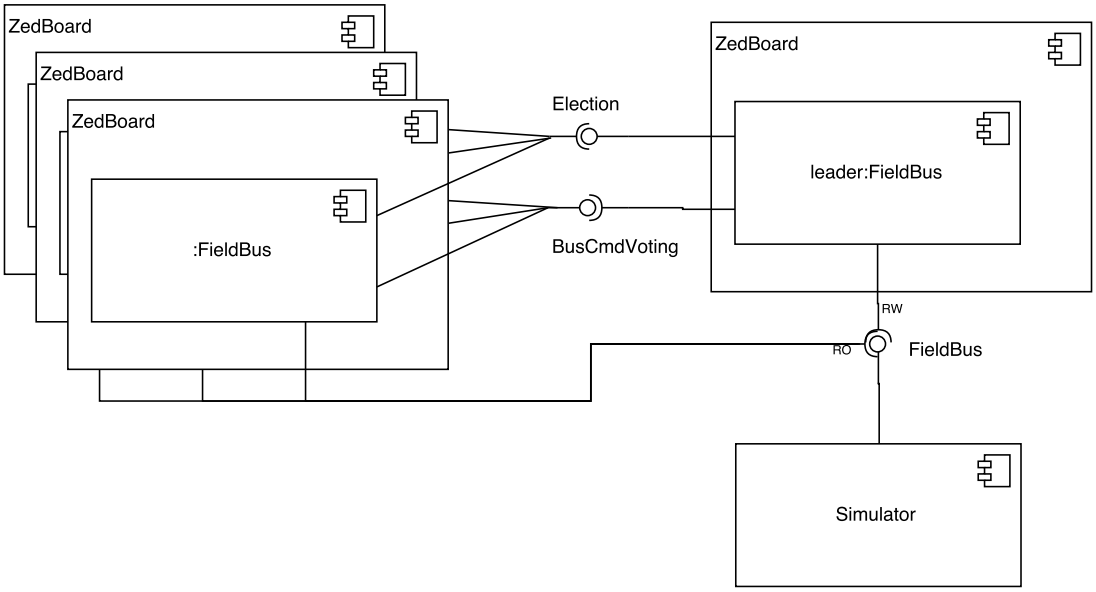


Abbildung 15.2.: Koordination der Feldbusschnittstelle.

- *FieldBus:* Über diese Schnittstelle wird von den ZedBoards lesend auf die Schnittstelle zugegriffen. Es können alle Befehle, die an die Simulation gehen und alle Rückmeldungen von ebendieser mitgelesen werden. Des Weiteren hat der Leader die Möglichkeit Befehle an die Simulation zu senden.

15.1.4. Missionskoordination

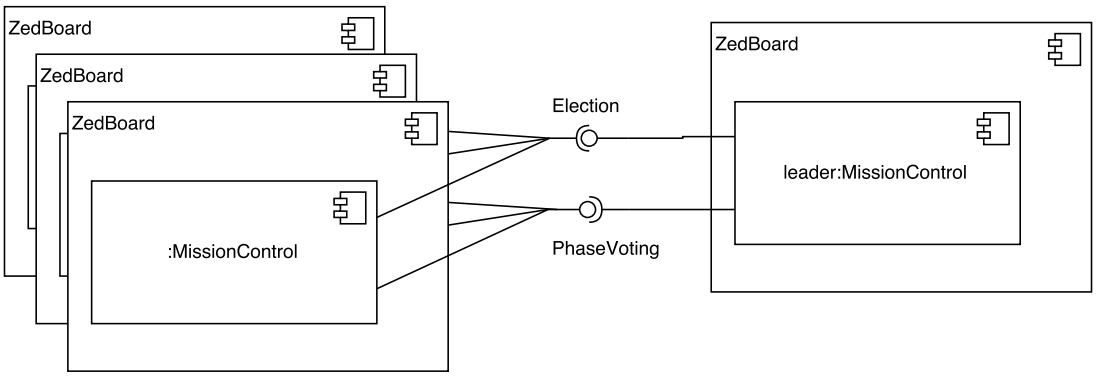


Abbildung 15.3.: Koordination der Missionsphasen

Die Koordination der einzelnen Phasen funktioniert analog zur Feldbuskoordination. Die beteiligten Komponenten wurden in Abbildung 15.3 dargestellt.

15.1.5. Komponententests

Die Testbenches der Komponenten finden sich im Ordner `model/test/unit/satellite_control`. Die folgenden Komponententests sind dort zu finden:

- field_bus

- log_server
- mission_control
- orbit_planning

Obwohl keine Ergebnisse der Komponententests vorliegen, konnten folgende Fehler identifiziert werden:

Kennzeichnung:	**Fault 002**
Bezeichnung:	**Payload Vector: Bufferüberlauf und -unterlauf**
Referenz:	
Beschreibung	Die `push_back`-Methoden überprüfen nicht, ob der Buffer bereits voll ist. Es wird dann einfach in die darüberliegenden Speicherbereiche geschrieben. Ein 80 Byte Buffer war aber bisher ausreichend. Die längste Payload besitzen die Orbit Informationen mit sieben doubles und einem Enum (insgesamt 57 Bytes). Dasselbe gilt für die `get`-Methode, die nicht überprüft, ob der Lesestand den Füllstand überschreiten kann.
Korrektes Verhalten:	Die `push_back`-Methoden und `get`-Methoden müssten die aktuelle Größe des Buffers kennen und dementsprechend bei einem Bufferüberlauf oder -unterlauf die eingehenden Daten verwerfen bzw. das Auslesen der Daten verhindern, wenn der Buffer überläuft.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Bei den aktuell benutzten Feldbuspaketen kann es niemals zu einem Überlauf kommen. Wird das System mit weiteren Paketen erweitert, die größere Datenmengen versenden, muss eventuell die Größe des Buffers erhöht werden.

Kennzeichnung:	**Fault 003**
Bezeichnung:	**Payload Vector: Typsicherheit beim auslesen nicht gewährleistet**
Referenz:	
Beschreibung	Die `get`-Methoden des Feldbuspaketes überprüfen nicht den gelesenen Datentyp. Die Driver müssen also wissen, wie die Pakete aufgebaut sind und dann die `get`-Methoden in einer passenden Reihenfolge aufrufen. Wird beispielsweise ein `bool` an erster Stelle in den Buffer geschrieben, aber dann ein `double` an dieser Stelle gelesen, werden insgesamt zwei Bytes geschrieben, während neun Bytes gelesen werden. Die sieben Bytes Differenz führen nun zu Fehlern in den nachfolgenden Werten.
Korrektes Verhalten:	Beim Auslesen der Werte aus dem Feldbuspaket, muss mittels der `get`-Methoden der beinhaltete Datentyp mit dem gewollten verglichen werden.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Jede Funktion, die Daten aus einem Feldbuspaket ausliest, muss die Reihenfolge der Daten genau kennen, dann können keine Fehler auftreten.

Kennzeichnung:	**Fault 004**
Bezeichnung:	**Fieldbus_Packet: Bufferüberlauf möglich bei Serialisierung**
Referenz:	
Beschreibung	Bei der Serialisierung von Feldbuspaketen kann das Paket die maximale Feldbuspaketlänge (`Fieldbus_Packet::MAX_PACK_SIZE`) übersteigen. Dies wird zu einem Problem, da der Buffer üblicherweise mit einer festen Größe von `Fieldbus_Packet::MAX_PACK_SIZE` initialisiert wird.
Korrektes Verhalten:	
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	In der aktuellen Version des Feldbuses können aufgrund der Paketlänge der benutzten Pakete keine Überläufe auftreten.

Kennzeichnung:	**Fault 005**
Bezeichnung:	**Fieldbus_Packet: Bufferüberlauf möglich bei Deserialisierung**
Referenz:	
Beschreibung	Es wird nicht überprüft, ob die Größe des empfangenen Paketes (`pack_sz`) größer als die maximale Feldbus-Paketlänge ist.
Korrektes Verhalten:	Beim Deserialisieren muss die Länge des Fieldbus Paketes im Blick behalten werden. Wird das Paket zu lang, muss dieses verworfen werden.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	In der aktuellen Version des Feldbuses werden keine Pakete gesendet, die einen Bufferüberlauf verursachen.

Kennzeichnung:	**Fault 006**
Bezeichnung:	**Der WaiterHeap in der Zeitimplementierung auf dem Zedboard besitzt manchmal zu viele Elemente**
Referenz:	
Beschreibung	Aufgrund einer unbekannten Abfolge von Registrierungen an der Zeitimplementierung kommt es manchmal dazu, dass zu viele Tasks in der Warteliste eingetragen sind.
Korrektes Verhalten:	Die Anzahl der Elemente in der Warteliste sollte nie die Anzahl der registrierten Tasks überschreiten.
Behoben:	Fehler wird voraussichtlich bis zur Endpräsentation behoben.
Lösungsansatz:	

Kennzeichnung:	**Fault 007**
Bezeichnung:	**Ausführung der Satellitensteuerung verklemmt sich manchmal**
Referenz:	
Beschreibung	Aufgrund einer unbekannten Abfolge von Warteeinträgen in die Zeitimplementierung kann es manchmal zu Verklemmungen in der Satellitensteuerung kommen.
Korrektes Verhalten:	Die Satellitensteuerung sollte komplett verklemmungsfrei sein.
Behoben:	Fehler wird voraussichtlich bis zur Endpräsentation behoben.
Lösungsansatz:	

15.2. Modulentwurf

Im Modulentwurf werden die Komponenten aus Kapitel 12 in Module unterteilt und diese in Form von Klassen- und Sequenzdiagrammen erläutert.

15.2.1. Klassendiagramm der Satellitensteuerung

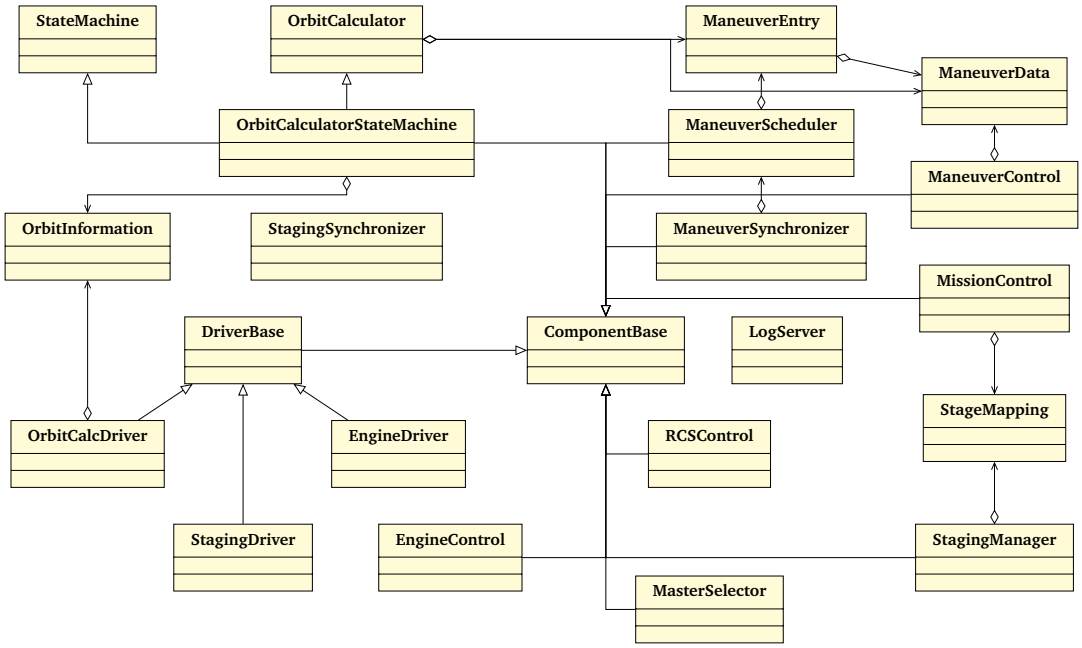


Abbildung 15.4.: Klassendiagramm Satellitensteuerung

Die Abbildung 15.4 zeigt das Klassendiagramm der Satellitensteuerung. Die grundlegenden Module der Satellitensteuerung sind abgeleitete Klassen der *Component_Base* Klasse. Die Information über die momentane Phase der Mission und eine Verbindung zum Logserver sind über die *Component_Base* Klasse abgedeckt. Diese Funktionen sind für die meisten Module elementar und sind daher in dieser Klasse bereits zusammengefasst.

Feldbustreiber

Die Feldbustreiber sind dafür verantwortlich, Anfragen der Satellitensteuerung in das spezifizierte Felsbusprotokoll umzusetzen, Daten aus ankommenden Feldbuspaketen zu extrahieren und an den entsprechenden Sender der Anfrage zurückzugeben.

Alle Feldbustreiber erben von der Basisklasse *Driver_ Base*. Diese stellt Ports und Funktionen zum Empfangen und Senden von Feldbuspaketen bereit. Alle eingehenden Pakete werden an alle Treiber zur Bearbeitung weitergereicht. Jeder Treiber ist dafür verantwortlich sich die Pakete, die mit ihn betreffenden Daten gefüllt sind, herauszusuchen. Für Anfragen gibt es jeweils einen Port mit dem Präfix *cmd*, über den die Anfrage gesendet werden kann und einen Port ohne Präfix, über den die Daten zurückgesendet werden. Steuerbefehle besitzen einen Port mit dem Präfix *set*.

Insgesamt gibt es drei Feldbustreiber mit den Namen *Engine_ Driver*, *Orbit_ Calc_ Driver* und *Staging_ Manager_ Driver*.

Der Treiber *Engine_ Driver* stellt Funktion zum Ansteuern des Antriebssystems bereit. Hierzu gehören:

- Abfragen des aktuellen Antriebwertes
- Setzen des Antriebwertes
- Abfragen der aktuellen Geschwindigkeit
- Abfragen der aktuellen Masse des Satelliten
- Einstellen der Roll, Nick und Gierwinkel
- Einstellen der aktuellen Reglerparameter des Autopiloten
- Abfragen der aktuellen Roll, Nick und Gierwinkel
- Aktivieren und deaktivieren des Autopiloten
- Ansteuern der RCS-Antriebe
- Abfragen ob ein Antrieb noch über Treibstoff verfügt
- Einstellen der Zielrichtung des Autopiloten
- Einstellen des Bezugssystems
- Abfragen des Fehlerwinkels zur Zielrichtung
- Einstellen des SAS

Der Treiber *Orbit_ Calc_ Driver* stellt Funktionen, die in der Orbitkalkulation benötigt werden, zur Verfügung. Hierzu gehören:

- Abfragen des Orbits des Satelliten, Zielasteroiden und Kerbin
- Abfragen der Distanz zum Ziel
- Abfragen des aktuellen Bezugssystems des Satelliten
- Abfragen der Position des Asteroiden
- Abfragen des Geschwindigkeitsvektors des Asteroiden
- Umrechnung zwischen zwei Bezugssystemen
- Abfragen der Geschwindigkeit des Asteroiden
- Abfragen der Höhe des Satelliten

Der Treiber *Staging_ Manager_ Driver* stellt Funktionen die für das Staging benötigt werden zur Verfügung:

- Lösen der Halteklammern am Start
- Ausfahren und Einklappen der Solarmodule

- Starten eines Antriebs
- Abfeuern eines Entkopplers
- Absprengen einer Verkleidung

Zu allen drei Treibern befindet sich die Testbench im Ordner `model/test/unit/satellite_control/field_bus`. Es liegen keine Ergebnisse für den Modultest vor.

MissionControl

Das Modul *Mission_ Control* ist für die Koordination der Missionsausführung zwischen den anderen Modulen der Satellitensteuerung verantwortlich. Zu den Aufgaben des Moduls gehört es, der Orbitkalkulation und dem Antriebssystem Signale zum Starten der einzelnen Aufgaben, wie Start, Orbitanpassung und das Andocken an den Asteroiden, zu geben. Des Weiteren gibt das Modul an, zu welcher Stufe die Rakete wechseln soll. Hierfür werden an bestimmten Stellen der Ausführung Teile der Rakete abgesprengt oder es wird zur nächsten Stufe gewechselt, wenn die aktuelle über keinen Treibstoff mehr verfügt. Zuletzt ist das Modul dafür verantwortlich, anderen Modulen die aktuelle Phase der Missionsausführung anzugeben, wodurch diese auf das Bezugssystem der Rakete schließen können.

Die Testbench befindet sich in `model/test/unit/satellite_control/mission_control`. Es liegen keine Ergebnisse für den Modultest vor.

StagingManager

Das Modul *Staging_ Manager* ist für das Staging verantwortlich. Es verfügt über eine vollständige Abbildung zwischen den Stufen der Rakete und den dazugehörigen Antrieben und Entkopplern, welche in der Klasse *Stage_ Mapping* gespeichert sind. Über das eingehende Signal *go_ to _ stage* kann angegeben werden, zu welcher Stufe gewechselt werden soll. Das Modul geht beim Wechseln dieses Signals die einzelnen Stufen bis zum Ziel durch und aktiviert alle Antriebe, beziehungsweise sprengt die Entkoppler ab.

Die Testbench befindet sich in `model/test/unit/satellite_control/staging_manager`. Es liegen keine Ergebnisse für den Modultest vor.

ManeuverControl

Das Modul *Maneuver_Control* ist abhängig von der Missionsphase für unterschiedliche Funktionen zuständig. Während des Starts der Rakete bis zum Erreichen der gewünschten Apoapsis um den Planeten Kerbin, werden sowohl Schubkraft und Richtung der Rakete berechnet und an das *Engine_Control* Modul und *RCS_Control* Modul weitergeleitet. Solange das *Orbit_Calculator* Modul arbeitet, werden die Manöver vom *Maneuver_Scheduler* angenommen und nach Orientierungsmanövern und Beschleunigungsmanövern unterschieden. Die Informationen werden dann entweder an das *RCS_Control* Modul oder *Engine_Control* Modul weitergeleitet. Sobald der *Orbit_Calculator* seinen Endzustand erreicht hat, beginnt das Dockingmanöver mit dem Asteroiden. Das *Manuver_Control* Modul überprüft während dieser Phase die Entfernung und den Geschwindigkeitsvektor relativ zum Asteroiden und leitet gegebenenfalls Korrekturmanöver ein, um den Satelliten am Asteroiden andocken zu lassen.

Die Testbench befindet sich in `model/test/unit/satellite_control/propulsion_system`. Es liegen keine Ergebnisse für den Modultest vor.

RCSControl

Das Modul *RCS_Control* beinhaltet die Orientierung des Satelliten. Die entgegengenommen Stellwerte vom *Maneuver_Control* Modul werden an den Autopiloten von kRPC weitergegeben. Über die Rückgabe des Winkelfehlers wird entschieden wann der Satellit die gewünschte Ausrichtung erreicht hat.

Die Testbench befindet sich in `model/test/unit/satellite_control/propulsion_system`. Es liegen keine Ergebnisse für den Modultest vor.

EngineControl

Das Modul *Engine_Control* ist für die Regelung des Satellitenantriebs zuständig. Während das *Orbit_Calculator* Modul aktiv ist, werden ΔV Werte an das Modul gesendet, die in eine Brenndauer für die Antriebe umgerechnet werden. Somit kann die erste Hälfte des Beschleunigungsmanövers exakt vor dem berechneten Zeitpunkt für die Beschleunigung durchgeführt werden. Der *Orbit_Calculator* geht von einer Geschwindigkeitsänderung in einem Zeitpunkt aus. Um die dadurch entstehenden Abweichungen zu minimieren, wird die beschriebene Anpassung vorgenommen. Während das *Maneuver_Control* Modul den Satelliten am Asteroiden andocken lassen will, wird in der *Engine_Control* die Geschwindigkeit des Satelliten relativ zum Asteroiden geregelt.

Die Testbench befindet sich in `model/test/unit/satellite_control/propulsion_system`. Es liegen keine Ergebnisse für den Modultest vor.

MasterSelector

Im *Master_Selector* Modul wird ausgehandelt, welche Instanz als Master arbeitet. Dies geschieht über die ID der jeweiligen Instanz. Die höchste ID gewinnt die Wahl und wird zur Master Instanz. Diese ID wird daraufhin an die weiteren Instanzen übermittelt, damit der Master bekannt ist.

Die Testbench befindet sich in `model/test/unit/satellite_control/faulttolerance`. Es liegen keine Ergebnisse für den Modultest vor.

Synchronizer

Das *Maneuver_Synchronizer* Modul sendet jedes von der Instanz auszuführende Manöver an alle anderen Instanzen, genauso wie es von jeder Instanz die jeweiligen Manöver empfängt. Die ausgewählte Master Instanz überprüft daraufhin, ob es für das eigene Manöver eine Mehrheit gibt, um es auszuführen. Falls es zu keiner Mehrheit kommt, wird das Manöver verworfen. Das *Staging_Synchronizer* Modul funktioniert nach dem gleichem Prinzip. Es überprüft, ob ein Staging ausgeführt werden soll oder verworfen wird.

Die Testbench für den *Staging_Synchronizer* befindet sich in `model/test/unit/satellite_control/faulttolera`
Es liegen keine Ergebnisse für den Modultest vor.

Orbitkalkulation

Die Klasse *Orbit_Calculator* beinhaltet die Berechnungen der möglichen Manöver während der Phasen der Orbitkalkulation. Zusammen mit der *State_Machine*-Klasse entsteht so das *Orbit_Calculator_State_Machine*-Modul. Sobald die Startphase abgeschlossen ist, werden über die *State_Machine*-Klasse die möglichen Zustände der Orbitkalkulation abgebildet. Sobald der Satellit und der Asteroid eine bestimmte Entfernung voneinander unterschreiten, wechselt das *Orbit_Calculator_State_Machine*-Modul in den Endzustand. Das *Mission_Control*-Modul wird benachrichtigt und das Docking an den Asteroiden kann gestartet werden.

- *Austrittsmanöver:* Beim Austrittsmanöver wird zusätzlich zu dem in der Spezifikation beschriebenen Manöver (Abschnitt 12.4.8) ein günstiger Zeitpunkt ausgewählt. Im Austrittsmanöver wird die Zeit berechnet, bis der Asteroid an der Apoapse ist. Die Differenz von der Zeit, bis der Asteroid an dessen Apoapse ist, zu der Zeit, die die Flugbahn des Satelliten braucht, wird durch ein Newton-Verfahren zu Null. Also ist der Satellit gleichzeitig mit dem Asteroiden an dessen Apoapse. Hierbei gibt es einen schnellsten Weg des Satelliten, nämlich der, dass nach dem Austrittsmanöver die große Halbachse des Satellitenorbits gleich der großen Halbachse des Kerbin Orbits ist. Da Kerbin in dessen Flugrichtung um Kerbol verlassen wird, ist dieser der schnellste Weg, da der Satellit nach Verlassen nicht einen kleineren Orbit als Kerbin haben kann. Zusammenfassend wird vor dem Newton-Verfahren die Differenz zwischen schnellstem Weg des Satelliten und Ankunft des Asteroiden an dessen Apoapse berechnet. Ist diese Differenz größer als Kerbins Umlaufzeit, wird solange in einem Orbit um Kerbin gewartet, bis die Differenz nicht mehr größer ist als Kerbins Umlaufzeit.

- *Korrekturmanöver:* Beim Modulentwurf mussten, zusätzlich zu den in den Spezifikationen beschriebenen Manövern, Manöver zur Korrektur der Flugbahn implementiert werden. Diese sind notwendig, da die in den Spezifikationen dargestellten Manöver davon ausgegangen sind, dass diese genauso wie berechnet ausgeführt werden. Das ist jedoch nicht möglich, da eine sofortige Geschwindigkeitsänderung physikalisch nicht möglich ist. Der Satellit muss über einen Zeitraum beschleunigen und kann die Geschwindigkeit nicht an einem Punkt ändern. Dadurch werden die Manöver ungenau. Eine weitere Ursache für Ungenauigkeiten ist die nicht hundertprozentig richtige Ausrichtung des Satelliten während eines Manövers. Insbesondere haben Manöver zur Anpassung eines bestimmten Bahnelementes auch Auswirkungen auf andere Bahnelemente. Diese Ungenauigkeiten haben zur Folge, dass wiederholt Manöver zur Korrektur notwendig sind, um nahe genug an den Asteroiden heranzukommen. Die Korrekturmanöver beinhalten Manöver zur Korrektur der Inklination, des aufsteigenden Knotens und der Apoapse des Satellitenorbits. Diese sind innerhalb der Zustände der *Orbit_Calculator_State_Machine* implementiert. Dabei wird berechnet, welche Korrekturmanöver notwendig sind und welches als nächstes ausgeführt werden kann. Dieses wird dann ausgeführt. Es wird immer nur ein Manöver geplant, da Manöver aufgrund der Ungenauigkeiten nicht genau im Voraus berechnet werden können.

  Zur Notwendigkeit der Korrekturmanöver folgt hier ein Rechenbeispiel. Als Beispiel wird Kerbins Orbit verwendet, da der Satellitenorbit nach Verlassen Kerbins in derselben Größenordnung ist. Der Radius des Kerbinorbits ist ungefähr $1.4^{10}m$. Also ist der Umfang der

Kerbinorbits ungefähr $10^{10}m$. Eine Änderung des aufsteigenden Knotens um $0.0001Radian$ hat demnach eine Positionsverschiebung am aufsteigenden Knoten von $10^7km * \frac{1}{2*\pi} * 0.0001 = 159.16km$ zur Folge. Um eine Annäherung des Asteroiden auf wenige Hundert Kilometer zu erreichen, müssen demnach diese Bahnelemente bis auf diese Größenordnung genau angepasst werden. Analog ist die Rechnung für Änderung der Inklination.

- *Orbitkalkulation State Machine:* In Abbildung 15.5 ist die State Machine der Orbitkalkulation abgebildet. Neben dem Anfangszustand `Apoapsis Kerbin` gibt es auch die nicht in der Abbildung dargestellte Möglichkeit ein Reset durchzuführen, wodurch die State Machine in den zum Satellienorbit passenden Zustand geht. Die ersten beiden Zustände `Apoapsis Kerbin` und `Periapsis Kerbin` sind nur dazu da, den Satellitenorbit groß genug zu machen, damit in der Simulation auf schnellster Stufe vorgespult werden kann. Der dritte Zustand ist dazu da, dem Satelliten die richtige Inklination zum Verlassen von Kerbin zu geben. Im Zustand `Leave Kerbin` wird das Austrittsmanöver geplant. Nach dem Austritt aus Kerbins Einflusssphäre ist der Satellit auf einem Orbit um Kerbol. Ab diesem Zustand werden zusätzlich zu den spezifizierten Manövern Korrekturmanöver durchgeführt. Nachdem der Satellitenorbit kreisförmig gemacht wurde, um im nächsten Manöver mit dem Anpassen der Apoapse des Satelliten auf die Apoapse des Asteroiden auch das Argument der Periapse anzupassen, wird in den Zustand `Apoapsis Kerbol` gewechselt. Hier wird das Manöver zum Anpassen der Apoapse durchgeführt. Der vorletzte Zustand ist nun `Phasing Kerbol`, in dem das durchgeführte Manöver das Phasing ist. Hier werden Korrekturmanöver nur noch durchgeführt, wenn die Bahnelemente Inklination und aufsteigender Knoten eine zu geringe Genauigkeit haben. Sobald der Abstand zum Asteroiden unter $500km$ ist und der Satellit auch an der Apoapse ist, wird in den Endzustand gewechselt. Darauf folgen das Rendezvous und Docking mit dem Asteroiden.

- *Phasingmanöver:* Zusätzlich zu dem spezifizierten Phasingmanöver muss beim Phasing der Punkt auf dem Orbit, also die wahre Anomalie, auf dem das Phasingmanöver durchgeführt wurde, gespeichert werden. Wenn dieser nicht gespeichert wird, verwendet die nächste Berechnung des Phasingmanövers einen anderen Bezugspunkt, durch die im vorherigen Abschnitt erklärten Ungenauigkeiten. Dies führt dazu, dass die nächste Berechnung eine große zeitliche Differenz zwischen dem Ankommen des Satelliten an der Apoapse und dem Ankommen des Asteroiden an der Apoapse feststellt, die jedoch nicht mehr vorhanden ist. Dadurch macht das darauffolgende Phasingmanöver die erste Annäherung wieder zunichte. Deshalb wird direkt nach dem Phasingmanöver mit den neuen Bahnelementen die wahre Anomalie berechnet, damit an diesem auch das nächste Phasing stattfindet.

Die Testbench für den *Staging_Synchronizer* befindet sich in `model/test/unit/satellite_control/orbit_calc.`
Es liegen keine Ergebnisse für den Modultest vor.

15.2.2. Modulentwurf des logischen Modells

Anhand des Klassendiagramms 15.6 wird exemplarisch der Aufbau der Module der Satellitensteuerung aufgezeigt und erläutert. Die Kommunikation zwischen den einzelnen Modulen

wird bei einer 1:1 Verbindung über First In, First Outs realisiert. Handelt es sich bei der Kommunikation um 1:n Verbindungen, werden Signale verwendet.

Über die Funktion „set_current_execution()“ werden die ausführenden Module für die einzelnen Abschnitte der Mission eingeschaltet. Somit wird zum Beispiel über den „activate_start“ FIFO, die Startsequenz im „Maneuver Control“ gestartet. Nachdem der FIFO „start_done“ meldet, dass die Startsequenz abgeschlossen ist, kann über „activate_orbit_calc“ die Orbitkalkulation gestartet werden. Durch dieses Vorgehen wird verhindert, dass die einzelnen Module sich gegenseitig behindern und gleichzeitig eine Regelung des Satelliten vornehmen wollen. Die Kommunikation zum Simulator-Plugin wird ebenso über First In, First Outs realisiert. Am Beispiel der „Mission Control“ ist entscheidend zu wissen, ob ein Antrieb über Treibstoff verfügt. Dies geschieht indem in den First In, First Out „cmd_has_fuel“ die Adresse des zu überprüfenden Antriebs geschrieben wird. Über einen Rückkanal „has_fuel“ kann dann ein Bool ausgewertet werden.

Das Signal „acceleration_running“ hingegen kann von mehreren Modulen gesetzt werden. Es signalisiert der „Mission Control“, dass die Antriebe eingeschaltet sind. Nur dann werden die Antriebe auf zugänglichen Treibstoff überprüft.

Das Sequenzdiagramm 15.7 stellt den Kommunikationsfluss während der laufenden Orbitkalkulation dar. Der „OrbitCalulator“ erhält über die KSP-Adapter Informationen über die Umgebung und berechnet daraus ein Manöver, das zu einem bestimmten Punkt ausgeführt werden soll. Dieses Manöver wird mit Hilfe der „Maneuver_Entry“ Klasse an den „Maneuver Scheduler“ geschickt. Dieses Modul verwaltet eine Liste der anstehenden Manöver. Sobald der Zeitpunkt eines Manövers erreicht ist, wird mit Hilfe der „Maneuver_Data“ Klasse das Manöver an „Maneuver Control“ weitgeleitet. Beim Sequenzdiagramm 15.7 handelt es sich um ein Beschleunigungsmanöver und daher wird ein Δ_V Wert an die „Engine Control“ weitergeleitet, in der das Manöver ausgeführt wird. Über einen Rückkanal wird das abgeschlossene Manöver an die Orbitkalkulation zurückgemeldet.

15.2.3. Modultests

Bei den Modultests der Satellitensteuerung wird zwischen den Tests für Software (im SystemC-Modell) und der Hardware unterschieden. Für beide Teile gibt es beim Testen verschiedene Vorgehen, welche im Folgenden beschrieben werden.

Modultest der Software

Zur Durchführung von Modultests des SystemC Modells wurde für jedes Modul eine Testbench erstellt. Die Testbench verwendet ebenfalls den SystemC Simulator, weshalb für jede Testbench ein eigenes Programm kompiliert werden muss.

In jeder Testbench wird jeweils nur ein einzelnes SystemC Modul instanziert. Für jeden Port des Moduls wird eine FIFO mit dem selben Namen erstellt, welche anschließend mit den zugehörigen Ports verbunden wird. Jede Testbench besitzt genau einen Thread, namens `behav`, welcher nacheinander Testmethoden aufruft. In jeder Testmethode werden die FIFOs mit den entsprechenden Stimulidaten zum Auslösen des Tests befüllt und anschließend wird der Zustand des Moduls über Signale bzw. FIFOs festgestellt und mit dem Sollzustand verglichen. Mit

den Testmethoden wird jeweils versucht eine möglichst hohe Abdeckung des Moduls zu erreichen. Zum Überprüfen des Status werden die in Abschnitt 14.2.3 beschriebenen `SC_ASSERT` Methoden verwendet. Nach Abschluss der SystemC Simulation wird festgestellt, ob ein Fehler in den Tests aufgetreten ist und dementsprechend der Rückgabewert der Testbench festgesetzt. Es wird ebenfalls der erzeugte Log in der Konsole ausgegeben.

Um die Ausführung der Testbenches zu vereinfachen, wurden die Testbenches in CMake bzw. CTest eingebunden. Dies ermöglicht es, alle Testbenches auf einmal zu kompilieren und anschließend hintereinander ausführen zu lassen und einen Bericht über den Status der Testbenches zu generieren. Hier wird für jede Testbench angegeben, ob diese erfolgreich ausgeführt wurde, die Ausführungszeit und die Gesamtanzahl der erfolgreichen Tests und fehlerhaften Tests. Ein Test gilt als erfolgreich ausgeführt, wenn das Programm einen Rückgabewert von 0 hat, ansonsten ist der Test fehlgeschlagen. Eine Übersicht mit allen erzeugten Konsolenausgaben wird in einer Textdatei in dem `build`-Ordner erzeugt. Die Zusammenfassung der Testausführung gibt nur den Gesamtzustand der Testbenchausführungen wieder. Für genauere Informationen muss das generierte Logfile betrachtet werden.

Modultest der Hardware

Die beiden Funktionen auf der Hardware besitzen jeweils eine eigene Testbench. Die Testbenches sind ebenso wie die Module in C++ geschrieben und können in zwei Varianten verwendet werden:

C-Simulation Bei der C-Simulation werden sowohl das Modul als auch die Testbench kompiliert und die Testbench ausgeführt. So lässt sich zeitsparend die Funktionalität des Moduls testen.

C/RTL-Cosimulation Bei der C/RTL-Cosimulation wird weiterhin die kompilierte Testbench verwendet, jedoch wird das Modul synthetisiert und als Hardwarekomponente simuliert. So kann sichergestellt werden, dass das Modul, eine entsprechende Testbench vorausgesetzt, das gleiche Verhalten zeigt.

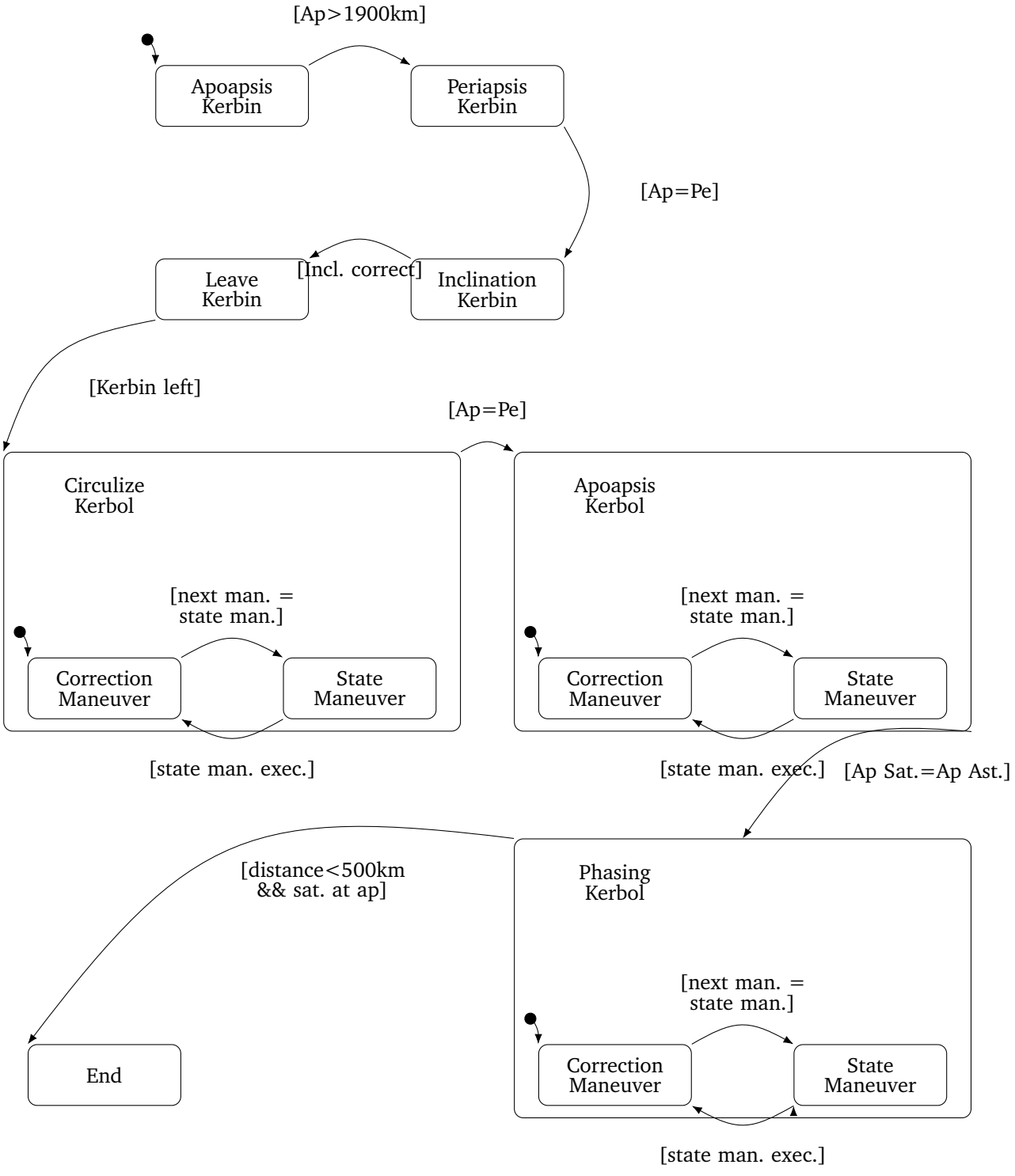


Abbildung 15.5.: State Machine Orbitkalkulation

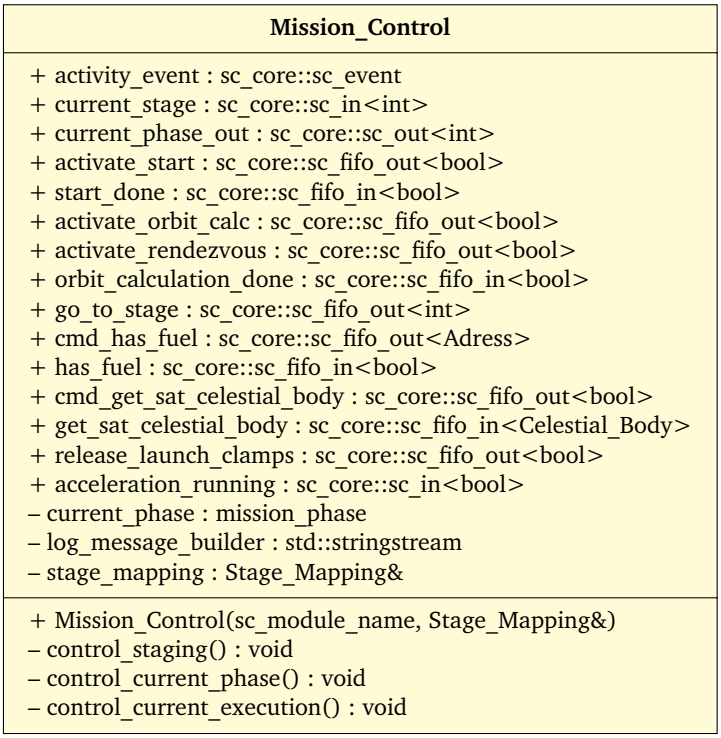


Abbildung 15.6.: Klasse: Mission Control

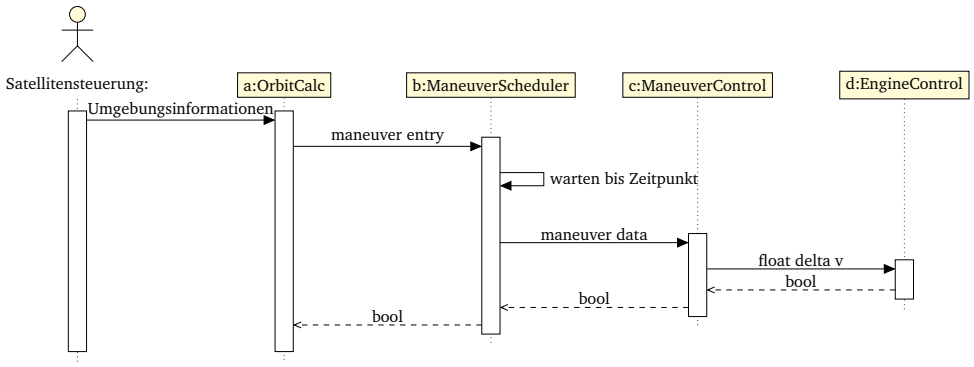


Abbildung 15.7.: Sequenzdiagramm: Beschleunigungsmanöver während der Orbitkalkulation

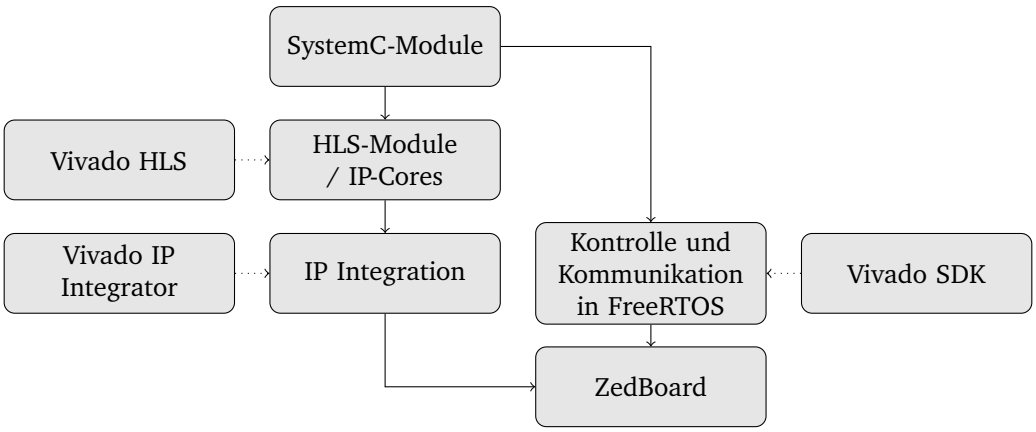


Abbildung 15.8.: Geplante, nicht realisierbare Implementierung der Satellitensteuerung.

15.3. Implementierung

Dieser Abschnitt beschreibt die Implementierung des logischen Modells der Satellitensteuerung auf den ZedBoards. Insbesondere wird auf das Vorgehen bei der Portierung eingegangen und weitere Entwicklungsschritte werden erläutert.

15.3.1. Hindernisse auf dem Weg zur Hardware

Bei der Implementierung des zuvor erstellten SystemC-Modells sind mehrere Probleme aufgetreten, welche das weitere Vorgehen zum Teil stark verändert haben. Die ursprüngliche Planung, aus den SystemC-Modulen Hardware-Module über High-Level-Synthese zu erzeugen, musste bereits zu Beginn der Implementierungsphase verworfen werden, siehe Abschnitt 15.3.1. Bei der Umstellung des kompletten Modells haben sich weitere Probleme mit der Toolchain ergeben, welche in **??** erläutert und gelöst werden.

Probleme mit der Modul-Synthese

Ursprünglich war vorgesehen, wie in Abbildung 15.8 die Module aus dem SystemC-Modell direkt in Module für Vivado HLS zu überführen. Über die Zeit der Modellentwicklung haben sich die Module jedoch so verändert, dass diese nicht ohne großen Aufwand als Hardwaremodul umsetzbar sind. Es wurden vielfach Konstrukte verwendet, welche Vivado HLS nicht verarbeiten kann

Eine Evaluation mit mehreren Projektbeteiligten hat ergeben, dass der Aufwand, die Module synthetisierbar zu machen für die verbleibende Zeit zu groß ist. Daher wurden diese, wie in Abbildung 15.9, als C++-Klassen umgesetzt und lediglich einige Funktionen auf die Hardware ausgelagert.

Inkompatiblitäten der Software

Nach der Entscheidung einen Großteil des Projektes auf die Software umzuziehen, musste die Entscheidung für das Betriebssystem neu evaluiert werden, siehe Unterabschnitt 15.3.3. Als Konsequenz aus dieser Entscheidung hat sich herausgestellt, dass die ursprünglich zur Verwendung gedachte Version Xilinx Vivado 2016.4 nicht ausreichend ist, da die dort enthaltene

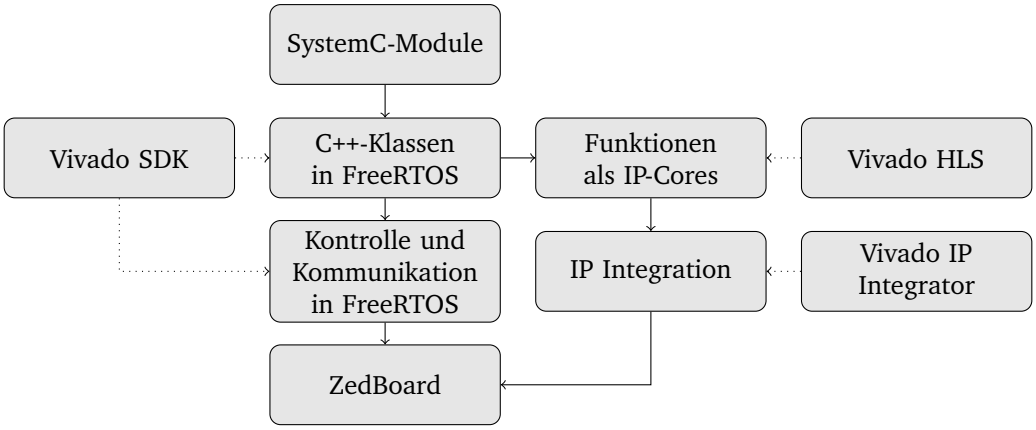


Abbildung 15.9.: Durchgeführte Implementierung der Satellitensteuerung.

Version des Betriebssystem zu alt ist. Der Wechsel auf die neuere Version Xilinx Vivado 2017.2 löste zwar das Problem, führte aber dazu, dass die Entwicklung auf den Privatrechnern, sowie auf den beiden Rechnern der Projektgruppe eingeschränkt wurde, da die Workstations nicht mit der neuen Version ausgestattet werden konnten.

15.3.2. Designentscheidungen beim Refactoring

Für das Refactoring vom logischen Modell zu einem Microcontroller-Programm auf dem ZedBoard, mussten zwei Designentscheidungen bezüglich des Betriebssystems und der Programmiersprache getroffen werden. Bei ersterem stehen *Petalinux*, *FreeRTOS* und *libmetal* für das Zedboard zur Auswahl. Letztere Designentscheidung beschränkt sich auf C++ und C. Nach dem Refactoring sollen alle Spuren von SystemC durch Funktionen des Betriebssystems, der Programmiersprache oder den Funktionen die Xilinx für das ZedBoard zur Verfügung stellt, ersetzt werden.

15.3.3. Betriebssystem

Als Favorit stand schon vor diesem Schritt FreeRTOS, allerdings kamen auf dem Zedboard als Alternative auch Petalinux[92] und libmetal[45] in frage.

Im Folgenden werden die Vor- und Nachteile der jeweiligen Betriebssysteme erläutert.

- *FreeRTOS:* Die Vor- und Nachteile von FreeRTOS wurden in Tabelle 15.1 aufgelistet und nach Priorität sortiert (je früher das Element aufgeführt ist, desto wichtiger ist). Ein Nachteil von FreeRTOS ist beispielsweise, dass es nicht den POSIX-Standard für Betriebssystemfunktionen berücksichtigt. Da es zum POSIX-Standard und zu FreeRTOS aber keine nennenswerten Vorkenntnisse in der Projektgruppe gab, hat dies nur eine geringe Priorität. Allerdings besitzt FreeRTOS nur eine Multicore-Erweiterung, die im Rahmen einer Masterarbeit entstanden ist, scheinbar nicht aktiv gepflegt wird und von dem der Entwickler abrät sie zu benutzen.
- *Petalinux:* Petalinux ist ein von Xilinx gepatchter Linuxkernel und Entwicklungsumgebung für FGPA-basiertes System-on-Chip-Design[83, S. 6]. Anwendungsprogramme

können als Kernelmodul in den Kernel hineinkompiliert werden, oder als Prozess im Userspace laufen[83, S. 57, 55]. Durch Mechanismen wie OpenAMP[89] werden auch Multicore-Systeme unterstützt, auf dem zweiten Core läuft dann üblicherweise ein bare-metal System oder RTOS, welches mit Petalinux über ein Remote-Procedure-Call-Interface und shared memory kommuniziert. Dieser Vorteil wird leider davon überschattet, dass die Entwicklung mittels Petalinux faktisch nur von einem unixoiden Entwicklungsrechner aus möglich ist, da alle Tutorials implizit davon ausgehen, dass der Anwender eine Shell benutzt. Hinweise auf die Entwicklung unter Windows wurden keine gefunden.

- *libmetal:* Diese Library für OpenAMP-Systeme soll als Abstraktionsebene dienen, sodass man unabhängig von dem Zielbetriebssystem Programme schreiben kann. Allerdings liefert Xilinx Vivado 2017.2 für diese Library keine Beispiele und auch eine Recherche im Internet erbrachte keine weiteren Informationen.

Tabelle 15.1.: Vor und Nachteile von FreeRTOS

Vorteile	Nachteile
• Simpel aufgebaut und umfassend dokumentiert • einfacher Workflow • Beispielprogramme decken die vielen Voraussetzungen (UDP-Kommunikation via lwIP, Tasks, Queues, Speicherverwaltung, ...) für unser Vorhaben ab • hardwarenahe Programmierung • Kompatibel zu C++	• keine stabile Multicore-Unterstützung • nicht POSIX-kompatibel • in C geschrieben

Tabelle 15.2.: Vor- und Nachteile von Petalinux

Vorteile	Nachteile
• Vollständiges Linux • POSIX-konform • Multicore fähig durch OpenAMP • Petalinux unterstützt C und C++	• komplexerer Workflow[84] • komplexere Toolchain[84] • Workflow sieht Linux-Entwicklungssystem vor • Wenige Beispielprojekte

Tabelle 15.3.: Vor- und Nachteile von libmetal

Vorteile	Nachteile
• Unabhängig von der Zielplattform (Linux/RTOS)	• keine Beispielprojekte

Fazit

Es wurde FreeRTOS gewählt. Die Entwicklung ist einfach und unabhängig vom Betriebssystem, auf dem es entwickelt wird.

Programmiersprache

Die Klassen und Sequenzdiagramme implizieren zwar schon, dass C++ eingesetzt wird, allerdings lässt sich auch in ANSI C objektorientiert programmieren[70]. Für C spricht des Weiteren, dass auch FreeRTOS (der Favorit aus dem vorherigen Abschnitt) in C geschrieben wurde. Allerdings fehlt es C neben der richtige Objektorientierung an vielen Eigenschaften, die die Entwicklung vereinfachen, die C++ bereits mitbringt (v.a. Standard Template Library, streams, strings, Algorithm-Library). Es wird empfohlen, C- und C++-Quellcode nicht zu *mischen*, ohne Vorkehrungen zur Kompatibilität zu treffen (vgl. [15], [80]). Diese wurden aber bei FreeRTOS seit Version 9 vorgenommen, sodass FreeRTOS auch mit einem C++-Compiler übersetzt werden kann. Aufgrund dessen wurde sich letztendlich dazu entschieden, die eigentliche Satellitensteuerung in C++ zu schreiben.

15.3.4. Vom logischen Modell zur Zielimplementierung

Aus dem vorherigen Abschnitt geht hervor, dass wir FreeRTOS und C++ verwenden werden. In diesem Abschnitt soll nun vorgestellt werden, wie die entsprechenden Module und Datenstrukturen auf FreeRTOS übertragen werden können. Des Weiteren werden hier Abschnitte beschrieben, die aus dem logischen Modell nicht übernommen werden konnten, bzw. stark verändert werden mussten.

Datenstrukturen Zuerst können die Datenstrukturen übertragen werden. Diese sollten nicht mehr die `sc_trace`-Funktion besitzen. Dies kann aber einfach über das Macro `SYSTEMC_INCLUDED` entschärft werden, sodass die Datenstrukturen noch zum logischen Modell aufwärtskompatibel bleibt.

Module Als nächstes können Module übertragen werden. Ein Modul (resp. eine Klasse in SystemC) besteht üblicherweise aus internen Variablen, FIFOs, Signals, Threads und Methoden. Variablen können direkt übernommen werden. Signale, die zur Prozesssynchronisation verwendet werden, müssen entweder durch Semaphoren[37, Kapitel 4] oder FIFOs ersetzt werden.

Für FIFOs wurde die Klasse `fifo` (in `vivado/sdkfiles/system/channels.hpp`) geschrieben. Diese stellt eine Abstraktion für Queues in FreeRTOS[37, Kapitel 3] dar.

Threads werden in FreeRTOS Tasks genannt[37, Kapitel 2]. Jeder Task benötigt eine auszuführende Funktion, einen Namen, eine Stackgröße und Priorität. Des Weiteren kann noch ein beliebiger Parameter und ein Pointer auf ein Taskhandle angegeben werden. Ein Taskhandle ist nur nötig, falls man den Task fernsteuern will, z.B. indem der Task sich periodisch schlafen legt und aus einem anderen Task oder Interrupt geweckt wird.

Der größte Unterschied zu SystemC ist hierbei, dass Task-Funktionen (innerhalb einer Klasse) statisch sein und einen Pointer-Parameter besitzen müssen - im Gegensatz zu SystemC,

in dem Threads keine Parameter besitzen sollen und nicht statisch sein sollen. Der Parameter wurde `self` genannt. Bei der Erstellung des Tasks mittels `xTaskCreate` wird dann dem Task `this` als Parameter mitgegeben, damit dieser dann wieder auf objektgebundene Funktionen und Variablen zugreifen kann.

Darüber hinaus besitzen FreeRTOS und SystemC weitere Unterschiede, wie Zeitscheiben-Scheduling mit Interrupts und Prioritäten. SystemC besitzt dies nicht. Dies ermöglicht aktives Warten, ohne dass ein Task das gesamte System blockiert.

network: Abstraktionslayer zu lwip

Für die Initialisierung des Netzwerkes, und um lwip einfacher in das FreeRTOS Umfeld zu integrieren, wurde die Netzwerkklasse `network` geschrieben. Des Weiteren können Strukturen fester Größe von einem UDP-Port in eine FIFO geschrieben werden, bzw. aus einer FIFO via UDP an eine IP-Adresse übertragen werden.

In der Praxis hat sich dies allerdings als nicht sehr praktikabel erwiesen, da hierfür immer ein weiterer Sender, bzw. Empfängerthread erzeugt werden musste und es nicht die Möglichkeit gab, Empfänger im Nachhinein zu ändern. Des Weiteren hat meistens die Payload der UDP-Pakete variiert, weswegen direkt lwip benutzt wurde. Die network-Klasse wird aber weiterhin zur Initialisierung des Netzwerkes verwendet. Das letzte Byte der IP-Adresse und der MAC-Adresse der einzelnen Zedboards ermittelt sich anhand der ID. Diese wird wiederrum durch die Schalter (`axi_gpio_sws`) als „one-hot"-Kodierung auf dem Zedboard ermittelt. Alle Zedboards haben die IP-Adresse `192.168.0.(10+id)`, sowie die MAC-Adresse `0:1d:c0:ff:ee:(id)`. Ist also der erste Switch (Switch 0) gesetzt. besitzt das Zedboard die IP `192.168.0.11`.

Die Ermittlung über die Schalter führt zu Problemen mit dem Master Selector, da dieser davon aus geht, dass vier Zedboards existieren. Diese sollen demzufolge die IPs 192.168.0.11 bis 192.168.0.14 besitzen. Ist dies nicht der Fall wird nicht explizit ein Fehler geworfen, kann aber zu fehlerhaftem Verhalten führen.

ksp_time

Im logischen Modell wurde hierfür `sc_ksp::wait()` äquivalent zum `wait` aus SystemC aufgerufen. Diese Funktion hat die SystemC-Simulation angehalten, bis zum entsprechenden Zeitpunkt in KSP gewarpt wurde. Dies ist in FreeRTOS nicht möglich.

Auf dem Zedboard wird nun zwischen Tasks unterschieden, die von der Zeit bei KSP abhängig sind („waitable"), und welche die davon nicht abhängig sind. Erstere müssen vorher mittels `ksp_time::register_waitable()` registriert werden. Wenn alle diese Tasks warten, wird zum kürzest-möglichen Zeitpunkt vorgespult. Falls ein Task durch eine FIFO blockiert, wird er als „unendlich-wartend" eingetragen. Die Klassendiagramme zu `ksp_time` sind in Unterabschnitt D.1.2 dargestellt.

Fieldbus

Für den Feldbus mussten die Klassen `Payload_Vector` und `Fieldbus_Packet` aus dem logischen Modell überarbeitet werden, da diese zu sehr den Heap beanspruchen.

Der `Payload_Vector` besitzt nun einen Buffer `buf` fester Größe (Hier 80 Bytes, siehe Konstante `BUF_SZ`) und Informationen über den aktuellen „Füllstand" (write index, `sz`) und Le-

sestand (read index, `read`) des Buffers. Mit der überladenen Methode `push_back` können Parameter verschiedener Datentypen (bool, float, double, sowie Enumeratoren) nacheinander in den Buffer geschrieben werden. Hierbei werden die Daten schon Feldbus-kompatibel kodiert. Dies hat den Vorteil, dass der Buffer direkt per `memcpy` in die Binärrepresentation des Feldbus-Paketes übernommen werden kann. Über die überladene Methode `get(ptr)` können nacheinander (FIFO-Verhalten) die Daten wieder ausgelesen werden. Sie werden dann in den entsprechenden Pointer geschrieben.

Die Methoden `Fieldbus_Packet::to_binary` und `Fieldbus_Packet::from_binary` wurden komplett umgeschrieben. Erstere benötigt nun einen Pointer auf einen Zielbuffer, in dem die Binärrepresentation geschrieben werden soll und gibt die Anzahl der geschriebenen Bytes zurück. Letztere benötigt einen Pointer (`packet`) auf einen Buffer in dem sich die Binärrepresentation befindet, die Größe des Paketes (`pack_sz`), sowie einen Pointer auf die Zielstruktur (`conv`). Die Methode deserialisiert immer und gibt *true* zurück, falls Länge und Prüfsumme stimmen, ansonsten *false*.

Die angepassten Klassen sind in Unterabschnitt D.1.2 dargestellt.

15.3.5. Funktionen auf der Hardware

Gemäß Unterabschnitt 15.3.1 sind auf der Hardware einzelne Funktionen implementiert, welche aus der Software heraus angesteuert werden. Bei der Implementierung der Funktionen wurden diese komplett aus der Software übernommen, damit sie die gleiche Signatur wie die zu ersetzende Softwarefunktion hat. Die Funktionen wurden so ausgewählt, dass diese in Hardware implementierbar sind (also keine nicht-synthetisierbaren Konstrukte beinhalten), gleichzeitig aber auch ausreichend aufwendige Operationen beinhalten, damit sich der Aufwand der Implementierung und der Kommunikation zwischen Hard- und Software lohnt. Solche Funktionen lassen sich vielfach in der Orbitkalkulation finden, daher wurden aus dieser einige Funktionen ausgewählt und mit Vivado HLS als IP Core umgesetzt.

Auswahl der Funktionen

In der Orbitkalkulation finden auf Grund der Positionen von Asteroid und Satellit die Berechnungen für die auszuführenden Manöver statt. Dazu sind zum Teil eine Reihe komplexer Kalkulationen notwendig, welche ideal auf Hardware ausgeführt werden können, um Rechenzeit zu sparen.

Implementierung der Funktionen

Die Implementierung von einzelnen Funktionen ist leicher umzusetzen als die von ganzen Modulen. Funktionen haben feste Ein- und Rückgabewerte, welche im Verlauf der Funktion verarbeitet und erzeugt werden. Die Funktionen der Orbitkalkulation erhalten in der Regel mehrere double-Werte und geben auch einen solchen zurück.

Zu jedem IP Core wird eine Testbench erstellt. Der erste Entwicklungschritt nach dem Schreiben des Codes ist das Testen des Codes. Anschließend erfolgt mit der Synthese ein Refactoring des Codes auf die Zielplattform. Die Zielplattform ist mit dem ZedBoard vorgegeben. Die Synthese ist in der Entwicklung ein entscheidender Schritt, denn hier zeigt sich, ob der Code

nicht nur kompilierbar sondern auch in Hardware abbildbar ist. An diesem Punkt scheiterte der ursprüngliche Plan, ganze SystemC-Module umzusetzen, da diese sehr umfangreiche Änderungen benötigt hätten, um synthesefähig zu werden.

Nach der Synthese kann das Modul erstmalig als simulierte Hardwarekomponente ausgeführt werden. Um die Funktionsfähigkeit zu testen, kommt wieder die ursprüngliche Testbench ins Spiel. Bei der sogenannten Co-Simulation wird die erzeugte Hardwarekomponente mit der C-Testbench ausgeführt. So kann verifiziert werden, dass die Funktion der Hardware der Software entspricht, eine entsprechende Testabdeckung vorausgesetzt. Für die Co-Simulation muss erstmal die Modelierungssprache bestimmt werden, mit welcher gearbeitet werden soll. In der Projektgruppe wurde sich auf VHDL geeignet, da hier die größten Kompetenzen vorgelegen haben. Im Idealfall sollte die Sprache jedoch nicht relevant sein, da auf der Ebene der Hardwarebeschreibungssprachen keine Änderungen vorgenommen werden sollen.

Nach erfolgreicher Co-Simulation wird das Modul als IP Core in einem IP Repository für den Vivado IP Integrator exportiert.

Implementierung der Kommunikation

Neben Berechnung muss der IP Core natürlich auch eine Schnittstelle zur Kommunikation besitzen. Über diese Schnittstelle kann die Software den IP-Core steuern, Daten an diesen senden und die Ergebnisse abfragen. Für die Verwendung der IP Cores hat sich die Nutzung des AXI-Lite Busses als ideal erwiesen.

Für die Ansteuerung in der Software werden automatisch Treiber generiert, welche den Registerzugriff abstrahieren und auch die Steuerung vereinfachen. Die Ansteuerung folgt dabei der Ressource XAPP745[93] von Xilinx. Das Board Support Package im Vivado SDK erzeugt ein Makro mit der Adresse der IP Cores. Dieser wird anschließend über den Treiber initialisiert, mit Eingabedaten gefüllt und gestartet. Anschließend wird aktiv auf das Ergebnis gewartet, da die Alternative mit Interrupts sich als Problematisch unter FreeRTOS erwiesen hat – scheinbar gibt es Probleme mit Interrupts und Tasks, welche Letztere zum Stillstand bringen. Zuletzt wird das Ergebnis aus dem IP Core ausgelesen und weiter verarbeteitet.

Integration der Hardware

Die im vorherigen Abschnitt erzeugten IP Cores werden zusammen mit weiteren Komponenten im Vivado IP Integerator, siehe Abbildung 15.10, zu einem Gesamtsystem verbunden. Hauptkomponente ist dabei das Zynq Processing System, welches Zugriff auf die Interna des Zedboards, wie zum Beispiel den Speicher, ermöglicht. Erweitert wird dieser durch einen Reset-Baustein sowie den AXI-Interconnect, welcher die zahlreichen weiteren Komponenten mit dem AXI-Bus verbindet. Der AXI-Interconnect ist mit den GPIOs zur Ansteuerung von Buttons, Schaltern und LEDs sowie den IP Cores verbunden.

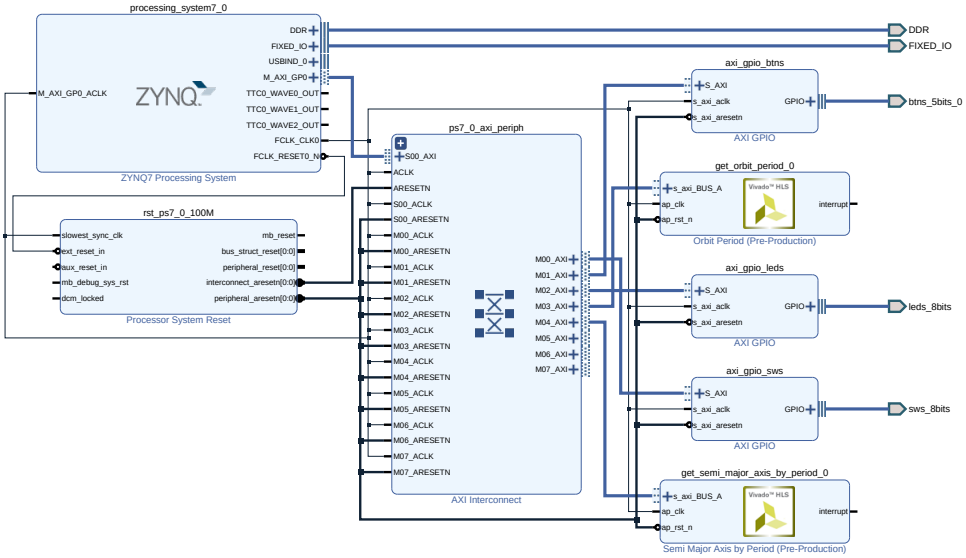


Abbildung 15.10.: Generiertes Blockdesign mit dem Zynq, dem AXI-Interconnect, den GPIOs und den IP Cores

15.3.6. Demonstrator

Die Implementierung der Satellitensteuerung wird zum Abschluss als Demonstrator aufgebaut. Der Demonstrator besteht aus dem Simulator-PC, dem Fehlerinjektions-PC, den vier Zedboards sowie der Vernetzung untereinander. Die beiden PCs gehören nicht direkt zum Aufbau der Satellitensteuerung und werden hier daher nicht weiter behandelt.

Die vier Zedboards werden mit einer identischen Konfiguration aus Hard- und Software bespielt und sind per Netzwerk untereinander und mit den PCs verbunden. Die ZedBoards bestimmen untereinander einen Master, die anderen werden zu Slaves.

16. Fehlerinjektion

In diesem Kapitel werden System- und Modulentwurf, sowie die Implementierung des Teilsystems Fehlerinjektion vorgestellt.

16.1. Systementwurf

In diesem Abschnitt wird der Systementwurf der Fehlerinjektion vorgestellt. Dies beinhaltet den Systementwurf der Fehlerinjektion auf dem Host-PC, der Fehlerinjektion auf dem Mikrocontroller, der Fehlerinjektion auf dem FPGA und den folgenden Protokollen:

- Das Protokoll zwischen der Fehlerinjektion auf dem Host-PC und der Fehlerinjektion auf dem Mikrocontroller
- Das Protokoll zwischen der Fehlerinjektion auf dem Host-PC und dem Simulator
- Das Hardware-Software-Protokoll

16.1.1. Fehlerinjektionskomponente auf dem Fehlerinjektions-PC

In diesem Abschnitt wird der Entwurf des Teilsystems Fehlerinjektion auf dem FI-PC (Abbildung 16.1) vorgestellt. Die FI-Komponente besteht aus acht Teilkomponenten, welche über mehrere Schnittstellen untereinander oder mit der Umgebung, also der Satellitensteuerung (Satellite Control), dem Simulator (Simulator) und dem Nutzer (User), kommunizieren können.

So erhält die FI-Komponente über den Simulator-Satelliten-Kanal (Simulator-Satellite Channel) die Umweltdaten (environment signals) des Simulators und die Steuersignale (control signals) der Satellitensteuerung, welche immer an das Monitoring (monitoring) der Fehlerinjektion weitergeleitet werden (fetch data). Genaueres dazu wird in Abschnitt 16.1.1 erläutert. Sollen keine Fehler injiziert werden, werden die Daten des Simulator-Satelliten-Kanals direkt an die Satellitensteuerung oder den Simulator weiter geleitet. Soll jedoch ein Fehler injiziert werden, werden die Daten nach Aktivierung seines Aktivierungsmuster vom Fehlerinjektor (Fault Injector) manipuliert.

Eine ähnliche Aufgabe zum Simulator-Satelliten-Kanal hat auch die Log-Protokoll-Komponente (Log Protocol Component), welche Informationen über den Satellitensteuerungszustand erhält und an das Monitoring (fetch log) und den Simulator weiterleitet. Diese Daten werden jedoch nicht manipuliert.

Ob Fehler injiziert werden, legt der Controller (Controller) fest. Dieser erhält von der Nutzerschnittstelle (User Interface) Informationen, ob und welche Fehler injiziert werden sollen. Wie genau die Nutzerschnittstelle funktioniert wird in Abschnitt 16.1.1 aufgeführt. Außerdem übergibt der Controller dem Monitoring Informationen über die Injektion von Fehlern (fetch data). Auch erhält der Controller von der Satellitensteuerung die Information, ob diese einen bestimmten Zustand geladen hat (load state) und bestätigt diesen (confirm state). Wird ein neuer Zustand der Satellitensteuerung geladen, werden alle injizierten Fehler entfernt. Weiter teilt der Controller dem Simulator mit, ob die FI-Komponente aktiv oder inaktiv (deliver fault injection state) ist.

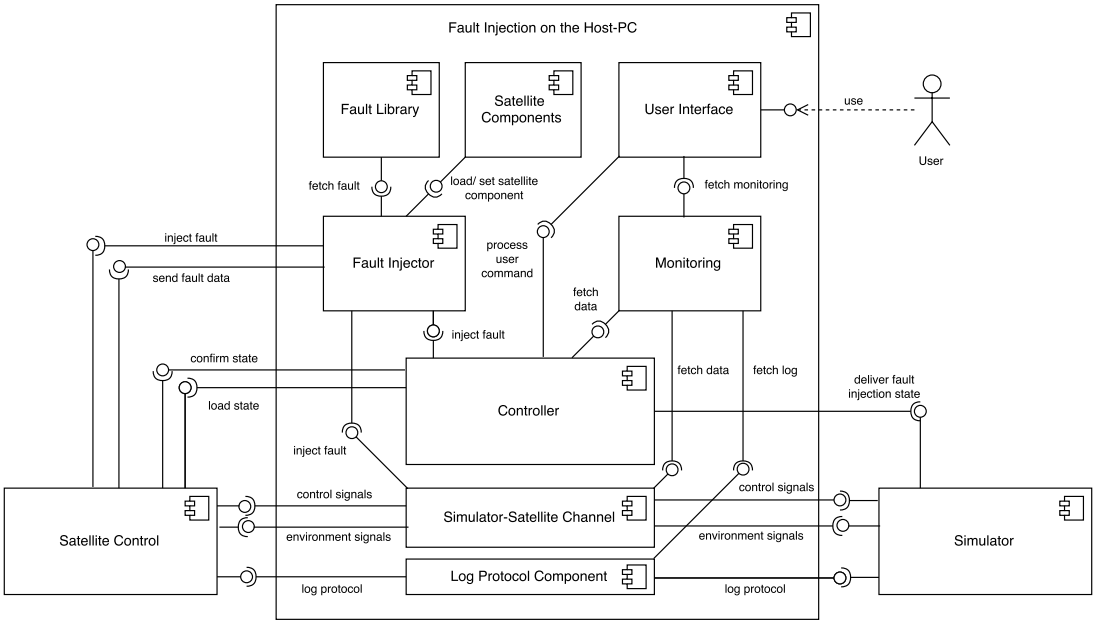


Abbildung 16.1.: Komponentendiagramm der Fehlerinjektionskomponente auf dem FI-PC

Soll ein Fehler injiziert werden, gibt der Controller dem Fehlerinjektor den Auftrag zum Injizieren eines Fehlers (inject fault). Dieser injiziert, nach dem Auftrag des Controllers, mit Hilfe der Fehlerbibliothek (Fault Library) und den Satellitenkomponenten (Satellite Components), Fehler (inject fault) entweder direkt in die Satellitensteuerung oder den Simulator-Satelliten-Kanal. Dabei stellt die Fehlerbibliothek die Funktionen zur Injektion der verschiedenen Fehler zur Verfügung (fetch fault) und die Satellitenkomponenten geben Informationen darüber, wo ein Fehler injiziert werden kann (load/ set satellite component). So können zum Beispiel die Umweltdaten aus dem Simulator manipuliert und dann an die Satellitensteuerung weitergeleitet werden.

Systementwurf der Steuerung

Die Steuerung wird über die Nutzerschnittstelle ausgeführt. Sie bezieht ihre Daten vom Controller, sowie vom Monitoring und sendet die Nutzerbefehle an den Controller. Die anderen Komponenten werden nur indirekt über den Controller angefragt. Die Steuerung soll die folgenden Aufgaben (siehe Abschnitt 13.4.5 der Spezifikation) anbieten:

- Das Abrufen des Monitorings direkt aus der Monitoring-Komponente
- Das Abfragen und Setzen des Fehlerinjektions-Zustandes aus dem Controller
- Das Injizieren und Ausschalten von Fehlern (Diese Befehle werden dem Controller übergeben, welcher sie mit dem Fehlerinjektor injiziert.)
- Das Ausgeben von Daten aus der Fehlerbibliothek (Diese Anfrage geschieht indirekt über den Fehlerinjektor und dem Controller.)
- Das Ausgeben von Daten der Satellitensteuerungskomponenten (Diese Anfrage geschieht indirekt über den Fehlerinjektor und dem Controller.)

- Das Laden eines Zustands der Satellitensteuerung über den Controller
- Das Weitergeben von Fehlerexperimenten an den Controller (Der Controller verarbeitet das Fehlerexperiment und injiziert die Fehler mit den angegeben Aktivierungsmustern.)

Systementwurf des Monitorings

Für das Monitoring ist nur eine Komponente vorgesehen. Diese Komponente hat die Aufgabe Methoden anzubieten, die die Nutzerschnittstelle anfragen kann, um dem Nutzer Informationen über die bereits in der Spezifikation (Abschnitt 13.4.5) genannten Aufgaben zu geben:

- Informationen zu Fehlern
- Informationen zu Fehlerexperimenten
- Informationen zu dem Fehlerinjektions-Zustand
- Informationen zu Satellitensteuerungskomponenten
- Informationen zu dem geladenen Missionsstand (Satellitenzustand - Phase)
- Informationen zu den Umweltsignalen
- Informationen zu den Steuerungsbefehlen
- Informationen des Satellitensteuerungslogs

Alle Informationen werden dabei von Komponenten der Fehlerinjektion auf dem FI-PC verwaltet, sodass die Monitoring-Komponente diese Informationen über einfache Methoden der jeweiligen Komponente abfragen kann. Im Genauen greift das Monitoring die Daten der Simulator-Satelliten-Kommunikation direkt aus dem Simulator-Satelliten-Kanal und den Satellitensteuerungslog aus der Log-Komponente. Die Informationen zu den Fehlern und Fehlerexperimenten, dem Fehlerinjektions-Zustand, den Satellitensteuerungskomponenten und den geladenen Missionsstand werden indirekt über den Controller angefragt. Nach der Fehlerinjektionsspezifikation ist ein automatisches Auswerten des Logs nicht geplant, sodass der Log ohne weitere Verarbeitung über die Monitoring-Komponente an den Nutzer weitergeleitet werden kann.

16.1.2. Fehlerinjektion auf dem Mikrocontroller und dem FPGA

Die Fehlerinjektion auf dem Mikrocontroller (siehe Abbildung 16.2) besteht aus fünf Komponenten:

- *Komponente zur Kommunikation mit dem Fehlerinjektor auf dem FI-PC:* Die Host-PC Fault Injektor Communication-Komponente nimmt Fehlerinjektionsbefehle entgegen und sendet diese an die jeweilige zuständige Komponente. Zum Beispiel ist für Fehler in Umwelt- und Steuerungssignalen auf Seiten des Mikrocontrollers die Komponente zur Fehlerinjektion in die Satellitensteuerungs-Simulator-Kommunikation integriert. Sind die Fehlerinjektionsbefehle für das FPGA gedacht, sendet die Komponente die Fehler an die FPGA-Fehlerinjektor-Kommunikationskomponente (FPGA Fault Injector Communication) weiter. Von den jeweils zuständigen Komponenten werden Fehlerdaten übermittelt,

die durch diese Komponente an die Fehlerinjektion auf dem FI-PC gesendet werden. Unter Fehlerdaten versteht man Daten wie beispielsweise die (partielle) Aktivierung eines Fehlers.

- *Komponente zur Fehlerinjektion in die Ethernet-Kommunikation:* Die Fault Injector Ethernet Communication erhält von der Kommunikationskomponente die zu injizierden Fehler in den Umweltsignalen, den Steuerungssignalen und der Mikrocontrollerkommunikation, sowie die Aktivierungmuster die gebraucht werden, um die Fehler zu injizieren. Neben diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Umwelt- und Steuerungssignalen, sowie der Mikrocontrollerkommunikation ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Liegt die Komponente nicht auf dem FPGA, wird die Aktivierung über die FPGA-Fehlerinjektor-Kommunikationskomponente übertragen. Für eine bessere Übersichtlichkeit wurden die Verbindungen zu den Komponenten durch einen Verteiler (Activation-Distributer) dargestellt. Die Fehlerdaten werden wieder an die Host-PC-Kommunikationskomponente gesendet.

- *Komponente zur Fehlerinjektion in die Satellitensteuerung auf dem Mikrocontroller:* Die Fault Injector Microcontroller-Komponente erhält von der Kommunikationskomponente die zu injizierden Fehler und die Aktivierungmuster, die gebraucht werden, um die Satellitensteuerungsfehler zu injizieren. Neben diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Satellitensteuerungssignalen ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Hier gilt auch wieder: Liegt die Komponente nicht auf dem FPGA, wird die Aktivierung über die FPGA-Fehlerinjektor-Kommunikationskomponente übertragen. Die Fehlerdaten werden wieder an Host-PC-Kommunikationskomponente gesendet.

- *Komponente zur Fehlerinjektion in die Software-Hardware-Kommunikation:* Die Fault Injector SWHW-Communication-Komponente erhält von der FI-PC-Kommunikationskomponente die zu injizierden Fehler in die Software-Hardware-Kommunikation und die Aktivierungmuster, die gebraucht werden, um die Fehler zu injizieren. Neben diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Software-Hardware-Kommunikationssignalen ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Hier gilt auch wieder: Liegt die Komponente nicht auf dem FPGA, wird die Aktivierung über die FPGA-Fehlerinjektor-Kommunikationskomponente übertragen. Die Fehlerdaten werden wieder an die Host-PC-Kommunikationskomponente gesendet.

- *Komponente zur Weiterleitung der FPGA-Fehlerinjektionen:* Die FPGA-Fault Injector Communication-Komponente nimmt Fehlerinjektionsbefehle von der FI-PC-Kommunikationskomponente entgegen und sendet diese an das FPGA weiter. Die Rückmeldungen von dem FPGA werden an Host-PC-Kommunikationskomponente gesendet. Zudem dient diese Komponente als Weiterleitung von Aktivierungen auf dem Mikrocontroller zu den Fehlern der Satellitensteuerung auf dem FPGA.

Die Fehlerinjektion auf dem FPGA (siehe Abbildung 16.3) besteht aus vier Komponenten:

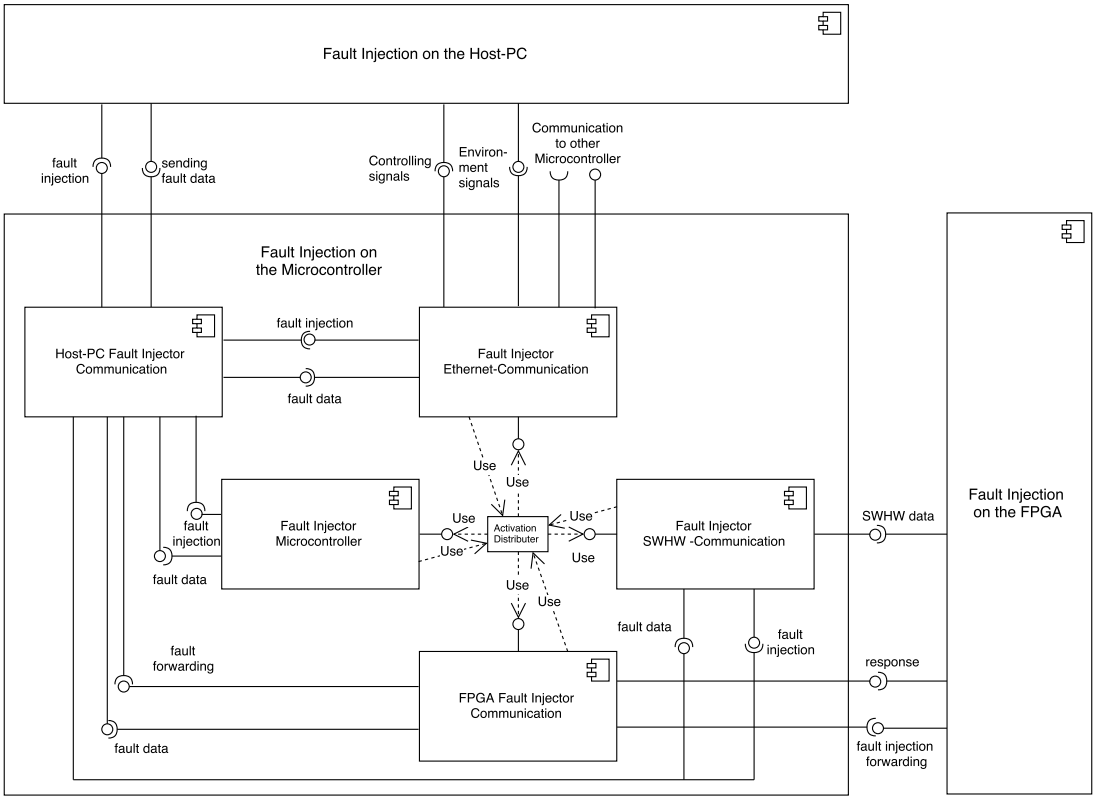


Abbildung 16.2.: Komponentendiagramm von der Fehlerinjektion auf dem Mikrocontroller

- *Komponente zur Kommunikation mit dem Mikrocontroller:* Die Microcontroller-FPGA-Communication-Komponente nimmt Fehlerinjektionsbefehle entgegen und sendet diese an die jeweilige zuständige Komponente. Von den jeweils zuständigen Komponenten werden Fehlerdaten übermittelt, die durch diese Komponente an die Fehlerinjektion auf dem Mikrocontroller gesendet werden. Zudem dient diese Komponente als Weiterleitung von Aktivierungen auf dem FPGA zu den Fehlern der Satellitensteuerung auf dem Mikrocontroller.

- *Komponente zur Fehlerinjektion in die Satellitensteuerung auf dem FPGA:* Die Fault Injector FPGA-Komponente erhält von der Mikrocontroller-Kommunikationskomponente die zu injizierden Fehler und die Aktivierungmuster, die gebraucht werden, um die Satellitensteuerungsfehler zu injizieren. Neben diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Satellitensteuerungssignalen ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Für eine bessere Übersichtlichkeit wurden auch hier die Verbindungen zu den Komponenten durch einen Aktivationsverteiler (Activation Distributer) dargestellt. Die Fehlerdaten und Rückmeldungen werden an die Mikrocontroller-Kommunikations-komponente gesendet.

- *Komponente zur Fehlerinjektion in die Software-Hardware-Kommunikation:* Die Fault Injector SWHW Communication-Komponente erhält von der Mikrocontroller-Kommunikationskompon-ente die zu injizierden Fehler in die Software-Hardware-Kommunikation und die Aktivierungmuster, die gebraucht werden, um die Fehler zu injizieren. Neben

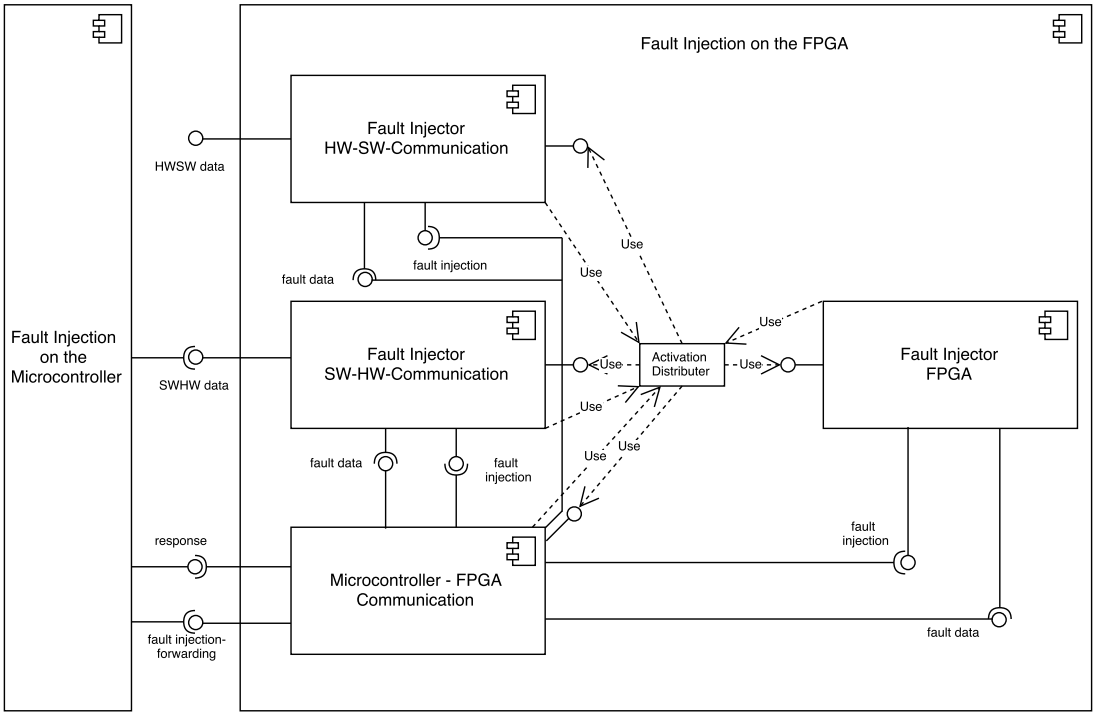


Abbildung 16.3.: Komponentendiagramm von der Fehlerinjektion auf dem FPGA

diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Software-Hardware-Kommunikationssignalen ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Die Fehlerdaten und Rückmeldungen werden wieder an die Mikrocontroller-Kommunikationskompo-nente gesendet.

- *Komponente zur Fehlerinjektion in die Hardware-Software-Kommunikation:* Die Fault Injector HWSW Communication-Kompnente erhält von der Mikrocontroller-Kommunikationskompo-nente die zu injizierden Fehler in die Hardware-Software-Kommunikation und die Aktivierungmuster, die gebraucht werden, um die Fehler zu injizieren. Neben diesen Aktivierungmustern werden auch die Aktivierungen anderer Fehler gespeichert, die von den Hardware-Software-Kommunikationssignalen ausgelöst werden sollen. Wird solch eine Aktivierung ausgelöst, werden die jeweiligen Komponenten benachrichtigt. Die Fehlerdaten und Rückmeldungen werden wieder an die Mikrocontroller-Kommunikationskomponente gesendet.

16.1.3. Protokoll zwischen der Fehlerinjektion auf dem Host und der Fehlerinjektion auf dem Mikrokontroller

Das Protokoll dient zur Kommunikation zwischen Fehlerinjektionskomponente auf dem Host-PC und der Satellitensteuerung. Dabei gibt es sechs Funktionen, welche mit Hilfe des Protokolls durchgeführt werden können, welche durch Klassifizierungsbits am Anfang einer Übertragung festgelegt werden. Die Funktionen des Protokolls sind:

- *Laden eines Satellitensteuerungszustandes*: Die Lade-Funktion übermittelt zusätzlich zu den Klassifizierungsbits auch den Zustand, in den die Fehlerinjektion zurück gesetzt werden soll.
- *Fehlerinjektion*: Bei der Fehlerinjektion werden zusätzlich zu den Klassifizierungsbits eine Fehlerinjektions-ID, die Fehlerdaten und die Aktivierungsmuster übermittelt. Die Fehlerinjektions-ID, die Fehlerdaten und die Aktivierungsmuster werden unten genauer beschrieben.
- *Rückmeldung zu der Fehlerinjektion*: Bei der Rückmeldung der Fehlermeldung wird neben den Klassifizierungsbits, welche den gleichen Wert wie die Funktion zur Fehlerinjektion haben, zusätzlich die Fehlerinjektions-ID sowie Fehlerinjektionszustandsbits übertragen. Dabei geben die Fehlerinjektionszustandsbits an, ob die Fehlerinjektion eines Fehlers vorbereitet wurde, erfolgreich war oder fehlgeschlagen ist.
- *Abbrechen einer Fehlerinjektion*: Bei dem Abbrechen einer Fehlerinjektion werden zusätzlich zu den Klassifizierungsbits eine Fehlerinjektions-ID und die Aktivierungsmuster übermittelt.
- *(partielle) Aktivierung einer Fehlerinjektion*: Bei der (partiellen) Aktivierung einer Fehlerinjektion wird neben den Klassifizierungsbits der Injektionspunkt der Aktivierung übertragen.
- *Freigabe von Ressourcen nach der Fehlerinjektion*: Bei der Funktion zur Freigabe von Ressourcen nach der Fehlerinjektion wird neben den Klassifizierungsbits ein Injektionspunkt übergeben, welcher Informationen darüber enthält, welche Ressourcen freigegeben wurden.
- *Anfragen von allen Fehlerzuständen*: Bei der Funktion zum Anfragen von allen Fehlerzuständen wird neben den Klassifizierungsbits keine weiteren Informationen übertragen.
- *Übertragung der Informationen zu den Satellitenkomponenten an die Fehlerinjektion*: Die Funktion zur Übertragung der Informationen zu den Satellitenkomponenten an die Fehlerinjektion überträgt neben den Klassifizierungsbits, die Informationen über die Satellitenkomponenten in Form von Injektionspunkten, sowie Informationen über mögliche injizierbare Fehler auf den einzelnen Komponenten.

Die *Fehlerinjektions-ID* beinhaltet zum einen den Injektionspunkt des zu injizierenden Fehlers, also Informationen auf welchem ZedBoard, auf welchem Modul und von welcher Fehlerinjektionskomponente auf dem Mikrocontroller (siehe Abbildung 16.2) ein Fehler injiziert werden soll. Zum anderen enthält die Fehlerinjektions-ID Informationen über den Fehlertypen des zu injizierenden Fehlers.

Die *Fehlerdaten* der Fehlerinjektion enthalten eine Klassifizierung, ob der Fehler eine Auftrittsdauer oder eine Häufigkeit des Auftretens besitzt, die Reaktivierungsanzahl der Fehlerinjektion und einen Wert, welcher Informationen über Dauer und Häufigkeit gibt. Weiter wird in den Fehlerdaten der Wert übergeben, auf den der zu manipulierende Wert gesetzt werden soll. Neben den Fehlerdaten ist das Aktivierungsmuster des zu injizierenden Fehlers Teil einer Fehlerinjektion, welches aus einem Aktivierungspunkt als Klassifizierung, einem Aktivierungswert, einem Häufigkeitswert, einem Dauerwert, einem Reaktivierungswert und einem

regulären Ausdruck besteht. Dabei gibt der Aktivierungspunkt Informationen über den Ort an, an dem das Aktivierungsmuster auftreten soll. Der Aktivierungswert gibt den Wert an, wann das Aktivierungsmuster die Fehlerinjektion aktiviert. Der Häufigkeitswert und der Dauerwert geben jeweils an, wie oft beziehungsweise wie lange ein Fehler injiziert werden soll, der Reaktivierungswert gibt die Anzahl an Reaktivierungen an und der reguläre Ausdruck gibt an, wie das Aktivierungsmuster mit weiteren Aktivierungsmustern in Verbindung steht.

16.1.4. Protokoll zwischen der Fehlerinjektion auf dem FI-PC und dem Simulator

Das Protokoll dient der Übertragung der Fehlerdaten von der Fehlerinjektion auf dem FI-PC, damit der Simulator das Vorspulen der Simulation zur Injektion eines Fehlers beenden kann. Das Protokoll baut auf UDP auf und soll ähnlich gestaltet sein, wie das Protokoll zum Vorspulen von der Satellitensteuerung zum Simulator. Für die Fehlerinjektion braucht das Protokoll

- ein Klassifizierungsbit,
- ein Paritätsbit,
- ein Zustandsbit und
- einen optionalen Datensatz (Zeitangabe).

Es werden keine Kopfdaten speziell für die Fehlerinjektion gebraucht, da das Protokoll über einen vorher festgelegten, eigenständigen Socket laufen soll. Das Klassifizierungsbit soll angeben, ob ein Aktivierungsmuster eines Fehlers übergeben wird oder der Zustand der Fehlerinjektion. Es sollen Aktivierungsmuster übertragen werden, da diese den genauen Fehlerinjektionszeitpunkt angeben, also dann, wenn die Fehlerinjektion aktiv ist und nicht vorgespult werden darf. Gleiches gilt auch für den Zustand der Fehlerinjektion, da hier der Nutzer zu dem aktuellen Zeitpunkt der Eingabe den Fehler injizieren will. Da allerdings alle von der Zeit unabhängigen Aktivierungen nur aktiviert werden können, wenn der Zustand der Satellitensteuerung verändert wird und bei einem Wechsel des Zustandes der Satellitensteuerung nicht vorgespult werden darf, reicht es aus, dem Simulator nur die Zeitabhängigen Aktivierungen zu übergeben. Das Paritätsbit gibt an, ob die Nachricht richtig übertragen wurde. Wenn der Simulator einen Übertragungsfehler entdeckt hat, soll der Simulator eine Neusendung anfragen dürfen. Das Zustandsbit ist für die Übertragung des Zustands der Fehlerinjektion und der Datensatz ist für die Übertragung der Zeitangabe zuständig.

16.1.5. Hardware-Software-Protokoll

Die Kommunikation zwischen der Fehlerinjektion auf dem Mikrocontroller und der Fehlerinjektion auf dem FPGA ähnelt der Kommunikation zwischen der Fehlerinjektion auf dem FI-PC und der Fehlerinjektion des Mikrocontroller, da überwiegend die gleichen Daten übertragen werden müssen. Allerdings ist der Aufbau des Hardware-Software-Protokolls durch die Übertragung über Register anders darzustellen. Da alle Register nur von einer Seite beschreibbar und von der anderen Seite auszulesen sind, lässt sich das Protokoll in Software-Hardware-Kommunikation und Hardware-Software-Kommunikation unterteilen. Zudem ist durch die be-

grenzte Anzahl an Registern auszugehen, dass zur Übertragung einer Fehlerinjektion mehrere Sequenzielle Übertragungen von Daten stattfinden werden.

Für die Software-Hardware-Kommunikation werden Register für folgende Daten gebraucht:

- Die *Protokollübertragungsdaten* zur Sicherstellung, wann und wie viele Daten übertragen werden
- Das *Klassifizierungsbit* zur Angabe, ob Fehlerinjektionen, Fehlerdaten oder Aktivierungen übergeben werden sollen
- Die *Fehlerinjektions-ID* zur Übertragung des Fehlertyps, der Fehler-Submaske und des Injektionspunktes
- Die *Aktivierungsmuster*, wann ein Fehler auf dem FPGA ausgelöst werden soll (Die Aktivierungsmuster bestehen aus dem zugehörigen Fehler, dem Aktivierungsklassifier, dem Injektionspunkt, dem auszulösenden Wert, der Häufigkeit und der Zeitdauer.)
- Die *Aktivierungen* als ausgelöste Auslöser von dem Mikrocontroller für Fehler auf dem FPGA. Die Aktivierungen bestehen aus dem zugehörigen Fehler und der Auslöser-ID
- Die *Fehlerdaten* zur Übertragungen der Fehlereigenschaften

Zusätzlich werden Register zur Übertragung von Fehlerinjektionsbefehlen, wie zum Beispiel für das Auslesen von Satellitensteuerungskomponenten, gebraucht.

Für die Hardware-Software-Kommunikation werden Register für folgende Daten gebraucht:

- Die *Protokollübertragungsdaten* zur Sicherstellung, wann und wie viel Daten übertragen werden
- Das *Klassifizierungsbit* zur Angabe, ob Fehlerinjektionen, Fehlerdaten oder Aktivierungen übergeben werden sollen
- Die *Fehlerinjektions-ID* zur Übertragung des Fehlertyps, der Fehler-Submaske und des Injektionspunktes (Dieser ist zum Beispiel für die Bestätigung einer Fehlerinjektion gebraucht.)
- Die *Aktivierungen* als Auslöser für Fehler auf dem FPGA (Die Aktivierungen bestehen aus dem zugehörigen Fehler und der Auslöser-ID.)
- Den *Fehlerinjektions-Zustandsbits* zur Übertragung, ob eine Fehlerinjektion vorbereitet werden konnte
- Zusätzlich werden Register zur Übertragung von Fehlerinjektionsrückmeldungen, wie zum Beispiel den Daten der Satellitensteuerungskomponenten, gebraucht.
- Wenn Satellitensteuerungskomponenten übertragen werden sollen, können zusätzlich Register zur Übertragung von möglichen Fehler in den Komponenten gebraucht werden.

Es ist anzumerken, dass einige Daten des Hardware-Software-Protokolls nicht so lang sind, wie die des Fehlerinjektionsprotokolls. Dies liegt daran, dass jedes FPGA mit genau einem Mikrocontroller verbunden ist und der Mikrocontroller die restlichen Daten, wie zum Beispiel die ID des ZedBoards, hinzufügen kann.

16.1.6. Komponententests

Das Teilprojekt Fehlerinjektion hat zu jeder Komponente eine Testbench erstellt. Diese befinden sich in `model/test/unit/fault_injection/host_pc`. Des Weiteren wurden Tests für Hardware-in-the-loop (HIL) geschrieben, die sich im Ordner `model/test/hil` befinden und das logische Modell mit der Fehlerinjektion sowie die Umwelt und Steuerdaten mit der zwischengeschalteten FI testen. Bei den Komponententests der Fehlerinjektion sind folgende Fehler aufgetreten, die nun aufgeführt werden. Die Komponententestfälle CTC001 bis CTC013 wurden erfolgreich durchgeführt, die Komponententestfälle der Host-PC-Satellitensteuerung-Kommunikation, der Host-PC-Mikrocontroller-Kommunikation, der Kommunikation zwischen mehreren Mikrocontrollern, der Mikrocontroller-Komponenten, der Hardware-Software-Kommunikation sowie der FPGA-Komponenten mussten aus Zeitgründen gestrichen werden.

Kennzeichnung:	**Fault 008**
Bezeichnung:	**Die FI kann nicht ordnungsgemäß geschlossen werden**
Referenz:	
Beschreibung	Die Fehlerinjektion besitzt keinen Schließbefehl und kann somit nicht ordnungsgemäß geschlossen werden. Das hat den Nachteil, dass die Message Queues zu der UI nicht geschlossen werden. Man muss den Prozess beenden. Bei einem Neustart werden alte Message Queues gelöscht und neue erstellt.
Korrektes Verhalten:	Die Fehlerinjektion besitzt einen Befehl, der die gesamte Fehlerinjektion inklusive Message Queues schließt.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Man erstellt einen Befehl, der von dem User Interface auf der Fehlerinjektorseite verstanden wird und schließt zuerst die Message Queues und dann den gesamten Prozess.

Kennzeichnung:	**Fault 009**
Bezeichnung:	**Die FI muss durch einen Mutex geschützt werden**
Referenz:	
Beschreibung	Es können Events bei den UDP-Sockets und der UI erzeugt werden. Da diese Konkurrent auf die FI zugreifen, muss die FI durch einen FI-Mutex geschützt werden.
Korrektes Verhalten:	Wenn immer ein Event erzeugt wird, muss die gesamte FI ausschließlich dem Event zum Zugriff erlaubt sein.
Behoben:	Voraussichtlich bis zur Endpräsentation beseitigt.
Lösungsansatz:	Eine Mutex für die gesamte Fehlerinjektionimplementierung implementieren. Man könnte auch jede einzelne Komponente mutexen, um mehrere Events gleichzeitig auf unterschiedlichen Ressourcen zu erlauben und somit einen Geschwindigkeitszuwachs zu bekommen, das wird allerdings aufgrund der Komplexität von Racing Conditions und den geringen Leistungsbedarfs der Fehlerinjektion nicht vorgeschlagen.

Kennzeichnung:	**Fault 010**
Bezeichnung:	**Boost Threads können nicht auf die GUI Elemente zugreifen**
Referenz:	
Beschreibung	Boost Threads können nicht auf die GUI Elemente zugreifen.
Korrektes Verhalten:	Threads haben Zugriff auf die Elemente.
Behoben:	**Beseitigt.**
Lösungsansatz:	Es müssen QThreads verwenden werden.

Kennzeichnung:	**Fault 011**
Bezeichnung:	**Threading bei der Message Queue Erstellung erzeugen Segmentation Faults**
Referenz:	
Beschreibung	Threads, die auf eine gemeinsame Message Queue-Ressource zugreifen, erzeugen Segmentation Faults.
Korrektes Verhalten:	Konkurrenter Zugriff ist möglich.
Behoben:	**Beseitigt.**
Lösungsansatz:	Auslagern der Message Queue Erstellung und Threads benutzen nur geöffnete Message Queues.

Kennzeichnung:	**Fault 012**
Bezeichnung:	**Das Finden vom Logging Pfad funktioniert nicht im Monitoring**
Referenz:	
Beschreibung	Das Finden des Logging Pfades geht aufgrund des unbekannten Ausführungsortes nicht.
Korrektes Verhalten:	Der korrekte Pfad wird gefunden. Alternativ wird ein Logging Ordner im Ausführungsort erstellt.
Behoben:	**Beseitigt.**
Lösungsansatz:	Es wird ein Logging Ordner im Ausführungsort erstellt.

16.2. Modulentwurf

Im Modulentwurf werden die Komponenten aus Kapitel 13 in Module unterteilt und diese in Form von Klassen- und Sequenzdiagrammen erläutert.

16.2.1. Fehlerinjektionsstrukturen

In diesem Abschnitt werden die Fehlerinjektionsstrukturen vorgestellt, die als Basis für die Module auf dem Host-PC, Mikrocontroller und FPGA benutzt werden. Dazu werden die Strukturen

- Fault,
- Fault_Injection_Command,
- Fault_Experiment und
- FI_Components_MC

und deren Unterstrukturen erläutert.

Die Struktur *Fault* (siehe Abbildung 16.4) besitzt die vier Attribute *fid* des Typs *Fault_ID* zur genauen Angabe des Fehlers, *fault_data* des Typs *Fault_And_Activation_Data<T>* für zusätzliche Fehlerdaten, sowie die Bool-Variable *has_activation_pattern*, welche angibt, ob der Fehler ein Aktivierungsmuster besitzt. Die Struktur *Fault_ID* ist zur Identifikation des Fehlers und besteht aus dem Typ des Fehlers *fault_type*, aus der *sub_mask* zur genaueren Angabe des Schwankungsfehlers und des Injektionspunktes *injection_point*. Aufgrund des Designs der *Fault_ID* kann nur ein Fehler eines Fehlertyps auf einem Signal eines Satellitenmoduls vorbereitet und injiziert werden, da keine weitere Unterscheidung in der Identifikation existiert. Das hat den Nachteil, dass zum Beispiel zwei Bitflips auf dem gleichen Signal nicht gleichzeitig vorbereitet werden können. In dem Zeitraum zwischen der Rückmeldung des ersten Bitflips und der Übertragung des Fehlerinjektionsbefehles des zweiten Bitflips kann somit der zweite Bitflip nicht injiziert werden. Dieser kurze Zeitraum könnte allerdings der entscheidende Moment gewesen sein, in dem der zweite Bitflip injiziert werden sollte. Dieser Nachteil wird allerdings zum Sparen von weiteren Daten in der Identifikation akzeptiert. Dennoch sollte der Nutzer der Fehlerinjektion über diesen Nachteil Bescheid wissen.

Die Struktur *Fault_ID* ist zur Identifikation des Fehlers und besteht aus dem Fehlertyp *fault_type*, aus der *sub_mask* zur genaueren Angabe des Schwankungsfehlers und des Injektionspunktes *injection_point*. Aufgrund des Designs der *Fault_ID* kann nur ein Fehler eines Fehlertyps auf einem Signal eines Satellitenmoduls vorbereitet und injiziert werden, da keine weitere Unterscheidung in der Identifikation existiert. Das hat den Nachteil, dass zum Beispiel zwei Bitflips auf dem gleichen Signal nicht gleichzeitig vorbereitet werden können. In dem Zeitraum zwischen der Rückmeldung des ersten Bitflips und der Übertragung des Fehlerinjektionsbefehles des zweiten Bitflips kann somit der zweite Bitflip nicht injiziert werden. Dieser kurze Zeitraum könnte allerdings der entscheide Moment gewesen sein, in dem der zweite Bitflip injiziert werden sollte. Dieser Nachteil wird allerdings zum Sparen von weiteren Daten in der Identifikation akzeptiert. Dennoch sollte der Nutzer der Fehlerinjektion über diesen Nachteil Bescheid wissen.

Die Struktur *Injection_Point* dient zur Angabe des Injektionspunktes eines Fehlers und der Angabe des Aktivierungspunktes eines Aktivierungsmusters. Der *Injection_Point* besteht aus der Angabe der Hardwareeinheit *hardware_component*, das ist entweder der Host-PC oder eines spezifisches ZedBoard oder einer Mischung aus Host-PC und einem ZedBoard für spezielle Aktivierungsmuster oder Invalid_Hardware_Component, was einer fehlerhaften Angabe oder einer nicht existierenden Hardwareeinheit entspricht. Weiter besteht der *Injection_Point* aus einer Satellitensteuerungskomponente *satellite_component* und dem genauen Signal *component_signal*, sowie dem *signal_bit* auf dem der Fehler injiziert werden soll.

Die Struktur *Fault_And_Activation_Data<T>* ist eine Template-Klasse, wodurch es ermöglicht wird, dass *value* je nach Fehler oder Aktivierungsmuster einen anderen Datentyp, entweder *bool*, *int*, *float* oder *double*, besitzen kann und dient der Angabe der Fehler- und Aktivierungsdaten und besteht aus der Häufigkeit *occurences*, der Dauer *duration*, der Anzahl der Reaktivierungen des Fehlers *reactivations* und dem auszulösenden Wert des Aktivierungsmusters beziehungsweise dem optionalen Wert *value*, auf den das zu manipulierende Signal gesetzt werden soll. Zusätzlich besitzt *Fault_And_Activation_Data<T>* zum einen den *occurrence_ctr*, welcher angibt wie oft ein Fehler oder ein Aktivierungsmuster in der aktuellen *reactivation* bereits aufgetreten ist und zum anderen *time_in_ms*, welche angibt wann ein Fehler oder Aktivierungsmuster aktiviert werden soll. Jedes Aktivierungsmuster *Activation_Pattern* besteht aus der *aid* des Typs *Activation_ID* zur Definition des Aktivierungsmusters und den Aktivierungsdaten *activation_data* des schon zuvor beschrieben Typs *Fault_And_Activation_Data<T>*. Eine *Activation_ID* besteht aus dem *activation_point* des schon beschrieben Typs *Injection_Point*, zur Angabe des Aktivierungsmusters und der *Fault_ID* des *activated_fault*, zur Referenz des zu aktivierenden Fehlers.

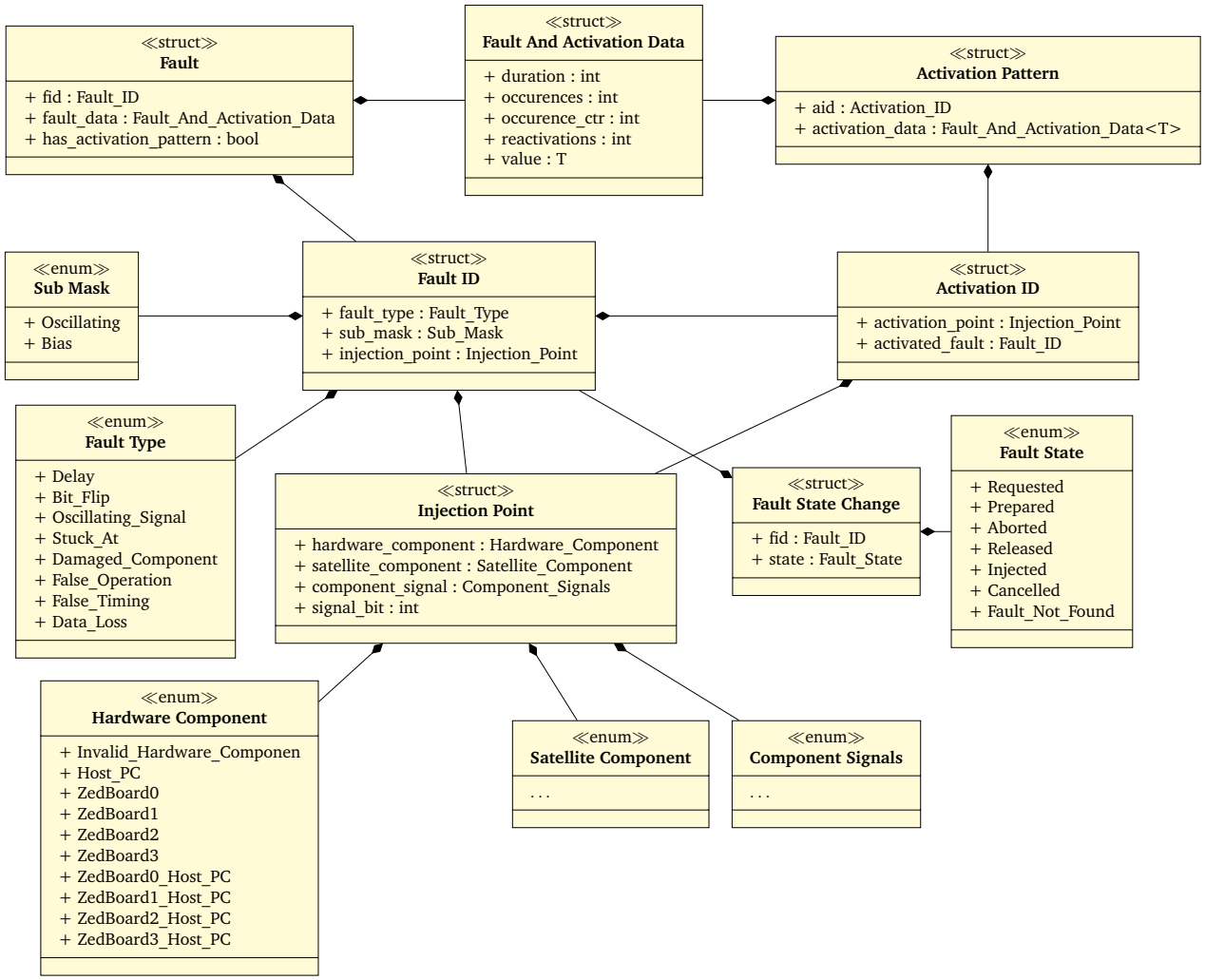


Abbildung 16.4.: Klassendiagramm Fault

Die Struktur *Fault_Injection_Command* (siehe Abbildung 16.5) zum Ausführen von Fehlerinjektionsbefehlen besteht aus den zwei Attributen *cid* des Typs *Command_ID* und einem Datenobjekt *data* des Typen *Parameter*. Die *Command_ID* gibt an, welcher Befehl ausgeführt werden soll. Es stehen die Befehle aus Tabelle 16.1 zur Verfügung. Das Datenobjekt *data* ist pro Fehlerinjektionsbefehl unterschiedlich und kann die Objekte *Fault*, *Activation_Pattern*, *Fault_State_Change*, *Mission_Phase*, *Activation_ID*, und *String* annehmen. Wann welches Objekt gewählt wird, ist auch in Tabelle 16.1 erkennbar. Die Strukturen *Fault*, *Activation_Pattern*, *Fault_ID* und *Activation_ID* wurden bereits erklärt und werden nicht genauer dargestellt. Das Datenobjekt *Fault_State_Change* besteht aus der Identifikation des Fehlers *fid* des Typs *Fault_ID* und dem Zustand des Fehlers *Fault_State*, der die in der Tabelle 16.2 beschriebenen Zustände annehmen kann. Die Struktur *Mission_Phase* kann als Wert verschiedene Missionsphasen annehmen.

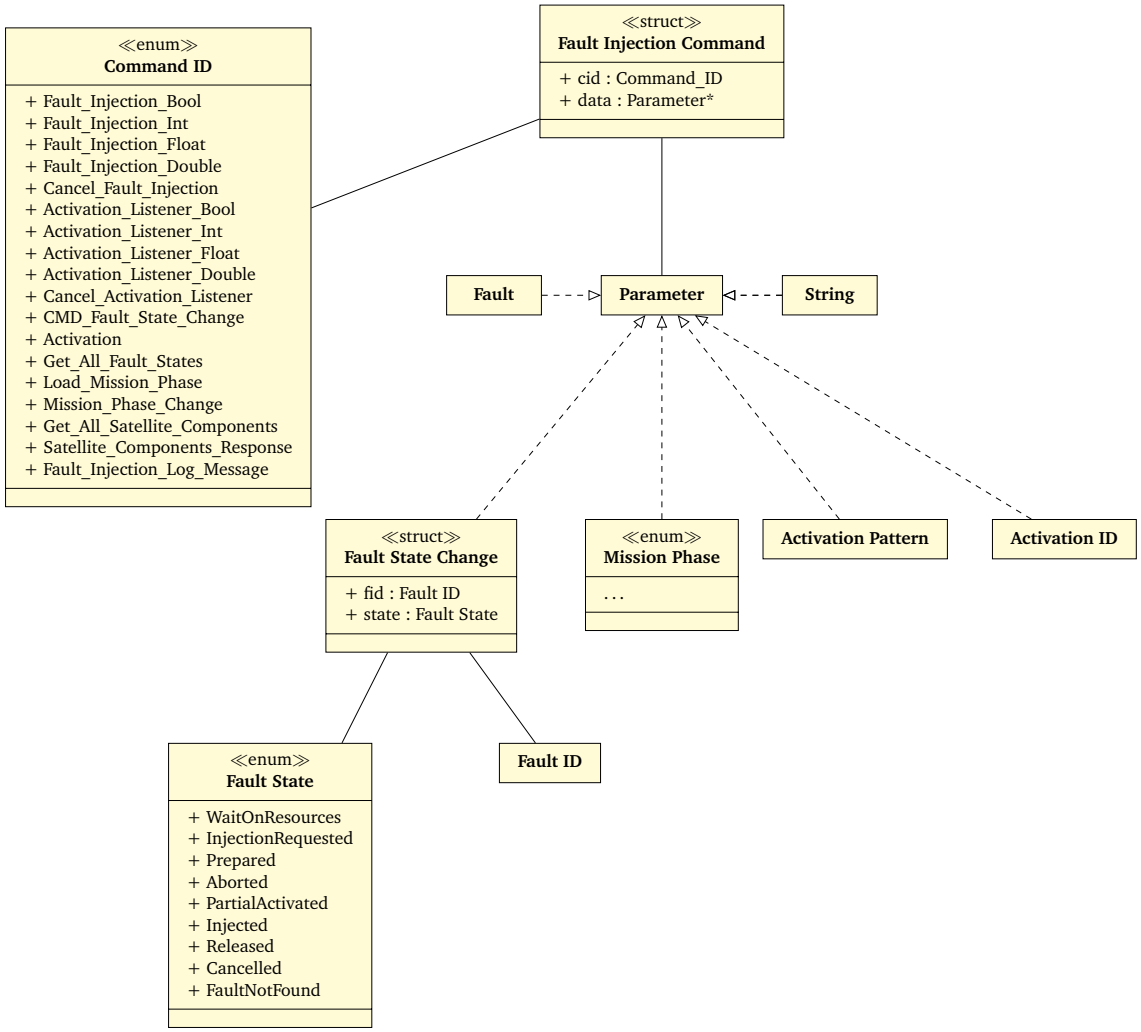


Abbildung 16.5.: Klassendiagramm Fault_Injection_Command

Die Klasse *Fault_Experiment* (siehe Abbildung 16.6) zum Speichern von Fehlerexperimenten besitzt die Attribute *experiment_counter*, zur Angabe der Anzahl der Fehler eines Fehlerexperiments, *experiment_id* zur Identifikation eines Fehlerexperiments und *completed* zur Überprüfung, wie weit das Fehlerexperiment injiziert wurde.

Fehlerinjektionsbefehl	Datenobjekt	Beschreibung
Fault_Injection_Bool	Fault<bool>	Befehl zum Ausführen einer boolischen Fehlerinjektion an das zu manipulierende Modul.
Fault_Injection_Int	Fault<int>	Befehl zum Ausführen einer ganzzahligen Fehlerinjektion an das zu manipulierende Modul.
Fault_Injection_Float	Fault<float>	Befehl zum Ausführen einer Floating Point Fehlerinjektion an das zu manipulierende Modul.
Fault_Injection_Double	Fault<double>	Befehl zum Ausführen einer Floating Point Fehlerinjektion an das zu manipulierende Modul.
Cancel_Fault_Injection	Fault_ID	Befehl zum Abbruch eines Fehlerinjektion an das zu injizierende Modul.
Activation_Listener_Bool	Activation_Pattern <bool>	Befehl zur Übertragung der Aktivierungsmuster zu einem Fehler an das zu aktivierende Modul.
Activation_Listener_Int	Activation_Pattern <int>	Befehl zur Übertragung der Aktivierungsmuster zu einem Fehler an das zu aktivierende Modul.
Activation_Listener_Float	Activation_Pattern <float>	Befehl zur Übertragung der Aktivierungsmuster zu einem Fehler an das zu aktivierende Modul.
Activation_Listener_Double	Activation_Pattern <double>	Befehl zur Übertragung der Aktivierungsmuster zu einem Fehler an das zu aktivierende Modul.
Cancel_Activation_Listener	Activation_ID	Befehl zum Abbruch eines Aktivierungsmuster an das zu aktivierende Modul.
CMD_Fault_State_Change	Fault_State_Change	Rückmeldung einer Zustandsänderung eines Fehlers
Get_All_Fault_States	-	Befehl zum Anfordern aller Fehlerzustände
Load_Mission_Phase	int	Befehl zum Setzen der Missionsphase
Mission_Phase_Change	int	Befehl zum Setzen der Missionsphase
Get_All_Satellite_Components	-	Befehl zum Anfordern aller Satellitensteuerungskomponenten.
Satellite_Components_Response	string	Übertragung der Komponenten der Satellitensteuerung
Fault_Injection_Log_Message	FI-Logmessage oder string	Logging-Message der Fehlerinjektions-Komponenten

Tabelle 16.1.: Beschreibung der Fehlerinjektionsbefehle

Fehlerzustand	Beschreibung
Requested	Der Fehler hat seine Ressourcen zugewiesen bekommen und wartet auf seine Rückmeldung der Vorbereitung.
Prepared	Der Fehler ist in dem zu manipulierenden Modul gespeichert und wartet auf seine Aktivierung.
Aborted	Der Fehler konnte aufgrund von Ressourcenmangel nicht in dem zu manipulierenden Modul gespeichert werden.
PartialActivated	Der Fehler wurde partiell aktiviert.
Injected	Der Fehler wurde injiziert.
Released	Die Ressourcen für den Fehler wurden freigegeben.
Cancelled	Der Fehler wurde abgebrochen. Die Ressourcen wurden freigegeben.
FaultNotFound	Der Fehler wurde nicht gefunden. Dies kann passieren, wenn eine Fehlerinjektion versucht wird abzubrechen, aber in dem Moment davor, die Fehlerinjektion beendet und die Ressourcen freigegeben wurden.

Tabelle 16.2.: Beschreibung der Fehlerzustände

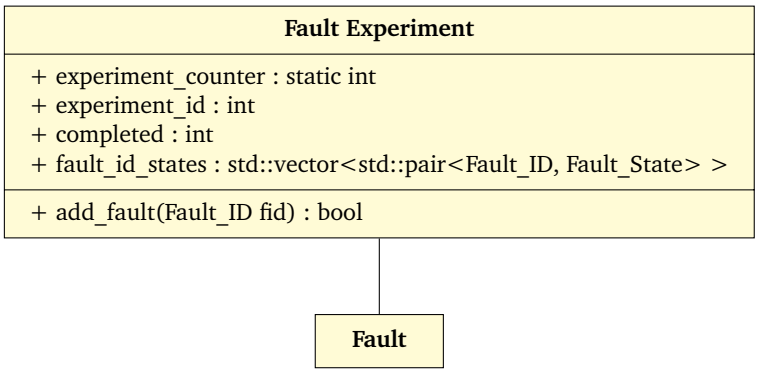


Abbildung 16.6.: Klassendiagramm Fault_Experiment

Die Klasse *FI_Components_MC* (siehe Abbildung 16.7) dient zur Speicherung der Fehlerinjektoren des Mikrocontrollers. Die Klasse ist erstellt worden, um die Injektoren in ein globales Objekt zu packen und somit viele lokale Zeiger auf die Injektoren zu vermeiden.

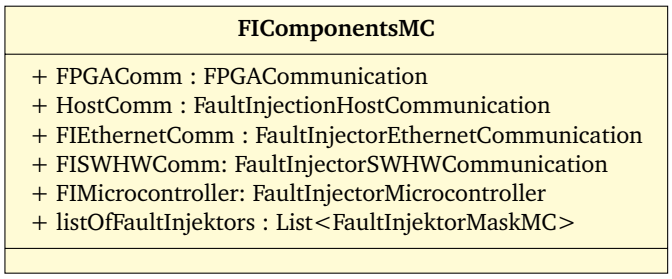


Abbildung 16.7.: Klassendiagramm FI_Components_MC

16.2.2. Fehlerinjektion auf dem FI-PC

In diesem Abschnitt wird der Modulentwurf der Fehlerinjektion auf dem FI-PC anhand einzelner Klassen erläutert.

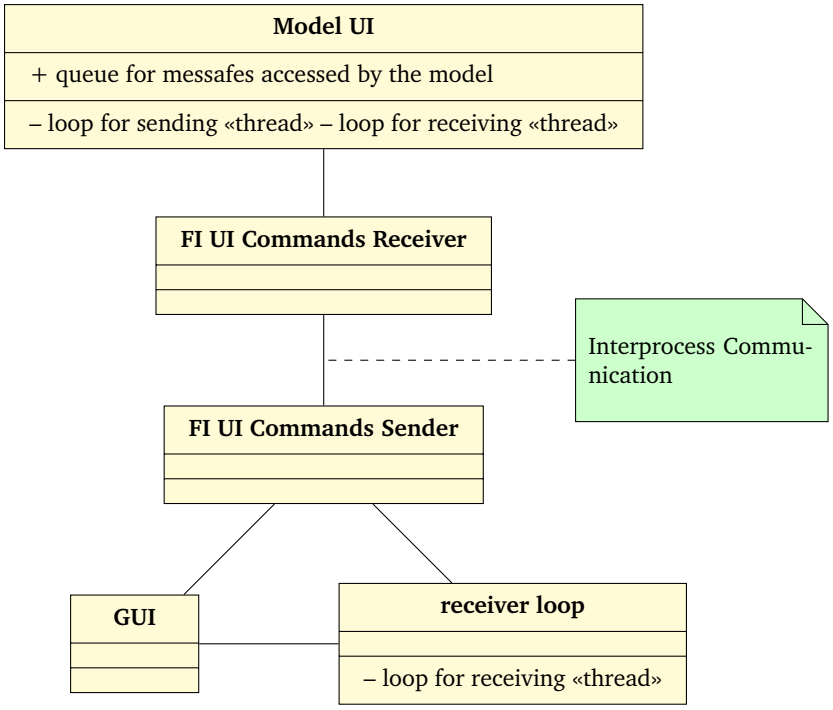


Abbildung 16.8.: Klassendiagramm der Kommunikation der UI-Befehle

Die Fehlerinjektion erhält über eine Interprozesskommunikation zwischen dem *FI_UI_Commands_Sender* (siehe Abbildung: D.24) und dem *FI_UI_Commands_Receiver* (siehe Abbildung: D.23), welche in Abbildung: 16.8 vereinfacht dargestellt wird, von einem Benutzer in der *GUI* erstellte Fehlerinjektionsbefehle. Über den selben Weg, nur in der umgekehrten Richtung, können dem Benutzer Logs der Fehlerinjektion und der Satellitensteuerung angezeigt werden.

Der *FI_UI_Commands_Receiver* leitet den Fehelerinjektionsbefehl an das interne *User_Interface* (siehe Abbildung: D.22) der Fehlerinjektion auf dem FI-PC weiter, welches den Fehlerinjektionsbefehl in der Funktion *receive_ui_command_packet* verarbeitet und an den *Fault_Injector_Controller* sendet.

Der *Fault_Injector_Controller* (siehe Abbildung: D.26) ist das Hauptmodul der Fehlerinjektion auf dem FI-PC. Der Controller ist für das Steuern der Fehlerinjektionen und dem Ausführen der Nutzeranfragen zuständig, arbeitet aber zu einem Großteil passiv, leitet Funktionsaufrufe zu dem Modul weiter, in dem diese ausgeführt werden sollen. Dazu speichert der Controller die Missionsphase in der Variablen *current_phase* ab, welche über die Methode *set_mission_phase* an die Satellitensteuerung übergeben werden kann und durch *update_mission_phase* verändert werden kann. Als zweite Aufgabe besitzt der Controller das Verwalten von Fehlerexperimenten. Dazu nimmt der Controller ein Fehlerexperiment vom Nutzer über die Methode *load_fault_experiment* entgegen und speichert es in der Variablen *fault_experiment* ab. Die darin enthaltenen Fehler werden an den Fehlerinjektor zum Injizieren weitergegeben. Neben den Fehlerexperimenten soll der Nutzer auch Fehler per Befehl über *inject_fault* injizieren können. Damit das Warpen des Simulators den Nutzer allerdings weder beim injizieren von einzelnen Fehlern, noch beim abarbeiten von Fehlerexperimenten stört, kann das Warpen durch die Methode *ui_set_fi_state* gestoppt und gestartet werden können. Dazu wird die Variable *fault-*

injection_state, die angibt, ob die Fehlerinjektion aktiviert ist und wann diese nicht durch das Warpen gestört werden soll. Die restlichen Methoden sind zum Übertragen der Kontrolldaten an das Monitoring da.

Das *Monitoring* (siehe Abbildung: D.25) hat die Aufgabe, dem *User_ Interface* Daten aus dem *Controller*, der *Simulator_Satellite_Channel_Component*, sowie dem *Log_Protocol* zur Verfügung zu stellen. Dafür werden die Daten über verschiedene *process*-Methoden aufbereitet. Die vom Monitoring aufbereiteten Daten beinhalten Informationen zu Steuer- und Umweltsignalen, wobei die manipulierten Daten hervorgehoben werden. Fehlerexperimente, Fehlerinjektionszustände sowie Umwelt- und Steuersignale können aus dem *Fault_Injektor_ Controller*, sowie dem Log ausgelesen werden.

Für den Vorgang der Fehlerinjektion ist der *Fehlerinjektor (Fault_Injector)* (siehe Abbildung: D.27) zuständig. Dafür sind die Fault_Library und die Satellite_Component_Library, sowie die Kommunikation zu den Injektionspunkten über den *sim_sat_channel* zu der Satellitensteuerung-Simulator-Kommunikation und den *sat_sim_channel* zu dem Zustand der Satellitensteuerung auf dem ZedBoard notwendig. Der Fault_Injector soll neben dem Injizieren auch einen Vektor *injected_faults* über die Zustände (siehe Tabellle 16.2) der zu injizierenden Fehler führen. Die Methoden *get_injected_faults*, *get_injectable_faults* und *get_satellite_components*, sowie der Änderungsmelder-Methoden *inform_monitoring_about_fault_state_change* sind für die Weitergabe der Fehlerzustände über den Controller an das Monitoring zuständig. Hierbei werden die injizierten Fehler, die Satellitensteuerungskomponenten und die möglichen Fehler übermittelt. Im folgenden wird zur Speicherung der injizierten Fehler und der Aktivierungsmuster von jeweils einem Vektor gesprochen, obwohl es jeweils vier Vektoren gibt. Dies wird durch Fault_And_Activation_Data bedingt und wird durch den einen Vektor abstrahiert. Die Methode *inject_fault* dient als Schnittstellenmethode zum Injizieren der Fehler vom Controller zum Fehlerinjektor. Wird *inject_fault* aufgerufen, wird zuerst überprüft, ob der übergebene Fehler und der übergebende Injektionspunkt existieren. Dafür werden die Methoden *verify_fault* und *verify_injection_point* verwendet. Ist dies jeweils der Fall, wird durch die Methode *compute_injection_point* überprüft ob ein Fehler mit der selben Fault_ID bereits vorbereitet oder injiziert wird, wenn dem so ist, wird die Fehlerinjektion des neuen Fehlers abgebrochen und wechselt in den Zustand *Aborted*. Sind die Ressourcen frei, wird ein Fehlerinjektionsbefehl an den Injektionspunkt gesendet.

Über die Methode *add_activation_pattern* werden die Aktivierungsmuster einem Vektor hinzugefügt, welcher die Aktivierungsmuster verwaltet. Nachdem die Aktivierungsmuster dem Vektor hinzugefügt wurden, werden sie in der Methode *adjust_injection_point* noch einmal dahingehend überprüft, ob der Injektionspunkt des Aktivierungsmuster zu dem des Fehlers passt und gegebenenfalls angepasst.

Wird ein Zustand eines Fehlers verändert, soll über die Methode *process_Fault_State_Change* dem Fehlerinjektor die Änderungen mitgeteilt werden und gegebenenfalls nach Freigebe von Ressourcen noch nicht vorbereitete Fehlerinjektionen vorbereiten. Bei diesem Methodenaufruf wird auch auch das Monitoring über die Änderung informiert werden.

Die letzte Methode *reset* ist für das Zurücksetzen des Fehlerinjektors, also dem Löschen aller injizierten Fehler, zuständig. Diese Methode wird bei dem Laden einer Missionphase verwendet.

Die *Fault_Library* (siehe Abbildung 16.9) dient als Speicherquelle aller injizierbaren Fehlern innerhalb des Vektors *fault_lib*.

Fault Library
+ fault_lib : std::vector<std::pair<Fault_Type,Injection_Point> >
+ FaultLibrary()

Abbildung 16.9.: Klassendiagramm Fault_Library

Die *Satellite_Component_Library* (siehe Abbildung 16.10) dient als Speicherquelle der verfügbaren Hardware und der Satellitensteuerungskomponenten, welche in den Vektoren *hardware_components* und *software_components* gespeichert werden.

Satellite Component Library
- hardware_components : std::vector<std::pair<std::string,Satellite_Component> > - satellite_components : std::vector<std::pair<std::string,Hardware_Component> >
+ Satellite_Component_Library()

Abbildung 16.10.: Klassendiagramm Satellite_Component_Library

Zur Kommunikation mit der Simulation und der Satellitensteuerung existiert die *Simulator_Satellite_Channel_Component* (siehe Abbildung: D.28), welche über die *UDP_Satellite_Control_Communication* (siehe Abbildung: D.29) an die Satellitensteuerung und über die *UDP_Simulator_Communication* (siehe Abbildung: D.30) an die Simulation angebunden ist. So können folgende Module mit einander kommunizieren:

- Der *Fault_Injection_Controller*
- Der *Fault_Injector*
- Der *Simulator_Satellite_Channel_Component*
- Die *UDP_Simulator_Communication*
- Die *UDP_Satellite_Control_Comunication*

Das Modul *Simulator_Satellite_Channel_Component* hat die Aufgabe, Umweltsignale vom Simulator an die Satellitensteuerung und Steuersignale von der Satellitensteuerung an den Simulator weiterzuleiten und gegebenenfalls zu manipulieren. Zu diesem Zweck beinhaltet der Channel ein Modul, das die Umwelt- und Steuerungssignale manipulieren kann. Dies geschieht, wenn vom *Fault_Injector* die Anweisung dazu kommt. Weiter stellt der Channel die Methoden zum Abrufen von Daten durch das Monitoring zur Verfügung.

Ein Beispielhafter Ablauf einer Fehlerinjektion, beginnend beim *User_Interface* der Fehlerinjektion auf dem FI-PC, in ein Umweltsignal der *Simulator_Satelliten_Channel_Component* sowie das Hinzufügen eines Aktivierungsmuster werden in den folgenden Sequenzdiagrammen (Abbildung: 16.11 und Abbildung: 16.12) abgebildet.

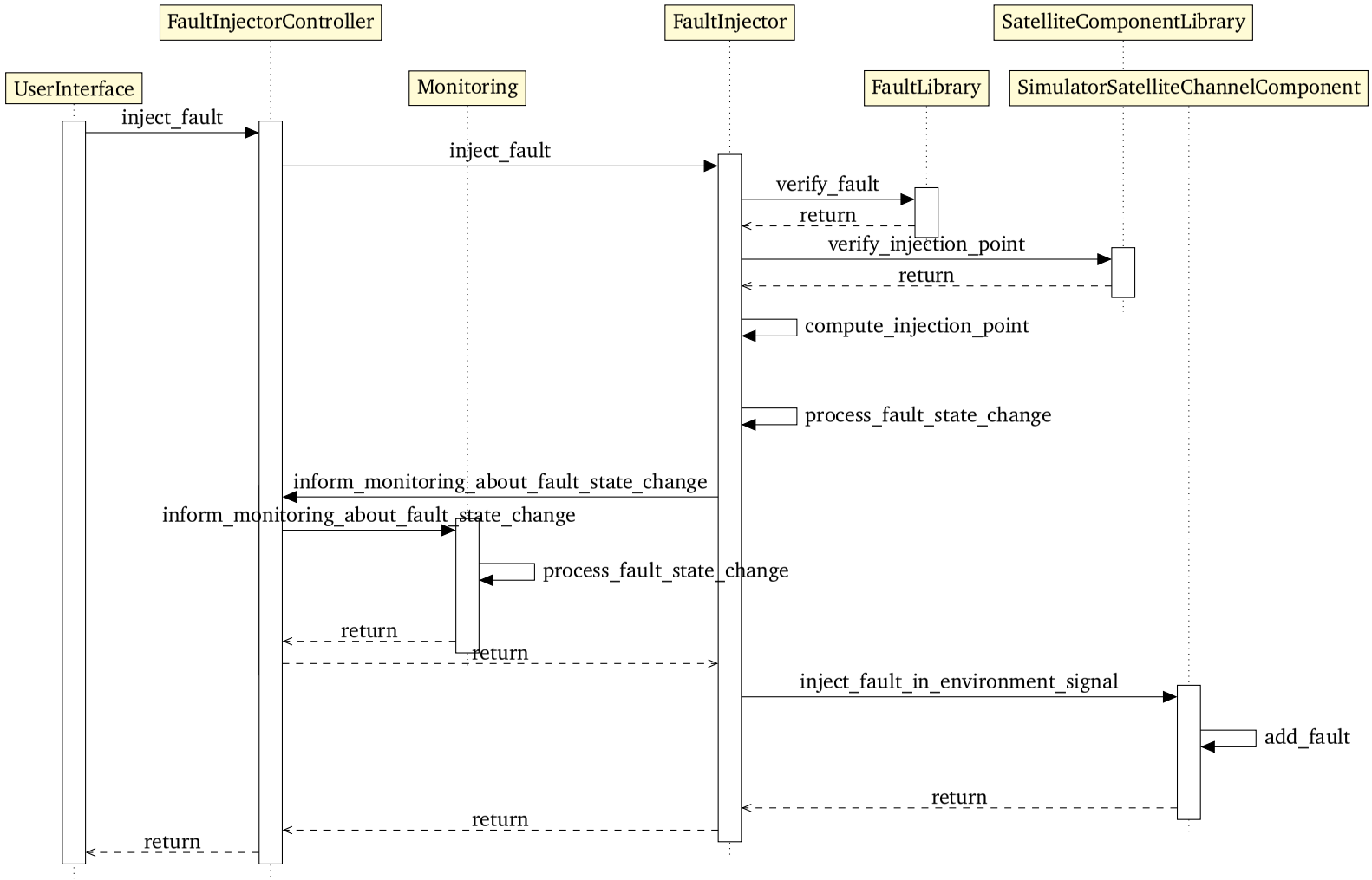


Abbildung 16.11.: Sequenzdiagramm: Beispielhafte Fehlerinjektion auf dem FI-PC in ein Umweltsignal der Simulator_Satelliten_Channel_Component

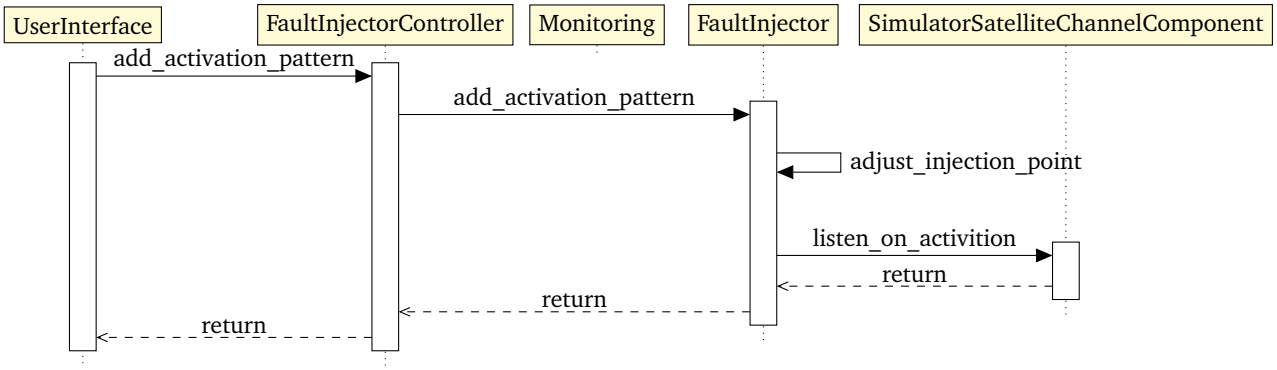


Abbildung 16.12.: Sequenzdiagramm: Beispielhaftes Hinzufügen eines Aktivierungsmusters in der Simulator_Satelliten_Channel_Component

16.2.3. Fehlerinjektion auf dem Mikrocontroller

In diesem Abschnitt wird der Modulentwurf der Fehlerinjektion auf dem Mikrocontroller anhand eines Klassendiagramms (Abbildung: 16.13) erläutert.

Die Fehlerinjektion auf dem Mikrocontroller basiert auf den zwei Klassen *Communication* und *FaultInjectionMaskMC*.

In der Klasse *Communication* werden die Attribute und Methoden definiert, welche zur Kommunikation des Mikrocontrollers mit dem FI-PC oder dem FPGA benötigt werden. Dabei verarbeiten die Kommunikationsklassen die über einen bestimmten *port* ankommenden Nachrichten. Die *FaultInjectionCommands* werden in den Warteschlangen *fiCommandQueue* und *respondQueue* zwischengespeichert und mit den Methoden *forwardFICommand*, *forwardPartialActivation* und *forwardFaultStateChange* weitergeleitet. Nachrichten die vom Typ *PartialActivation* sind, werden in der *partialActivationQueue* zwischengespeichert. Neben der Weiterleitung von Nachrichten besitzen Kommunikationsklassen auch eine Methode *acknowledge*, welche das Empfangen von Nachrichten bestätigt, und eine Methode *reset*, welche beim Zurücksetzen der Fehlerinjektion ausgeführt wird und die Warteschlangen leert.

Die Klasse *Communication* vererbt ihre Eigenschaften an die Klasse *FaultInjectionHostCommunication*, welche die Kommunikation des Mikrocontrollers mit der Fehlerinjektionskomponente auf dem FI-PC übernimmt und an die Klasse *FPGACommunication*, welche die Kommunikation des Mikrocontrollers mit dem FPGA von der Seite des Mikrocontrollers übernimmt. Zusätzlich fängt die *FPGACommunication* die *PartialActivations* ab, welche Teile von Aktivierungsmustern für Fehler sind, die auf dem FPGA injiziert werden sollen.

In der Klasse *FaultInjectionMaskMC* werden die Attribute und Methoden definiert, welche zur Fehlerinjektion auf dem Mikrocontroller benötigt werden. Mit den Methoden *computeFault*, *activateFault*, *injectFault* und *sendInjectionReport* wird eine Fehlerinjektion vorbereitet, (teilweise) aktiviert, durchgeführt und eine Rückmeldung der Fehlerinjektion verschickt. Dazu können in *ListOfFaultStates*, *ListOfActivationStates* und *ListOfActivationListeners* alle für den jeweiligen Fehlerinjektor relevanten Daten zwischengespeichert werden. Weiter kann mit der Methode *listenOnActivation* abgefangen werden, welche Daten zur Aktivierung von Fehlerinjektionen benötigt werden. Zudem wird mit der Methode *clock* die Laufzeit gemessen, um zeitbedingte

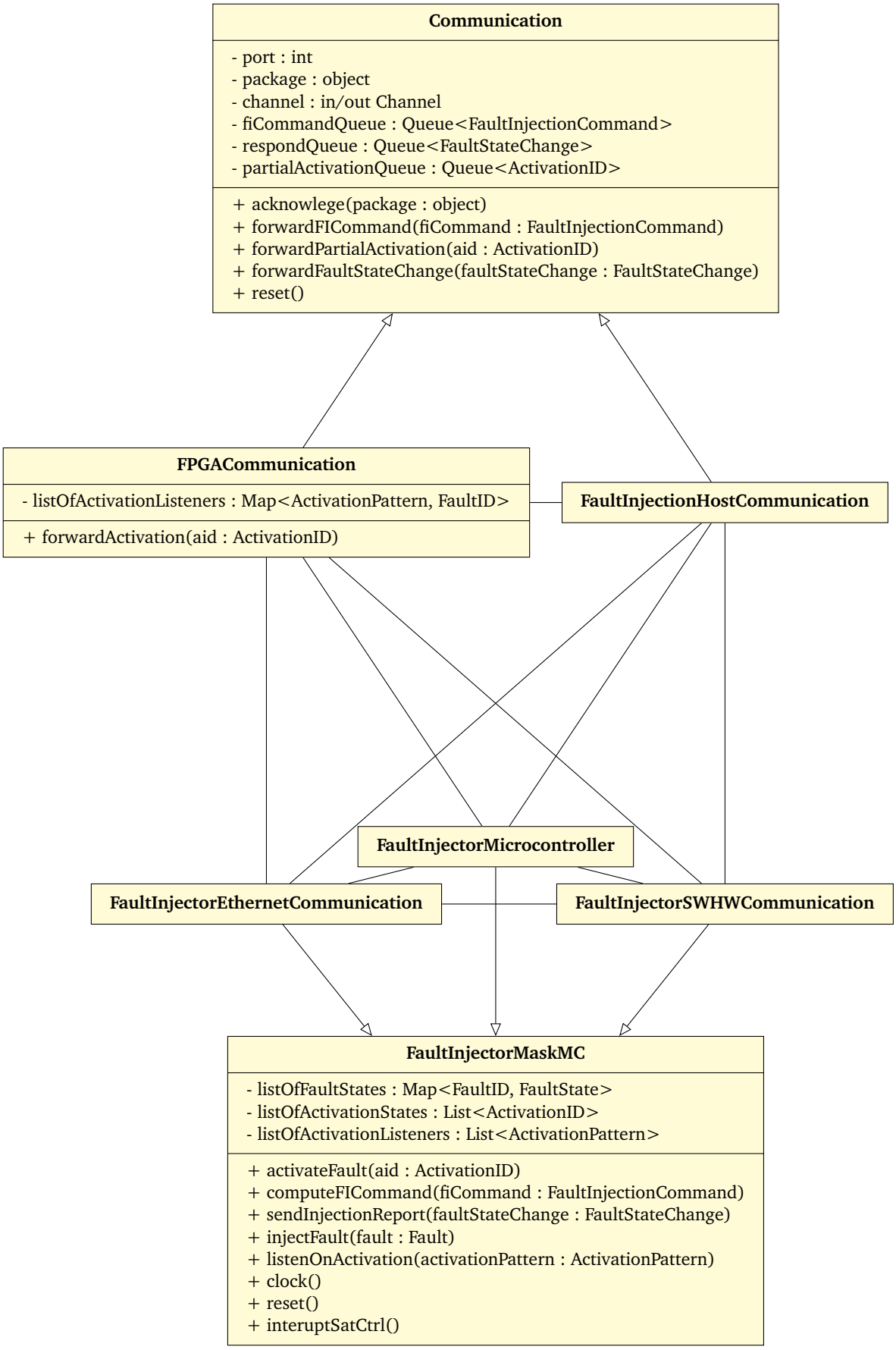


Abbildung 16.13.: Klassendiagramm der Fehlerinjektion auf dem Mikrocontroller

Fehlerinjektionen zu ermöglichen. Außerdem können mit der Methode *reset* die Fehlerinjektoren auf ihren Ausgangszustand zurückgesetzt und mit der Methode *interruptSatCtrl* die Satellitensteuerung kurzzeitig unterbrochen werden, damit sichergestellt werden kann, dass die Fehlerinjektion auf dem Mikrocontroller, nach vollständiger Aktivierung eines Fehlers, sofort durchgeführt wird.

Von der Klasse *FaultInjectionMaskMC* werden die Klassen *FaultInjectorEthernetCommunication*, *FaultInjectorMicrocontroller* und *FaultInjectorSWHWCommunication* abgeleitet. Der *FaultInjectorEthernetCommunication* ist für die Fehlerinjektion in die Ethernet-Kommunikation zwischen der Satellitensteuerung und dem Simulator oder zwischen den Mikrocontrollern zuständig. Der *FaultInjectorMicrocontroller* ist für die Fehlerinjektion direkt auf dem Mikrocontroller zuständig. Der *FaultInjectorSWHWCommunication* ist für die Fehlerinjektion in der Software-Hardware-Kommunikation zuständig.

16.2.4. Fehlerinjektion auf dem FPGA

In diesem Abschnitt wird zunächst eine Übersicht über die Module gegeben um diese im Anschluss einzeln zu beschreiben. Zum Schluss wird auf das gleichzeitige Injizieren von Fehlern eingegangen.

Auf dem FPGA gibt es vier Hauptkomponenten, die modularisiert werden müssen. Das erste Modul (siehe Abbildung 16.14) ist das Modul *Microcontroller-FPGA-Communication* zur Annahme und Verteilung der Fehlerinjektionen und Aktivierungsmuster vom Mikrocontroller, sowie zur Rückgabe von Fehlerzuständen als auch zur Weitergabe von Aktivierungen von Aktivierungsmustern auf dem FPGA für Fehler außerhalb des FPGAs. Sie kommuniziert mit den *FPGAStateFaultInjector*, sowie den Kommunikationsmodulen. Die Kommunikationsmodule *HardwareSoftwareCommunicationFaultInjector* und *SoftwareHardwareCommunicationFaultInjector* sind aus Platzgründen durch HWSWFaultInjector und SWHWFaultInjector abgekürtzt worden. Das Modul *FPGAStateFaultInjector* dient im größeren Sinne nur zur Weiterleitung der Fehlerzustände zu den Fehlerinjektionsschnittstellen in den Satellitensteuerungskomponenten und den Rückgaben der Fehlerinjektionsschnittstelle zum Mikrocontroller. Dabei implementieren die Fehlerinjektionsschnittstellen in den Satellitensteuerungskomponenten und die Kommunikationsmodule das Interface *FaultInjectorMask*.

Da die Satellitensteuerungskomponenten mit den Kommunikationsmodulen reden, aber die Fehlerinjektionsschnittstelle nur indirekt in Verbindung mit den Kommunikationsmodulen steht, sind die Pfeile zwischen den Satellitensteuerungskomponenten und den Kommunikationsmodulen schwächer ausgeprägt. Die Kommunikationsmodule dienen zur Manipulation der Mikrocontroller-FPGA-Kommunikation. Zusätzlich zu den Modulen wird für den *FPGAStateFaultInjector* und den Kommunikationsmodulen Uhren benutzt, die für das Aktivieren von zeitbasierten Aktivierungsmustern zuständig sind.

Das Modul *Microcontroller-FPGA-Communication* (siehe Abbildung 16.15) dient zur Annahme und Verteilung der Fehlerinjektionen und Aktivierungsmuster vom Mikrocontroller, sowie zur Rückgabe von Fehlerzuständen als auch zur Weitergabe von Aktivierungen von Aktivierungsmustern auf dem FPGA für Fehler außerhalb des FPGAs. Für die Weiterleitung der Fehlerinjektionssbefehle vom Mikrocontroller an die jeweiligen Fehlerinjektor-Module des FPGAs soll die Methode *forwardFaultInjectionCommand* dienen. Die Antwort wird über die Methode *pro-*

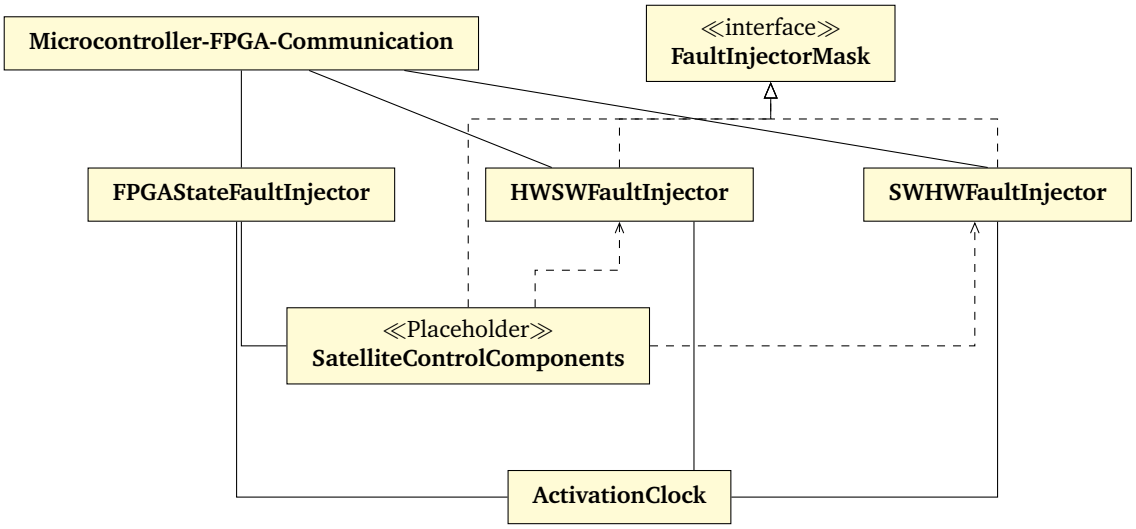


Abbildung 16.14.: Überblick über die Module auf dem FPGA

cessFaultStateChange abgewickelt. Da der Mikrocontroller langsamer arbeitet als das FPGA und das FPGA-Microcontroller-Communication-Modul von mehreren Fehlerinjektor-Modulen Fehlerzustände bekommt, braucht das Communication-Modul eine Warteschlange *faultStateChangeQueue* für Fehlerzustände, sowie ein Signal *faultStateChangeProcess* zur Verwaltung der Warteschlange. Für die Übertragung von aktivierten Aktivierungsmustern wird dasselbe getan: Für die Kommunikation von Aktivierungen vom Mikrocontroller wird die Methode *forwardPartialActivationMC* genutzt, während für die Kommunikation vom FPGA zum Mikrocontroller die Methode *forwardPartialActivationFPGA*, die Warteschlange *partialActivationDataQueue* und das Verwaltungssignal *partialActivationProcess* benutzt werden. Im Spezialfall, dass das Modul aufgrund von fehlendem Speicherplatz keine neuen Fehlerzustände oder Aktivierungen entgegen nehmen kann, soll durch die *Queues* und *Process*-Signale auch das erneute Laden der Daten ermöglicht werden. Die Methode *reset* ist für das Zurücksetzen des Moduls zuständig. Da die Methoden von Modulen auf dem FPGA nicht einfach aufgerufen werden können, sondern durch Eingabe- und Ausgabesignale angesprochen werden, müssen noch die folgenden Channels / Signale benutzt werden:

Eingabe-Signale	**Beschreibung**
faultInjectionCommand	Signal zur Übertragung der Fehlerinjektionsbefehle vom Mikrocontroller zu den Fehlerinjektoren auf dem FPGA.
faultStateChanges	Signal zur Übertragung der Fehlerzustandsänderungen von den Fehlerinjektoren zu dem Mikrocontroller.
partialActivationMC	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Mikrocontroller zu den Fehlerinjektoren auf dem FPGA.
partialActivationFPGA	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Fehlerinjektoren zu dem Mikrocontroller.
ReadEnable	Signal zur Angabe, dass der Mikrocontroller eine Nachricht geschrieben hat.
WriteEnable	Signal zur Angabe, ob der Mikrocontroller die geschriebene Nachricht vom FPGA gelesen hat.
reset	Signal zum Resetten des Moduls.

Ausgabe-Signale	Beschreibung
faultInjectionCommand-Output	Signal zur Übertragung der Fehlerinjektionsbefehle vom Mikrocontroller zu den Fehlerinjektoren auf dem FPGA.
faultStateChanges-Output	Signal zur Übertragung der Fehlerzustandsänderungen von den Fehlerinjektoren zu dem Mikrocontroller.
partialActivationMC-Forward	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Mikrocontroller zu den Fehlerinjektoren auf dem FPGA.
partialActivationFPGA-Forward	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Fehlerinjektoren zu dem Mikrocontroller.
ReadEnable	Signal zur Angabe, dass das FPGA eine Nachricht geschrieben hat.

Aus Platzgründen wurden die Signale bei den Klassendiagrammen weggelassen.

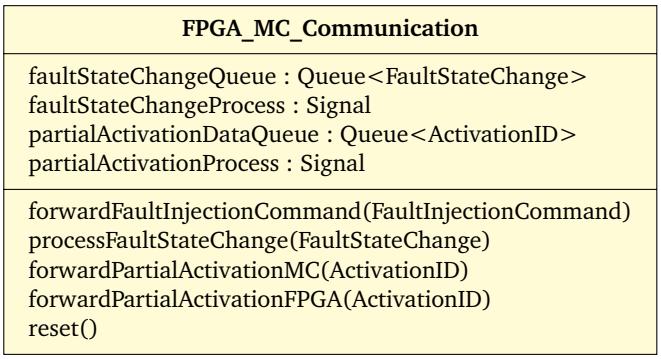


Abbildung 16.15.: Klassendiagramm FaultInjector

Der *FPGAStateFaultInjector* (siehe Abbildung 16.16) dient zum Weiterleiten der Fehlerinjektionen und Aktivierungen an die spezifischen Module der Satellitensteuerung und zum Weiterleiten der Rückmeldungen an das *FPGA_MC_Communication*-Modul. Dieses Modul verwendet die gleichen Daten und Methoden wie das *FPGA_MC_Communication*-Modul und kann als Multiplexer angesehen werden.

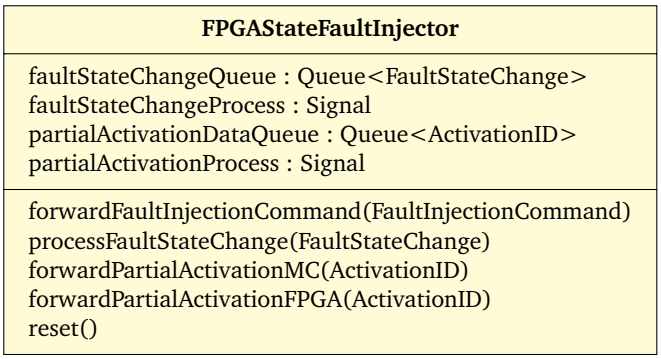


Abbildung 16.16.: Klassendiagramm FPGAStateFaultInjector

Bevor auf das Modul der Hardware-Software-Kommunikation eingegangen wird, wird die *FaultInjectorMask* (siehe Abbildung 16.17) als Fehlerinjektionsschnittstelle vorgestellt. Die FaultInjectorMask verhält sich ähnlich zu der *FaultInjectorMask* auf dem Mikrocontroller, welche aufgrund der Hardwarekommunikation allerdings ein paar Unterschiede aufweist. Die *FaultInjectorMask* beschreibt die Fehlerinjektionsschnittstelle, die neben den Kommunikations-Fehlerinjektoren benutzt wird, um jede Satellitensteuerungskomponente zu manipulieren. Die

Schnittstelle liegt selbst in der Satellitensteuerungskomponente drin, um auch den inneren Zustand einfach manipulieren zu können. Das Signal *ListOfFaultInjectionStates* speichert die Liste der zu injizierenden Fehler, die auf ihre Aktivierung warten. Mit der Methode *computeFICommand* kann ein Fehler der Liste hinzugefügt werden. Falls eine Zustandsänderung von einem Fehler geschieht, beispielsweise wenn ein Fehler injiziert wird, sollen zur Benachrichtigung des Nutzers die Änderung zurückgegeben werden. Dazu wird das Signal *FaultStateChange* verwendet. Da auch gleichzeitig zwei Zustandsänderungen passieren können, muss durch das Signal *faultStateChangeProcess* die Benachrichtigung aller Änderungen gewährleistet werden. Wie bereits bei dem *Microcontroller-FPGA-Communication*-Modul soll im Spezialfall durch die *Queues* und *Process*-Signale das erneute Laden der Daten ermöglicht werden. Der Fortschritt der Aktivierung eines Fehlers in das Modul wird in der Liste für Aktivierungsmuster *ListOfActivationStates* gespeichert. Durch die Methode *activateFault* kann eine Aktivierung ausgelöst werden. Wenn das gesamte Aktivierungsmuster des Fehlers aktiviert wurde, wird der Fehler injiziert. Soll ein Fehler in einem anderen Modul durch ein Signal dieses Moduls aktiviert werden, so werden die Aktivierungen in dem Signal *ListOfActivationListeners* gespeichert und bei Aktivierung über die Methode *actualizeListOfActivationListeners* und das Signal *partialActivation* ausgegeben. Da auch hier wieder zwei Aktivierungen gleichzeitig passieren können, muss durch das Signal *partialActivationProcess* die Übertragung aller Aktivierungen gewährleistet werden. Auch über die Methode *computeFICommand* kann ein neues Aktivierungsmuster abgespeichert werden. Die Methode *reset* wird zum Resetten des Moduls gebraucht. Die Eingabe- und Ausgabesignale sind:

Eingabe-Signale	**Beschreibung**
faultInjectionCommand	Signal zur Übertragung der Fehlerinjektionsbefehle.
partialActivationIn	Signal zur Übertragung der aktivierten Aktivierungsmustern von Fehlern dieses Moduls.
reset	Signal zum Resetten des Moduls.

Ausgabe-Signale	**Beschreibung**
faultStateChange	Signal zur Übertragung der Fehlerzustände.
partialActivationOut	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Fehlern in anderen Modulen.

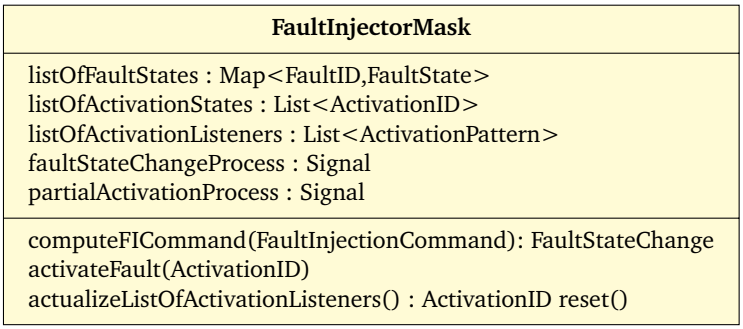


Abbildung 16.17.: Klassendiagramm FaultInjectorMask

Beispielhaft ist die Anwendung der *FaultInjectorMask* bei dem *HardwareSoftwareCommunicationFaultInjector*-Modul zu sehen.

Der *HardwareSoftwareCommunicationFaultInjector* (siehe Abbildung 16.18) dient zur Übertragung und Manipulation der Hardware-Software-Signale. Die ankommenden Kommunikationssignale der Satellitensteuerung auf dem FPGA werden über die Methode *processHardwareSoftwareData* angenommen, gegebenenfalls manipuliert und an den Mikrocontroller weitergegeben. Die Methode *computeFICommand* nimmt einen Fehlerinjektionsbefehl entgegen und speichert den in der *ListOfFaultInjectionStates* Liste ab, sodass die processHardwareSoftwareData Methode den Fehler injizieren kann, sobald der Fehler aktiviert wurde. Aktivierung und partielle Aktivierungen von Kommunikationsfehlern werden in der Liste *ListOfActivationStates* gespeichert. Externe Aktivierungen eines Kommunikationsfehlers können über die Methode *activateFault* übergeben werden. Neben den Kommunikationsfehlern speichert der Fehlerinjektor Aktivierungen von Fehlern aus anderen Modulen in der Liste *ListOfActivationListeners* und gibt Aktivierungensignale über die Methode *actualizeListOfActivationListeners* aus, sobald diese ausgelöst werden. Da es wieder mehrere Aktivierungen zur selben Zeit geben kann, wird wieder ein *partialActivationProcess* gebraucht. Gleiches gilt auch für die Rückgabe der Fehlerzustände. Die Methode *reset* wird wieder zum Resetten des Moduls gebraucht. Die Eingabe- und Ausgabesignale sind:

Eingabe-Signale	**Beschreibung**
HardwareSoftware-Signal	Signal zur Übertragung und Manipulation der Hardware-Software-Signale.
faultInjectionCommand	Signal zur Übertragung der Fehlerinjektionsbefehle.
partialActivationIn	Signal zur Übertragung der aktivierten Aktivierungsmustern von Kommunikationsfehlern.
reset	Signal zum Resetten des Moduls.

Ausgabe-Signale	**Beschreibung**
manipulatedHardware-SoftwareSignal	Signal zur Weitergabe der Manipulierten Hardware-Software-Signale an den Mikrocontroller.
faultStateChange	Signal zur Übertragung der Fehlerzustände.
partialActivationOut	Signal zur Übertragung der aktivierten Aktivierungsmustern von den Fehlern in anderen Modulen.

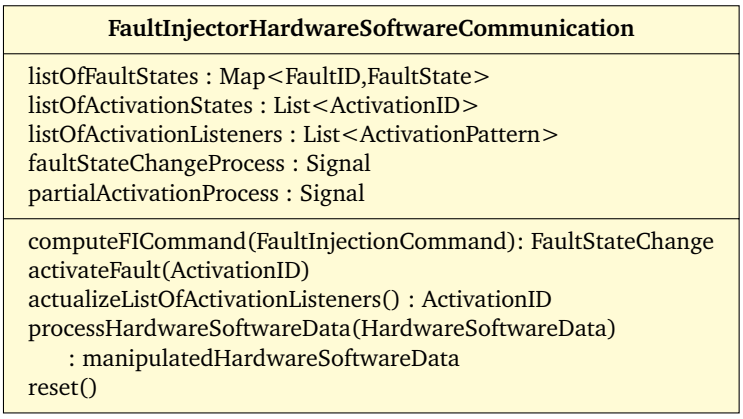


Abbildung 16.18.: Klassendiagramm HardwareSoftwareCommunicationFaultInjector

Der *SoftwareHardwareCommunicationFaultInjector* (siehe Abbildung 16.19) verhält sich ähnlich zum HardwareSoftwareCommunicationFaultInjector.

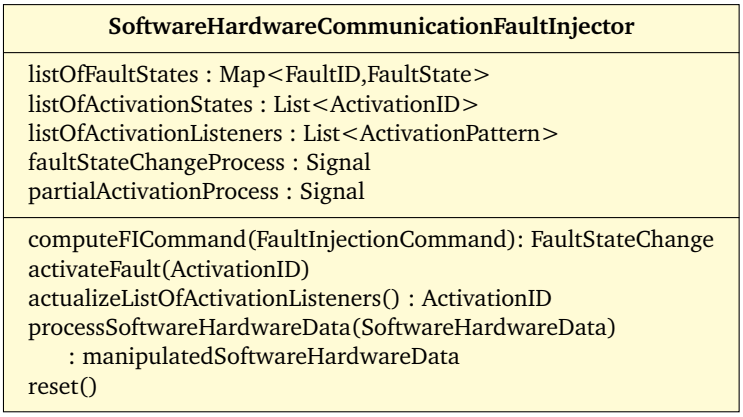


Abbildung 16.19.: Klassendiagramm SoftwareHardwareCommunicationFaultInjector

Neben den vier Hauptkomponenten, die bereits in dem Systementwurf 16.3 beschrieben wurden, stellt sich die Frage, ob zum Verteilen der Aktivierungen der im Diagramm enthaltenen *Activation Distributer* als Modul verwendet wird oder die Aktivierungen nur durch Verschaltungen verteilt werden. Während das Benutzen eines Moduls die Anzahl an Leitungen reduziert, da man den Activation Distributer als Multiplexer vorstellen kann, ergibt sich der Nachteil, dass das gleichzeitige Ansteuern des Multiplexers nur mit Feedback-Signalen sinnvoll modellieren lässt. Somit würde der Multiplexer die Kommunikation verzögern können, ohne einen wertvollen Vorteil erbringen zu können. Eine Verzögerung ist insbesondere dann schlecht, wenn zwei Fehler gleichzeitig injiziert werden sollen, aber eine Aktivierung am Multiplexer nicht durchgelassen wird. Daher wird kein Multiplexer verwendet.

Als weitere Module für das Aktivieren von zeitbasierten Aktivierungsmustern müssen Uhren zur Verfügung gestellt werden. Dafür wird das Modul *ActivationClock* (siehe Abbildung 16.20) eingeführt. Jeweils pro Fehlerinjektor wird eine Uhr zur Verfügung gestellt. Die ActivationClock soll zum Speichern mehrerer Timer und deren zugehörigen Fehler die Liste *listOfTimingSignals* besitzen, die mit der Methode *addTimingSignal* erweitert werden kann. Wird auf zu viele Timer gewartet, gibt die Methode *false* zurück. Der *InjektionsPoint* der *FaultID* muss aufgrund der direkten Anbindung an das injizierende Modul nicht vollständig angegeben werden. Bei den Uhren der Kommunikatiosmodule reicht sogar, die Angabe des Signals aus. Die Methode *timeout* gibt nach Erreichen eines Timers die *FaultID* des zugehörigen Fehlers zurück. Die Methode *reset* löscht alle Timer. Die Eingabe- und Ausgabesignale sind:

Eingabe-Signale	**Beschreibung**
addTimer	Signal zur Benachrichtigung, dass ein Timer gesendet wird.
timerInput	Signal zur Übertragung der Timer.
fidInput	Signal zur Übertragung der *FaultID*.
reset	Signal zum Resetten des Moduls.

Ausgabe-Signale	**Beschreibung**
timeout	Signal zur Benachrichtigung eines Timeouts.
fidOutput	Signal zur Ausgabe der *FaultID*.
writeSuccessful	Signal zur Benachrichtigung, ob ein Timer hinzugefügt wurde.

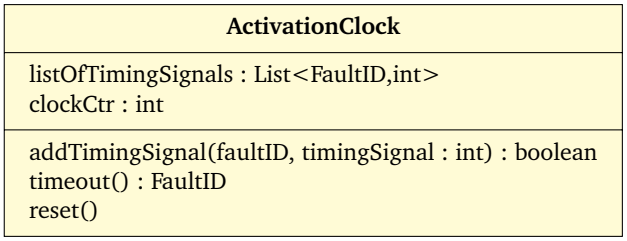


Abbildung 16.20.: Klassendiagramm ActivationClock

Aufgrund des bisherigen Designs der Module ergibt sich noch eine Anmerkung zu der gleichzeitigen Fehlerinjektion. Wenn man Fehler mit Aktivierungsmustern von anderen Fehlern in anderen Modulen abhängig macht, entsteht durch die unterschiedlichen Pfadlängen auch unterschiedliche Zeiten der Injektion. Es ist kein extra Synchronisierungs-Modul beschrieben worden, die vor der Fehlerinjektion überprüft, ob ein anderer Fehler synchron injiziert werden soll. Ebenso hätte ein solche Modul das Problem, dass andere Aktivierungsmuster des Fehlers verzögert würden. Zum Beispiel soll ein Fehler nach einem Erreichen eines bestimmten Zustands gemeinsam mit einem anderen Fehler injiziert werden. Sobald der Zustand erreicht ist, müsste das Synchronisierungs-Modul benachrichtigt werden, die beide Fehler zeitgleich injiziert. Durch das zusätzliche Senden an das Synchronisierungs-Modul würde die Injektion mindestens 1 Takt nach dem Erreichen den Zustandes passieren, sodass beide Fehler verzögert werden. Da auch das kein gewünschtes Design ist, wurde das Modul weggelassen. Im Schluss bedeutet das, dass gleichzeitige Fehlerinjektionen oft nicht gleichzeitig, sondern mit ein paar Takten Verzögerung injiziert werden. Der Nutzer sollte daher die gleichzeitigen Fehlerinjektionen mit Vorsicht benutzen. Dennoch stellt es für den Nutzer keinen großen Nachteil dar, da man immer noch davon ausgehen kann, dass zwei gleichzeitig aktivierte Fehler in *demselben* Modul auch gleichzeitig injiziert werden. Zudem kann man durch Ändern der Aktivierungsmuster gegebenenfalls das gleiche Verhalten erzeugen wie gleichzeitig aktivierte Fehler. Ein Beispiel für das Umbrechen der Fehler wäre der folgende Fall: Ein Fehler in einem bestimmten Modul 1 soll gleichzeitig mit einem anderen Fehler in einem anderen Modul 2 aktiviert werden, sobald der Zustand des Moduls 2 ein spezifischen Wert besitzt. Der Zustand des Moduls erzeugt ein *spezifisches* Output Signal, dass Input des Moduls 1 ist. Genau aus diesem Grund wollte der Nutzer auch, dass die Fehler gleichzeitig injiziert werden. Anstatt den Fehler in Modul 1 abhängig von den Zustand des Moduls 2 oder von dem zweiten Fehler zu machen, kann der Fehler in dem Modul 1 von den Input Signal abhängig gemacht werden, sodass hier das gewünschte Verhalten der gleichzeitigen Injektion eintritt.

16.2.5. Modulentwurf des Kommunikationsprotokolls der Fehlerinjektion auf dem FI-PC und dem Mikrocontroller

Das Protokoll zwischen der Fehlerinjektion auf dem FI-PC und der Fehlerinjektion auf dem Mikrocontroller hat die Aufgabe die Fehlerinjektionsbefehle aus Tabelle 16.1 zu übersenden. Da das Protokoll über UDP gesendet werden soll, wird einen Paritätsbit *Check* zur Prüfsummenbildung verwendet. Die Fehlerinjektionsbefehle werden über die *CommandID CID* und das Datenobjekt *Data* versendet.

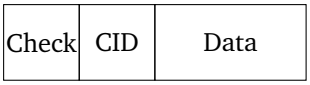


Abbildung 16.21.: Aufbau des Fehlerinjektionsprotokolls

16.2.6. Software-Hardware- und Hardware-Software-Protokoll

Das Software-Hardware-Protokoll und Hardware-Software-Protokoll auf dem ZedBoard haben die Aufgabe, jeweils einen Teil der Fehlerinjektionsbefehle aus Tabelle 16.1 zu übersenden. Das Software-Hardware-Protokoll (siehe Abbildung 16.22) soll die Befehle *FaultInjection, FaultInjectionActivations, CancelFaultInjection, CancelActivationPattern, LoadMissionPhase, GetAllFaultStates* und *PartialActivation* übertragen. Das Protokollübertragungsbit *RE (ReadEnable)* wird gebraucht, damit das FPGA während des Schreibens eines neues Befehles in die Register nicht reagiert und das Protokollübertragungsbit *WE (WriteEnable)* bestätigt den Erhalt einer Nachricht vom FPGA , sodass die Hardware-Software-Kommunikationsregister wieder neu beschrieben werden können. Der Typ des Fehlerinjektionsbefehls soll über die *CID*-Register und die *FaultID* über die *FID*-Register übertragen werden. Die Register *D*, *O*, *R* und *V* sind zur Übertragung der Fehler- und Aktivierungsdaten *occurences, duration, reactivations* und value zuständig. Der Reguläre Ausdruck *regEx* wird über das *regEx*-Register übertragen und die Aktivierungsmuster über das *ActiP*-Register für Aktivierungspunkte.

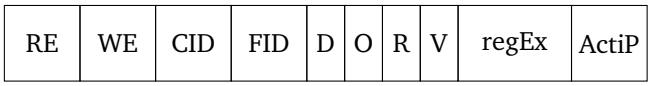


Abbildung 16.22.: Aufbau des Software-Hardware-Protokolls

Das Hardware-Software-Protokoll (siehe Abbildung 16.23) soll die Befehle *FaultStateChange* und *PartialActivation* übertragen. Das Protokollübertragungsbit *RE (ReadEnable)* wird gebraucht, damit das FPGA dem Mikrocontroller mitteilen kann, dass eine neue Nachricht in den Registern liegt. Der Gebrauch von *WriteEnable*-Signalen für den Erlaub des Neubeschreibens der Software-Hardware-Registern ist anhand der schnellen Verarbeitung des FPGAs nicht notwendig. Die *CID*-Register übertragen wieder den Fehlerinjektionsbefehl und die *FID*-Register die *FaultID*. Für die partielle Aktivierung des *PartialActivation*-Befehls übersenden die Register *ActiP* den Aktivierungspunkt. Für den Zustand *FaultState* des *FaultStateChange*-Befehls wird das Register *FS* verwendet.

Abbildung 16.23.: Aufbau des Hardware-Software-Protokolls

Eine beispielhafte Kommunikation zwischen Mikrocontroller und FPGA kann man in Tabelle 16.3 und Tabelle 16.4 erkennen. Zuerst wird über die Software-Hardware-Register der Befehl *FaultInjection* übertragen, um die Fehlerinjektion vor zu bereiten. Der Fehler ist wieder der Bitflip in dem *OrbitPlanning*-Signal in dem *FPGA-HW-SW-Communication* Modul. Der Fehler soll nur einmal injiziert werden und sich nicht reaktivieren lassen. Diesmal ist das Aktivierungsmuster des Fehlers gesetzt. Wenn in der Software-Hardware-Kommunikation auf dem FPGA

Tabelle 16.3.: Beispielhaftes Versenden eines FaultInjection- und eines FaultInjectionActivations-Befehles

RE	WE	CID	FID	D	O	R	V	regEx	ActiP
1	-	FaultInjection	BitFlip, - , HW-SW-Comm., OrbitPlanning[3]	-	1	0	-	"1"	SW-HW-Comm., MissionControl
0	-	FaultInjection-Activations	BitFlip, - , HW-SW-Comm., OrbitPlanning[3]	-	3	0	7	-	SW-HW-Comm., MissionControl

Tabelle 16.4.: Beispielhaftes Versenden zweier Fault_State_Change- und eines PartialActivation-Befehles

RE	CID	FID	ActiP	FS
-	FaultStateChange	BitFlip, - , HW-SW-Comm., OrbitPlanning[3]	-	Prepared
-	PartialActivation	BitFlip, - , HW-SW-Comm., OrbitPlanning[3]	SW-HW-Comm., MissionControl	-
-	FaultStateChange	BitFlip, - , HW-SW-Comm., OrbitPlanning[3]	-	Injected

das *MissionControl*-Signal einen bestimmten Wert erreicht hat, soll der Fehler aktiviert werden. Da der Fehler im Injektionspunkt nicht die genauen Aktivierungsdaten braucht, werden diese Daten im Befehl weggelassen. Im reglären Ausdruck *regEx* enthält den Wert „1“, was hier als Abkürzung von nur den einen Aktivierungsmuster im *MissionControl*-Signal bedeutet. Das hier willkürlich gesetzte *ReadEnable*-Bit gibt an, dass der Befehl zum Lesen von dem FPGA bereit ist. Zum Vorbereiten des Aktivierungsmusters des Fehlers wird der *FaultInjectionActivations*-Befehl übersendet. Dieser übergibt den Aktivierungspunkt im *MissionControl*-Signal und gibt zur Referenz auf den Fehler die *FaultID* an. Die Daten *D, O, R* und *V* sind diesmal die Daten des Aktivierungsmusters, die aussagen, dass die Aktivierung erst ausgegeben werden soll, wenn das *MissionControl*-Signal den Wert 7 dreimal erreicht hat. Auch diese Aktivierung soll nicht reaktiviert werden, daher nach der Aktivierung gelöscht werden. Das hier willkürlich gesetzte *ReadEnable*-Bit gibt an, dass der Befehl noch erst von dem Mikrocontroller bestätigt werden muss, bevor dieser vom FPGA gelesen werden darf.

Auf dem FPGA überträgt das Modul des Injektionspunktes des Fehlers ein *FaultStateChange*-Befehl über die Hardware-Software-Register an den Mikrocontroller. Dieser besitzt neben der bereits bekannten *FaultID*, den Zustand des Fehlers. In dieser Übertragung ist der Fehler vorbereitet, daher im Zustand *Prepared*. Da die Werte der *RE*-Register keine Rolle in diesem Beispiel haben, sind sie nicht weiter angegeben. Als nächstes wird zu einem Zeitpunkt auf dem FPGA das Aktivierungsmuster aktiviert. Es sendet den *PartialActivation*-Befehl zu dem Injektionspunkt des Fehlers, der diesen aktiviert. Der Mikrocontroller erhält auch den Befehl über die Hardware-Software-Register. In diesem Befehl ist die *FaultID* und der Aktivierungspunkt *ActiP* enthalten. Durch die Aktivierung des Fehlers wird auch wieder ein *FaultStateChange*-Befehl mit dem Zusatnd *injected* übertragen.

16.2.7. Modultests

Die Fehlerinjektion hat Modultests auf C++- und SystemC- Basis durchgeführt. In SystemC wurden Testbenches, wie in der Satellitensteuerung beschrieben (siehe Abschnitt: 15.2.3), durchgeführt und die SC_ASSERTS (siehe Abschnitt: 14.2.3) der nachfolgenden beschriebenen Simulationstests (siehe Abschnitt: 14.2.3) benutzt. Bei C++-Klassen wurden zwar auch SC_ASSERTS und Tests in CMake erstellt, aber die Modulanbindung konnte durch die Pointer auf die kommunizierenden Komponenten nur mit Mock-Ups getestet werden. Da die Mock-Ups für alle Tests verwenden wurden, wurde entschieden, dass die externen Methodenaufrufe nur ausgegeben werden. Das hat den Nachteil, dass die externen Methodenaufrufe nicht automatisch überprüft werden können. Bei einigen Tests, besonders bei denen mit viel Kommunikation, macht es Sinn die Testausgaben genau zu überprüfen.

16.3. Implementierung

Zwischen dem Modulentwurf und der finalen Implementierung wurden aufgrund der simulierten Satellitensteuerung mit SystemC zwei Zwischenmodelle eingeführt. In diesem Abschnitt werden die benutzen Modelle vorgestellt und die Implementierung auf dem Fehlerinjektions-PCmitsamt den Änderungen zum Modulentwurf vorgestellt.

16.3.1. Modellbeschreibung

16.3.2. Implementierung?

16.3.3. Meilensteine

Teil V.

Integrationstests

Integrationstests dienen dazu, voneinander abhängige Komponenten eines (Teil-)Systems im Zusammenspiel miteinander zu testen. Im Folgenden sollen nun die Begriffe Model-in-the-loop (MIL), Software-in-the-loop (SIL) und HIL geklärt werden.

Model-in-the-loop: Als Model in the loop wird ein Verfahren bezeichnet, bei dem das Model eines eingebetteten Systems in einer simulierten Umgebung betrieben wird, um zu prüfen ob es auf Sensordaten (im Projekt Umweltdaten) richtig verarbeitet und die Stimuli, die vom eingebetteten System stammen, richtig an die Umwelt, also die Simulationsumgebung, weitergegeben werden. In der Folge der weiterentwicklung des Modells wird zwischen SIL und hil unterschieden.

Software-in-the-loop: Bei SIL wird das Software-Modell des eingebetteten Systems mit der Simulationsumgebung verbunden und so in einer Schleife simuliert. So erhält das Software-Modell durchgehend Umweltdaten, während es mit seinen Aktuatoren die Umwelt beeinflussen kann.

Hardware-in-the-loop: Bei HIL wird das Modell auf die Hardware gebracht und diese dann mit der Simulationsumgebung verbunden. Im Rahmen dieses Projektes wird die Satellitensteuerung also auf die ZedBoard gespielt und diese dann mit KSP als Simulationsumgebung verbunden.

17. Systemtests

Es liegen nicht viele Systemtests vor, da zum Projektende nur wenig Zeit zum Testen war. Nichtsdestotrotz sollen die vorhandenen Systemtests vorgestellt werden. Die Testbenches befinden sich im Ordner `model/test/sys/` bzw. auch in `model/test/hil`. Nur das Teilprojekt Fehlerinjektion kann teilsystemübergreifende Tests vorweisen. Diese Tests bekommen verschiedene Eingaben:

- Kommandos der Benutzerschnittstelle
- Kontrollsignale, Managementdaten, Log-Daten und FI-Meldungen der ZedBoards.

Es werden folgende Ausgaben geliefert:

- UI-Kommandos für das GUI
- Kontrollsignale, Managementdaten, Log-Nachrichten und FI-Meldungen für die Simulation, außerdem
- Umweltsignale und Management-Daten für die ZedBoards.

Die Testergebnisse der Fehlerinjektion werden nun tabellarisch aufgeführt:

Tabelle 17.1.: Ergebnisse der Logische Fehlerinjektions Tests

Bez.	Testfall	Erwartetes Resultat	Testergebnis
Allgemeine logische Testfälle der Fehlerinjektions-Software (FI-Software)			
CTC001	Prüfen, ob die FI-Software gestartet werden kann.	Die FI-Software wird gestartet und das Hauptmenü wird angezeigt.	Erfolgreich
CTC002	Prüfen, ob die FI-Software geschlossen werden kann	Alle Prozesse der FI-Software wurden beendet.	Gestrichen
logische Testfälle zu den funktionalen Anforderungen			
CTC003	Prüfen, ob Umweltsignale manipuliert werden können.	Umweltsignale können manipuliert werden.	Erfolgreich
CTC004	Prüfen, ob die von der Satellitensteuerung an die Simulation gesendeten Steuerungsbefehle manipuliert werden können.	Steuerungssignale, die von der Satellitensteuerung an die Simulation gesendet werden, können manipuliert werden.	Erfolgreich
CTC005	Prüfen, ob Fehler zur Manipulation des Satellitenzustands vom Benutzer an die Satellitensteuerung gesendet werden können.	Fehler zur Manipulation des Satellitenzustands können vom Benutzer an die Satellitensteuerung gesendet werden.	Gestrichen
CTC006	Prüfen, ob die Satellitensteuerung Fehler zur Manipulation des Satellitenzustands erhält.	Die Satellitensteuerung erhält Fehler zur Manipulation des Satellitenzustands.	Gestrichen

Tabelle 17.1.: Ergebnisse der Logische Fehlerinjektions Tests

Bez.	Testfall	Erwartetes Resultat	Testergebnis
CTC007	Prüfen, ob empfangene Fehler zur Manipulation des Satellitenzustands injiziert werden.	Empfangene Fehler zur Manipulation des Satellitenzustands werden injiziert.	Gestrichen
CTC008	Prüfen, ob Fehler in einzelne Satellitenkomponenten injiziert werden können.	Fehler können in einzelne Satellitenkomponenten injiziert werden.	Gestrichen
CTC009	Prüfen, ob die Satellitenkomponenten der Fehlerinjektion bekannt sind.	Der Fehlerinjektion sind die Satellitenkomponenten bekannt.	Erfolgreich
CTC010	Prüfen, ob Informationen über die Satellitenkomponenten vorhanden sind.	Es sind Informationen über die Satellitenkomponenten vorhanden.	Erfolgreich
CTC011	Prüfen, ob die Kommunikation zwischen Simulator und Satellitensteuerung abgefangen oder zwischengespeichert wird.	Die Kommunikation zwischen Simulator und Satellitensteuerung wird abgefangen oder zwischengespeichert.	Erfolgreich
CTC012	Prüfen, ob es ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellitensteuerung gibt.	Es existiert ein Kommunikationsprotokoll zwischen Fehlerinjektion und der Fehlerinjektionskomponente auf dem Satellitensteuerung.	Gestrichen
CTC013	Prüfen, ob ein Satellitenlog existiert, welches an die Fehlerinjektionskomponente auf dem Host-PC gesendet wird.	Es existiert ein solcher Log.	Erfolgreich
CTC014	Prüfen, ob es eine Systemzustandsüberwachung der Satellitensteuerung für den Fehlerinjektionszugriff gibt.	Es existiert eine solche Systemzustandsüberwachung.	Gestrichen
CTC015	Prüfen, ob die Maßnahmen der Satellitensteuerung dem Anwender zur Verfügung stehen.	Die Maßnahmen der Satellitensteuerung stehen dem Anwender zur Verfügung.	Erfolgreich
CTC016	Prüfen, ob jeder Log-Eintrag mit einem Timestamp enthält.	Alle Log-Einträge enthalten einen Timestamp.	Erfolgreich
CTC017	Prüfen, ob eine Speichereinheit aller injizierten Fehler existiert.	Es existiert eine solche Speichereinheit.	Erfolgreich

Tabelle 17.1.: Ergebnisse der Logische Fehlerinjektions Tests

Bez.	Testfall	Erwartetes Resultat	Testergebnis
CTC018	Prüfen, ob auf die Speichereinheit aller injizierten Fehler zugegriffen werden kann.	Es kann auf die Speichereinheit aller injizierten Fehler zugegriffen werden.	Erfolgreich
CTC019	Prüfen, ob ein UI zur Anzeige aller injizierten Fehler existiert.	Es existiert ein UI zur Anzeige aller injizierten Fehler.	Erfolgreich
CTC020	Prüfen, ob die Kommandozeile auf die Speichereinheit aller injizierten Fehler zugreifen kann.	Die Kommandozeile kann auf die Speichereinheit zugreifen.	Gestrichen
CTC021	Prüfen, ob der Fehlerinjektionszugriff auf Speicherblöcken oder durch speziell definierte Komponenten geschieht.	Der Fehlerinjektionszugriff geschieht auf Speicherblöcken oder durch speziell definierte Komponenten.	Gestrichen
CTC022	Prüfen, ob dem Anwender Eingabebefehle zur Verfügung gestellt werden.	Den Anwender werden Eingabebefehle zur Verfügung gestellt.	Gestrichen
CTC023	Prüfen, ob eine GUI zur Steuerung der Fehlerinjektion existiert.	Es existiert eine solche GUI.	Erfolgreich
logische Testfälle zu den nichtfunktionalen Anforderungen			
CTC024	Prüfen, ob die Nutzerschnittstelle der Fehlerinjektion einfach bedienbar ist.	Die Nutzerschnittstelle der Fehlerinjektion ist einfach bedienbar.	Nicht getestet
CTC025	Prüfen ob Fehler in unter einer Sekunde injiziert werden.	Fehler werden in unter einer Sekunde injiziert.	Gestrichen
CTC026	Prüfen, ob die Fehlerinjektionskomponente auf dem FPGA möglichst platzsparend ist.	Die Fehlerinjektionskomponente auf dem FPGA ist so platzsparend wie möglich.	Gestrichen
CTC027	Prüfen, ob die Satellitenkomponenten auf mögliche Fehlerquellen untersucht wurden.	Die Satellitenkomponenten wurden auf mögliche Fehlerquellen untersucht.	Erfolgreich
CTC028	Prüfen, ob die Auswahl der injizierbaren Fehler definiert wurde.	Die Auswahl der injizierbaren Fehler wurde definiert.	Erfolgreich
CTC029	Prüfen, ob die Kommunikation weniger als 1 s Verzögerung verursacht.	Die Kommunikation verursacht weniger als 1 s Verzögerung.	Erfolgreich

Tabelle 17.1.: Ergebnisse der Logische Fehlerinjektions Tests

Bez.	Testfall	Erwartetes Resultat	Testergebnis
CTC030	Prüfen, ob die Verzögerung der Kommunikation möglichst konstant ist.	Die Verzögerung der Kommunikation ist so konstant wie möglich.	Gestrichen
CTC031	Prüfen, ob der Satellitensystemzustand übersichtlich angezeigt wird.	Der Satellitensystemzustand wird übersichtlich angezeigt.	Gestrichen
CTC032	Prüfen, ob Log-Einträge für den Anwender verständlich sind.	Anwender verstehen alle Log-Einträge.	Nicht getestet
CTC033	Prüfen, ob injizierte Fehler übersichtlich angezeigt werden.	Injizierte Fehler werden übersichtlich angezeigt.	Nicht getestet

Die folgenden Fehler des Teilprojekts Fehlerinjektion konnten identifiziert, aber nicht beseitigt werden:

Kennzeichnung:	**Fault 013**
Bezeichnung:	**Beim Systemtest funktioniert keine Fehlerinjektion**
Referenz:	
Beschreibung	Bei dem Systemtest wurde deutlich, dass einige der getesteten Fehler nicht wie geplant injiziert werden.
Korrektes Verhalten:	Die Fehler werden ordnungsgemäß injiziert.
Behoben:	**Voraussichtlich bis zur Endpräsentation erledigt.**
Lösungsansatz:	Die Fehler fixen.

Kennzeichnung:	**Fault 014**
Bezeichnung:	**Die FI wartet, wenn die Message Queues voll sind**
Referenz:	
Beschreibung	Ein Paket das durch geleitet werden soll, kann bei einer vollen Message Queue, zur Information der GUI, nicht an den Simulator geschickt werden, da es blockierend wartet.
Korrektes Verhalten:	Die FI wartet nicht und die Daten werden in eine Queue geschrieben. Nach einer bestimmten Größe werden die Queues nicht weiter gefüllt.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Kurze Lösung ist, voraus zu setzen, dass die GUI angeschlossen sein muss. Lange und bessere Lösung wäre die Befehle zwischenzuspeichern.

Im Rahmen der Systemtests konnten außerdem die folgenden Fehler im Teilprojekt Simulator identifiziert, jedoch nicht abgestellt werden:

Es konnten außerdem die folgenden Fehler im Teilprojekt Satellitensteuerung identifiziert, aber nicht beseitigt werden:

Kennzeichnung:	**Fault 015**
Bezeichnung:	**kRPC liefert falsche Wert für den Abstand zwischen Asteroid und Satellit**
Referenz:	LTC007
Beschreibung	Folgendes Verhalten wird beobachtet: Für Abstände zwischen 0 und 200m liefert kRPC den richtigen Abstand. Für einen Abstand von 200.1m liefert kRPC einen Abstand von 37m, für alle größeren Abstände ist der Abstand um 170m zu gering. Der Fehler erschwert die Abschätzung, ob der Satellit wirklich in der Nähe des Asteroiden ist. Andocken ist nicht zuverlässig möglich. Fehler wurde von mehreren Personen bestätigt und kommt nicht von den Berechnungen aus dem Modell.
Korrektes Verhalten:	Die Daten werden konsistent vom Plugin geliefert, bzw von KSP zur Verfügung gestellt.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Dieses Problem besteht bereits in KSP selber. Ein Grund für dieses Verhalten ist nicht ersichtlich.

Kennzeichnung:	**Fault 016**
Bezeichnung:	**Setzen von retrograde und target**
Referenz:	
Beschreibung	Die Befehle zum Setzen des SAS auf retrograde und target werden im logischen Modell manchmal von kRPC ignoriert.
Korrektes Verhalten:	Die Befehle sollten von kRPC niemals ignoriert werden.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Fehler liegt wahrscheinlich in KSP, beim Auftreten des Fehlers ist die Mission vorbei. Fehler ist nicht schwerwiegend, da diese Funktion erst gegen Ende der Mission benötigt werden.

Kennzeichnung:	**Fault 017**
Bezeichnung:	**KSP Steuerung wird beim Setzen von Steuerwerten zu bestimmten Zeiten außer Kraft gesetzt**
Referenz:	
Beschreibung	Wenn in KSP Steuerbefehle zu bestimmten Zeiten gesetzt werden, ist anschließend die gesamte Steuerung, sowohl über Addons, als auch im Spiel selber, nicht mehr funktionsfähig. Dies geschieht ebenfalls wenn man das Spiel per Hand steuert.
Korrektes Verhalten:	
Behoben:	**Nicht beseitigbar. Interner Fehler von KSP.**
Lösungsansatz:	Wenn dieser Fehler auftritt ist die Mission gescheitert und KSP muss neu gestartet werden.

Kennzeichnung:	**Fault 018**
Bezeichnung:	**KSP reagiert bei dem Senden von Paketen manchmal nicht auf Befehle**
Referenz:	
Beschreibung	Bei dem Benutzen des eigenen Plugins werden manchmal eingehende Pakete richtig empfangen und verarbeitet, allerdings werden die Aktionen nicht in KSP ausgeführt. Wenn dieser Fehler auftritt kann KSP ebenfalls nicht mehr direkt gesteuert werden.
Korrektes Verhalten:	Alle Pakete sollten korrekt ihre entsprechende Aktion ausführen.
Behoben:	**Nicht beseitigbar. Interner Fehler von KSP.**
Lösungsansatz:	Wenn dieser Fehler auftritt ist die Mission gescheitert und KSP muss neu gestartet werden.

Kennzeichnung:	**Fault 019**
Bezeichnung:	**Beim Andocken wird das Ziel verloren**
Referenz:	
Beschreibung	Bei dem erfolgreichen Andocken an das Ziel geht die Zielmarkierung in KSP verloren. Anschließend können keine Informationen mehr über das Ziel abgefragt werden. Das anschließende Abfragen von Informationen führt zum Abstürzen des Systems.
Korrektes Verhalten:	Das anschließende Abfragen von Informationen sollte nicht das System zum abstürzen bringen.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Die Mission ist nach dem erfolgreichen Andocken an den Asteroiden beendet. Der nachfolgende Absturz des Systems ist nicht schwerwiegend.

Kennzeichnung:	**Fault 020**
Bezeichnung:	**lwIP verwirft Pakete**
Referenz:	
Beschreibung	Bei der Ausführung der Satellitensteuerung auf dem Zedboard gehen Pakete verloren, wenn zu viele Pakete in zu kurzer Zeit geschickt werden.
Korrektes Verhalten:	Es sollten keine Pakete verloren werden.
Behoben:	**Nicht beseitigt.**
Lösungsansatz:	Durch die Begrenzung der Anzahl an Pakete die in einer bestimmten Zeit gesendet werden kann dieses Problem größtenteils umgangen werden.

Teil VI.

Fazit

18. Zusammenfassung

Die PG Guardian hat im Laufe eines Jahres ein funktionierendes Endprodukt fertiggestellt. Um dies zu realisieren, hat das Teilprojekt Simulator das Teilsystem Simulator entworfen, welches die Schnittstelle zur Simulationsumgebung KSP darstellt. Die Schnittstelle ist angebunden an das Teilsystem Fehlerinjektion, welches dem Benutzer die Möglichkeit liefert, per GUI Fehler in das System zu injizieren. Auch wird der, über die FI fließende Datenverkehr überwacht und protokolliert. Das Teilsystem Fehlerinjektion leitet die Daten weiter an das Teilsystem Satellitensteuerung. Diese wird im Endprodukt über vier ZedBoards realisiert, auf deren FPGA-Komponente Teile der Orbitkalkulation laufen, während die restlichen Funktionen durch den Mikrocontroller umgesetzt werden.

Das Endprodukt erfüllt die selbst gewählte Mission und navigiert den Satelliten von Kerbin aus zum Asteroiden, der Kerbin bedroht. Dort angekommen nähert er sich dem Asteroiden und verändert seinen Kurs mit Hilfe eines Greifarms derart, dass der Asteroid keine Gefahr mehr für Kerbin darstellt.

19. Probleme

Im Rahmen dieses Projektes haben sich Probleme verschiedenster Art ergeben, auf die nun eingegangen wird.

Personalprobleme: An der Carl-von-Ossietzky Universität sind die Projektgruppen auf eine Stärke von zwölf Personen ausgelegt. Diese Projektgruppe hatte effektiv zwei Fälle, durch die Arbeitskraft kompensiert werden musste. Ein Mitglied hat die Projektgruppe im ersten Drittel verlassen, ein weiteres Mitglied konnte nur mit verminderter Stundenzahl teilnehmen. Erschwerend kam hinzu, dass zwei Mitglieder neben dem Studium in Vollzeit außerhalb Oldenburgs beschäftigt waren, sodass sie oft nicht persönlich anwesend sein konnten.

Der Simulator: KSP diente der Projektgruppe als Simulationsumgebung. Es stellt eine API zur Verfügung [47], eine Dokumentation dieser API wird aber nicht geliefert. Zwar gibt es in der Fancommunity Projekte, eine Dokumentation zu rekonstruieren [59, 4], die Inhalte sind aber unvollständig und oft geraten. Auch die Physik innerhalb von KSP ist eine vereinfachte Version, die die Projektgruppe vor verschiedene Hindernisse gestellt hat. Weiter ist die Unity Engine, Forenberichten zufolge, nicht threadsafe. Dasselbe konnte für die von KSP bereitgestellte API festgestellt werden. Werden bestimmte Funktionen von einem externen Thread aufgerufen, kann dies dazu führen, dass das gesamte Spiel einfriert und teilweise sogar ohne Fehlermeldung abstürzt. Dieses Verhalten konnte auf die Unity Engine zurückgeführt werden.

Fehlplanungen: Im Rahmen dieses Projektes wurde schon früh ein Zeitplan ausgearbeitet, der das komplette Projekt bis zur Endabgabe umfasste. Bei der Planung wurden einige Faktoren nicht berücksichtigt, so z. B. eine ausreichende Pufferzone, um unvorhergesehene Entwicklungen abzufedern. Auch wurde der Aufwand für den Prototypen für die Zwischenpräsentation falsch geplant, sodass die Teilprojekte in zeitlichen Verzug gerieten. Aus diesem und anderen Gründen musste der Funktionsumfang der Teilsysteme gekürzt werden.

20. Erkenntnisse

In diesem Projekt wurde versucht, den Ansatz des SLDs umzusetzen. Die Idee war, ein SystemC Modell der Satellitensteuerung zu entwickeln und mit Hilfe von High Level Synthese auf die Hardware zu bringen. Es hat sich gezeigt, dass die Technik in diesem Bereich noch verbesserungsfähig ist, denn um High Level Synthese durchführen zu können, müssen bestimmte Einschränkungen beim modellieren beachtet werden. Dies hätte durch einen höheren Erfahrungsschatz oder gründlichere Recherche vermieden werden können.

Eine weitere Erkenntnis ist, dass die Projektgruppe mit der Motivation der Mitglieder steht und fällt. So konnte, trotz vieler Hindernisse, das Ziel der Projektgruppe erfolgreich erreicht werden, was ohne hochmotivierte Mitglieder nicht möglich gewesen wäre. Allerdings kann auch eine motivierte Gruppe fehler in der Planung nur begrenzt ausgleichen.

Ein äußerst wichtiger Faktor bei so einem Projekt ist die Kommunikation. Trotz räumlicher Hindernisse konnten Kommunikationsstrukturen etabliert werden, die die Funktionsfähigkeit und Effektivität der Teilprojekte sicherstellen konnte.

In dieser Projektgruppe gab es nur wenig soziale Aktivitäten. Um den Teamgeist weiter zu festigen, ohne natürlich Arbeitszeit zu opfern, kann eine Erhöhung des Umfangs sozialer Aktivitäten die Motivation, den Zusammenhalt und zwischenmenschliche Beziehungen stärken.

21. Ausblick

In dieser Projektgruppe konnten aus diversen Gründen nicht alle Funktionen implementiert werden, die zum Testen verschiedener Adaptionsverfahren einer Satellitensteuerung nötig sind. Wird die Arbeit fortgesetzt, muss die Fehlerinjektion noch einmal überarbeitet und erweitert werden.

Auch kann die Satellitensteuerung um weitere Adaptionsverfahren erweitert werden, die dann durch die Fehlerinjektion getestet werdem können.

Eine Möglichkeit der Optimierung kann sein, den Mikrocontroller auf den ZedBoards zu entlasten und mehr Berechnungen auf die programmierbare Logik zu verschieben. Sollte dies mit den vorhandenen Boards nicht möglich sein, muss über eine Erweiterung nachgedacht werden. Auch die Simulationsumgebung ist nicht ideal gewählt. Hier lohnt es sich zu recherchieren, um Alternativen zu finden, möglicherweise kann auch eine Kooperation mit Instituten etabliert werden, die eine realitätsnahe Simulationsumgebung besitzten.

Teil VII.

Anhang

A. Gruppenmitglieder

An der Carl von Ossietzky Universität bestehen die Projektgruppen aus mindestens sechs und maximal zwölf Studierenden. Die PG Guardian besteht aus elf Personen, die hier kurz vorgestellt werden sollen.

Alexander van Düllen

Aufgabe: Systemadministrator

Gruppe: Simulator

Werdegang: Fach-Bachelor Informatik mit Schwerpunkt Eingebettete Systeme und Mikrorobotik an der Carl von Ossietzky Universität Oldenburg

Interessen: Hardwarenahe Systementwicklung, Sicherheitskritische Systeme, Realzeitbetriebsysteme, Vim

Paul Hannibal

Aufgabe: Urlaubsbeauftragter

Gruppe: Satellitensteuerung

Werdegang: Fach-Bachelor Informatik an der Carl von Ossietzky Universität Oldenburg

Interessen: Informatik, Mathematik

Nils Heinig

Aufgabe: Dokumentation

Gruppe: Fehlerinjektion

Werdegang: Fach-Bachelor Informatik an der Carl von Ossietzky Universität Oldenburg

Interessen: Theoretische Informatik

Lennart Hoffhues

Aufgabe: Stellvertretender Testmanager, Integrationsmanager

Gruppe: Satellitensteuerung

Werdegang: Fach-Bachelor Informatik mit Schwerpunkt Eingebettete Systeme und Mikrorobotik an der Carl von Ossietzky Universität Oldenburg

Interessen: Hardwarenahe Entwicklung

Tino Hoffmann

Aufgabe: Projektmanager

Gruppe: Simulator

Werdegang: IT-Entwickler, Fach-Bachelor Wirtschaftsinformatik an der Jade Hochschule Wilhelmshaven / Oldenburg / Elsfleth

Interessen: Projektmanagement, Netzwerktechnik, Qualitätsmanagement, Testmanagement, Java

Marten Horstmann

Aufgabe: Schriftführer

Gruppe: Satellitensteuerung

Werdegang: Ausbildung zum Elektroniker für Automatisierungstechnik, Bachelor of Engineering Elektrotechnik an der PHWT Oldenburg, Inbetriebnehmer bei ThyssenKrupp System Engineering GmbH

Interessen: Regelungstechnik, Eingebettete Systeme

Mark Kettner

Aufgabe: Subgruppenleiter Satellitensteuerung, Stellvertretender Projektleiter, Öffentlichkeitsarbeit

Gruppe: Satellitensteuerung

Werdegang: Fach-Bachelor Informatik mit Vertiefung Technische Informatik an der Hochschule Emden / Leer

Interessen: Entwicklung eingebetteter Systeme, Emacs

Oliver Klemp

Aufgabe: Subgruppenleiter Fehlerinjektion

Gruppe: Fehlerinjektion

Werdegang: Fach-Bachelor Informatik mit Schwerpunkt Theoretische Informatik an der Carl von Ossietzky Universität Oldenburg

Interessen: Hardwarenahe Entwicklung, KI, Theoretische Informatik

Sven Lampe

Aufgabe: Infrastrukturbeauftragter

Gruppe: Fehlerinjektion

Werdegang: Zwei-Fächer-Bachelor Informatik/Germanistik

Interessen: Automatentheorie, Logik, Realzeitsysteme, künstliche Intelligenz

Marvin Menke

Aufgabe: Testmanager

Gruppe: Simulator

Werdegang: Fach-Bachelor Wirtschaftsinformatik an der Jade Hochschule Wilhelmshaven / Oldenburg / Elsfleth

Interessen: Testmanagement, Java

Patrick Uven

Aufgabe: Öffentlichkeitsarbeit

Gruppe: Satellitensteuerung

Werdegang: Schulische Ausbildung zum staatlich geprüften technischen Assistenten für Informatik, Abitur, Fach-Bachelor Wirtschaftsinformatik an der Carl von Ossietzky Universität Oldenburg

Interessen: Eingebettete System, Hardwarentwicklung, FPGAs, Realzeitbetriebssysteme, Unix, Vim

B. Funktionsumfang des Prototypen

In den folgenden Unterabschnitten wird der Funktionsumfang des Host-Only-Prototypen dokumentiert. Es wird eine Co-Simulation zwischen KSP und dem SystemC-Modell stattfinden (siehe Abschnitt B.1). Der Start der Trägerrakete wird per Mechjeb[1] automatisiert (siehe Abschnitt B.2). Angelangt im LEO, übernimmt das SystemC-Modell die Kontrolle über den Satelliten und befördert diesen zum Zielasteroiden (siehe Abschnitt B.3).

Der Funktionsumfang der Fehlerinjektion zum Prototypen ist in Abschnitt B.4 beschrieben.

Des Weiteren wird im Kapitel C vorgestellt, wie die Kommunikation zwischen dem ZedBoard und dem Host-PC abläuft.

B.1. SystemC-KSP-Co-Simulation

Eine Teilmenge der Komponenten wird in SystemC für den Prototypen modelliert. Dabei handelt es sich um Orbitplanung, zum Teil dem Fortbewegungskonzept und zum Teil der Schnittstelle.

Die Orbitplanung teilt sich hierbei in Orbitberechnung und Maneuverplanung auf. Ersteres berechnet die entsprechenden Maneuverdaten und Zeiten. Letzteres sendet die Maneuverdaten an das Fortbewegungskonzept, welches diese in die Steuerbefehle für die Aktuatoren auf dem Satelliten umsetzt.

Die Synchronisation wird durch Erweiterung der `wait(time)`-Methode von SystemC realisiert. Zu diesem Zweck wurde ein `sc_ksp::wait` angelegt, welches das selbe Interface bereit stellt wie `sc_core::wait`. In dieser Methode wird zunächst der entsprechende Aufruf des `sc_core::wait` ausgelöst um die Zeit des SystemC-Modells voranzutreiben. Anschließend wird die in dem SystemC-Modell vorherrschende Zeit auf die Simulation übertragen. Synchronisation kann für den Prototypen maximal auf eine Sekunde genau hergestellt werden. Die Möglichkeit einer genaueren Synchronisation wird noch evaluiert. Es werden darüber hinaus keine Delta-Zyklen Implementiert. Es ist daher möglich das die Zeiten zwischen dem SystemC-Modell und KSP divergieren falls die Zeit im SystemC-Modell im Vergleich langsamer verläuft.

Die Schnittstellenkomponente übernimmt hierbei die Kommunikation mit KSP. Dabei werden bisher direkt Befehle zur Ausrichtung an KSP übergeben, da kRPC nur eine Fernsteuerung für das Computerspiel darstellt. Des Weiteren werden Umweltwerte, die normalerweise nur durch Sensorwerte ausgerechnet werden können, direkt über kRPC[44] ausgelesen und über die Schnittstellenkomponente an die Modelle in SystemC weitergegeben.

B.2. Mechjeb-Steuerung

Der Satellit soll zunächst in die Umlaufbahn der Erde gebracht werden. Hierzu wird die Mod *MechJeb 2* verwendet. Der Mod wird den Start und das Staging der Trägerrakete bis zu einer Höhe von 110km übernehmen. Von dort muss über ein Manöver eine Umlaufbahn um die Erde geflogen werden. Für einen erfolgreichen Start bis zu einer Höhe von 110km sind folgende Einstellungen in MechJeb 2 vorzunehmen.

- Engage Autopilot
- Orbit altitude 110 km
- Prevent overheats = true

[1] `http://wiki.mechjeb.com/`

- Autostage = true
- Stop at stage 2

Die Restlichen Einstellungen benötigen keiner weiteren Anpassung.

B.3. Orbitplanung der Satellitensteuerung ab LEO

Die Phasen 0-3 werden durch die Mod MechJeb 2 durchgeführt. Danach hat die Trägerrakete eine Umlaufbahn in der Höhe von 110 km erreicht. In den folgenden Phasen wird die Trägerrakete über mehrere Manöver durch das SystemC-Modell an den Asteroiden angenähert. Wichtige Eckdaten zum Erreichen der Mission sind:

- Start der Übernahme der Trägerrakete durch das SystemC-Modell bei einer stabilen Umlaufbahn in Höhe von 110 km. Eine stabile Umlaufbahn hat die Trägerrakete erreicht wenn Apoapsis und Periapsis eine Differenz von Maximal 20 km haben und nicht eine Höhe von 80 km unterschreiten.
- Das Ziel der Mission des Prototypen ist erreicht, wenn sich dem Asteroiden auf 200 m angenähert wurde.

B.4. Fehlerinjektion

Der Teil der Fehlerinjektion des Prototyps umfasst eine Monitoring Komponente. Die Komponente hat zu jeder modellierten Satellitensteuerungskomponente einen Kanal, über die die Logeinträge gesendet werden. Danach sollen die Logeinträge auf der Konsole ausgegeben werden.

C. Kommunikationsprototyp

Im Rahmen des Zwischenberichts wurde ein Prototyp zur Demonstration der Kommunikation zwischen den ZedBoards sowie dem Host-PC gefordert. Im Folgenden werden die Funktionsweise, die Bestandteile und der Aufbau des Prototypen beschrieben.

C.1. Komponenten

Der Prototyp ist in zwei Komponenten aufgeteilt, welche über das UDP/IP-Protokoll kommunizieren. Physisch wird für die Kommunikation zwischen den Komponenten eine gewöhnliche Ethernet-Verbindung verwendet. Über diese können spätere mehrere ZedBoards, sowie der Host-PC verbunden werden. Innerhalb der UDP-Pakete wird ein von der Projektgruppe festgelegtes Prokoll verwendet, welches eine interne Adresse, die Daten, sowie eine Prüfsumme beinhaltet. Im Rahmen des Prototypen werden nur die Daten betrachtet, die Felder der Adresse und Checksumme werden erst in folgenden Zyklen implementiert. Zudem haben sowohl die Adresse, die Daten als auch die Checksumme eine Länge von einem Byte. Abbildung C.1 zeigt den Aufbau der im Folgenden näher beschriebenen Komponenten.

C.1.1. Host-PC

Der Host-PC stellt ein x86_64-System dar, das unter Windows 7 betrieben wird und sowohl den Simulator KSP als auch eine virtuelle Maschine, in der Ubuntu 16.04 als Umgebung bereit gestellt wird, ausführt. Der virtuellen Maschine steht ein Netzwerkinterface im Bridged Modus zur Verfügung, was die direkt Ethernet basierte Kommunikation zwischen Host-PC und ZedBoard ermöglicht.

Da eine Erweiterung des kRPC-Plugin um die vom ZedBoard verwendete UDP-Schnittstelle zu viel Zeit in Anspruch nimmt, wird diese für den ersten Prototypen nicht angepasst. Um dennoch eine Kommunikation zu ermöglichen wurde ein Programm entwickelt, das zwischen diesen beiden Protokollen als Adapter dient. Dieser erfüllt zwei Aufgaben:

1. Er lauscht auf dem Port 50.003 auf Signale vom ZedBoard im festgelegten, UDP basierten Protokoll. Um die erhaltenen Steuersignale in KSP umzusetzen, werden die entsprechenden kRPC-Methoden ausgeführt.

2. Periodisch wird über die kRPC-Schnittstelle die aktuelle Antriebsstärke ermittelt, in das UDP basierte Protokoll übersetzt und an das ZedBoard gesendet.

Der Adapter wird in virtuellen Maschine ausgeführt da kRPC einige Abhängigkeiten besitzt die unter Windows nur schwer zu befriedigen sind. Für den Prototypen wird daher auf GNU/-Linux zurück gegriffen.

C.1.2. ZedBoard

Das ZedBoard besteht aus zwei Prozessoren der ARM-Architektur sowie einem FPGA-Modul. Da die Ethernet-Schnittstelle auf dem ZedBoard mit den Prozessoren verbunden ist, findet die Verarbeitung der Pakete in Software statt. Dazu wird auf den Prozessoren das Realzeitbetriebssystem FreeRTOS verwendet. Die Ansteuerung der LEDs, Schalter und Knöpfe erfolgt mithilfe

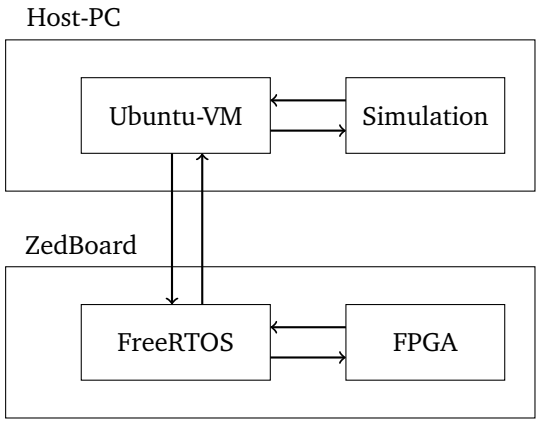


Abbildung C.1.: Kommunikationsprototyp

des FPGA-Moduls, welches Schnittstellen für die Software zur Verfügung stellt. Die Entwicklung der Hard- und Software des Prototypen hat in Vivado und dem Vivado SDK stattgefunden. Mit Vivado wurde eine Verbindung der Knöpfe, Schalter und LEDs mit den Mikroprozessoren erstellt. Auf Basis dieser so generierten Hardwarekonfiguration wurden anschließend zwei Software-Projekte erstellt:

FSBL: Das First Stage Boot Loader (FSBL) dient zum Initialisieren und Starten des ZedBoard von der SD-Karte. Dieses Projekt kann mit Unterstützung des SDK halbautomatisch angelegt werden und dient anschließend zur Erzeugung eines Start-Abbildes für die SD-Karte. Dabei wird ein Startbereich, die Hardwarekonfiguration sowie die Software in das Abbild geschrieben und für das ZedBoard aufbereitet. Mit Hilfe des FSBL kann das ZedBoard ohne einen angeschlossenen Computer starten, das FPGA-Modul konfigurieren und die Software ausführen.

Kommunikationsprototyp: Die eigentliche Aufgabe des Prototypen wird in einem zweiten Projekt realisiert. Das Projekt wird mit FreeRTOS realisiert, daher wurde in dem Projekt bereits Gebrauch von Features wie Threads (in FreeRTOS Taskäs genannt), Timern und Mutexes gemacht. Zudem stellt FreeRTOS das Netzwerk-Projekt LwIP zur Verfügung, mit welchem die Netzwerkkommunikation realisiert wird. Die Software initialisiert nach dem Start zuerst die Anbindung an das FPGA-Modul, um Zugriff auf die Hardwareein- und ausgaben zu erhalten, und die Netzwerkschnittstelle. Dabei übernimmt LwIP die Autonegotiation und die Erstellung des UDP/IP-Stacks mit einer über die Schalter einstellbaren IP-Adresse. Anschließend werden Tasks für das Senden und Empfangen von Paketen erstellt, welche die empfangenen Daten per LED dauerhaft anzeigen und die, durch die Knöpfe ermittelten Daten versenden.

C.2. Ablauf / Funktion

Das ZedBoard wird mit dem auf einer SD-Karte abgelegten Abbild initialisiert und richtet anschließend gemäß der Schalterstellung die Netzwerkverbindung ein. War dieser Vorgang erfolgreich, reagiert das Board, falls einer der 5 Positionsknöpfe gedrückt wird. Ein Paket mit Angabe der betätigten Knöpfe (5 Bit) wird erstellt und an den Host-PC übermittelt.

Der Host-PC empfängt die übermittelten Werte und normalisiert diese für die Simulation. Auf diese Weise können yaw und pitch manipuliert, sowie das nächste Staging ausgelöst werden. Für die Kommunikation in die Gegenrichtung erstellt der Host-PC ebenfalls ein Paket mit einem 8 Bit-Wert der auf dem Streifen von 8 LEDs des ZedBoards angezeigt wird. Übertragen wird die Entfernung der Apoapsis zur Oberfläche des Planeten in dessen SOI sich der Satellit befindet. Durch die begrenzte Anzeigemöglichkeit wird nur der Wertebereich von 0 bis 2000000 auf die 8 verfügbaren Bits der Anzeige normiert.

D. Weitere Diagramme

D.1. Klassendiagramme

D.1.1. Simulation

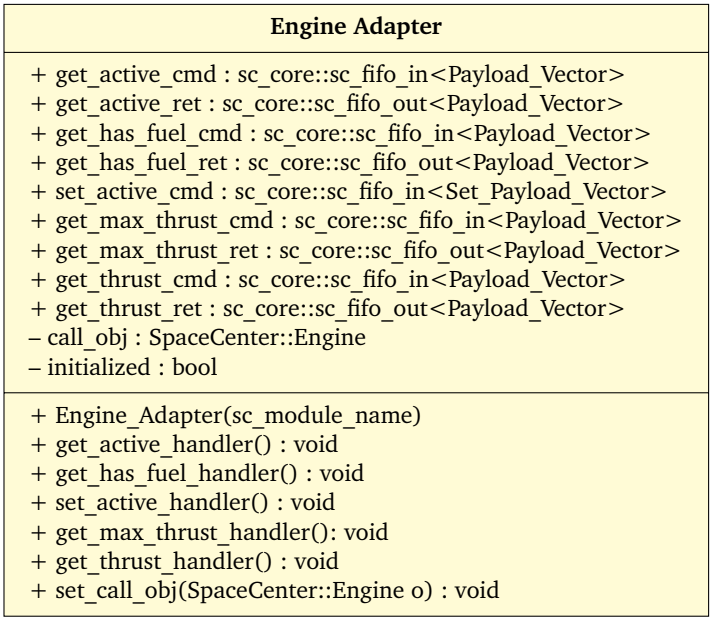


Abbildung D.1.: Klasse Engine Adapter

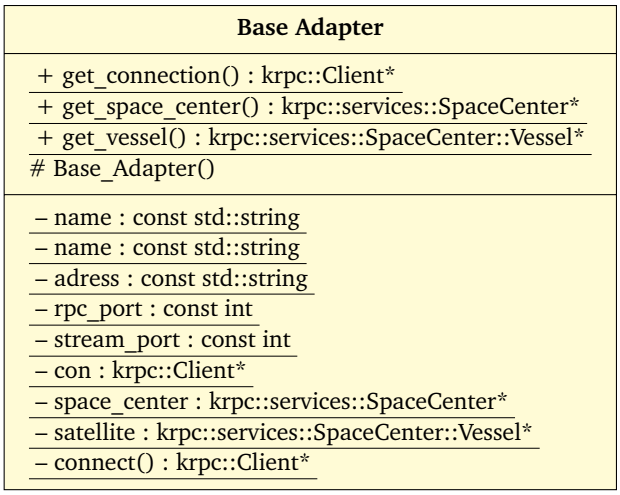


Abbildung D.2.: Klasse Base Adapter

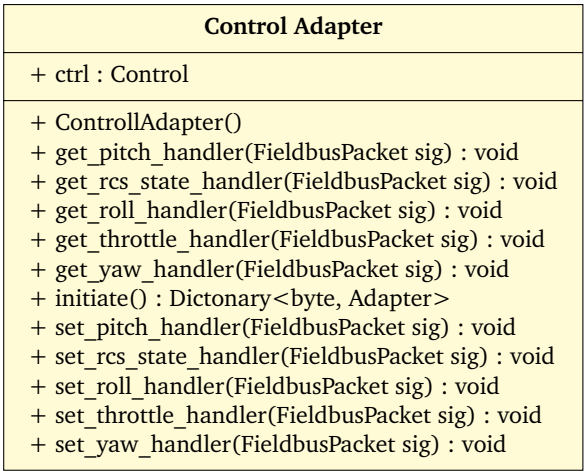


Abbildung D.3.: Klasse Control Adapter

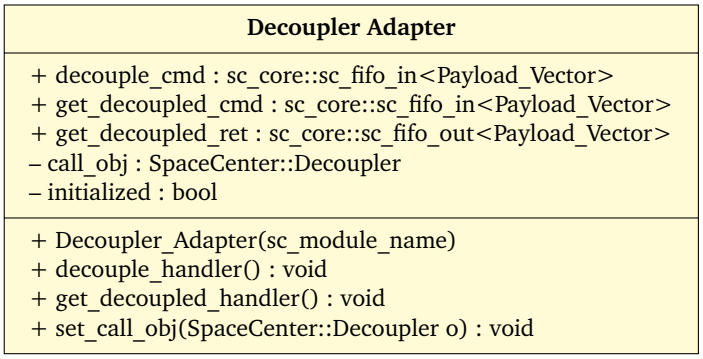


Abbildung D.4.: Klasse Decoupler Adapter

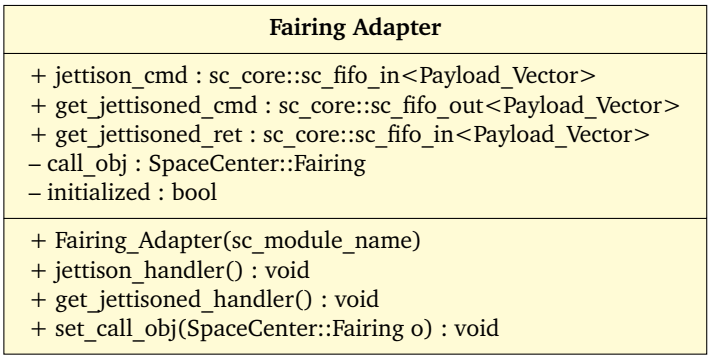


Abbildung D.5.: Klasse Fairing Adapter

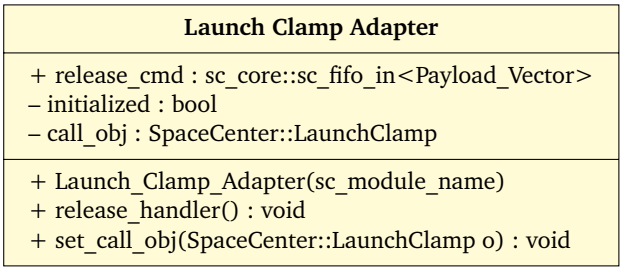


Abbildung D.6.: Klasse Launch-Clamp Adapter

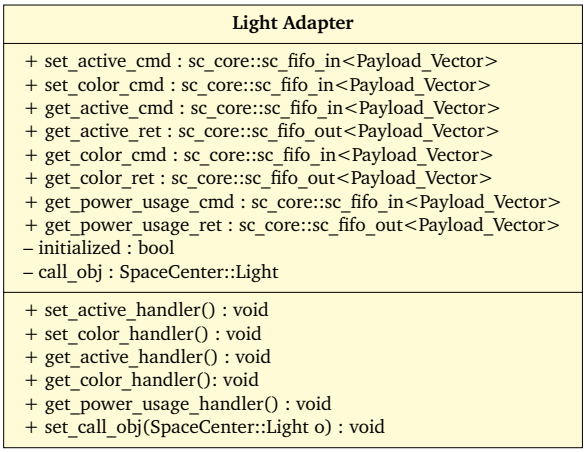


Abbildung D.7.: Klasse Light Adapter

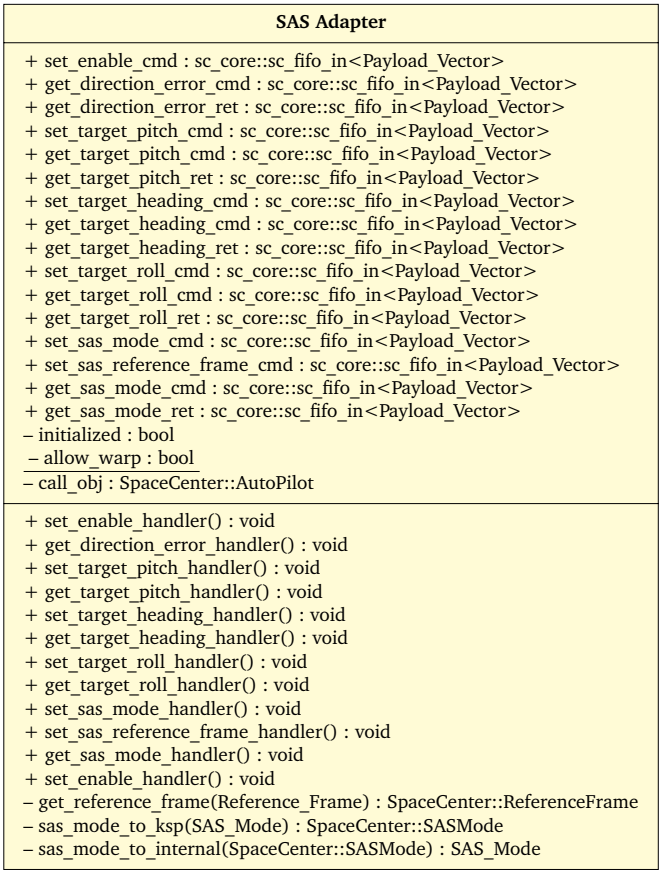


Abbildung D.8.: Klasse SAS Adapter

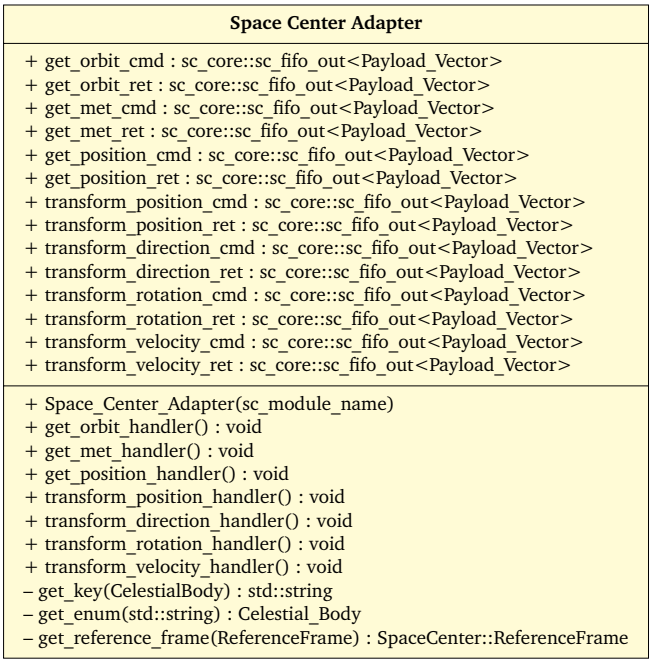


Abbildung D.9.: Klasse Space Center Adapter

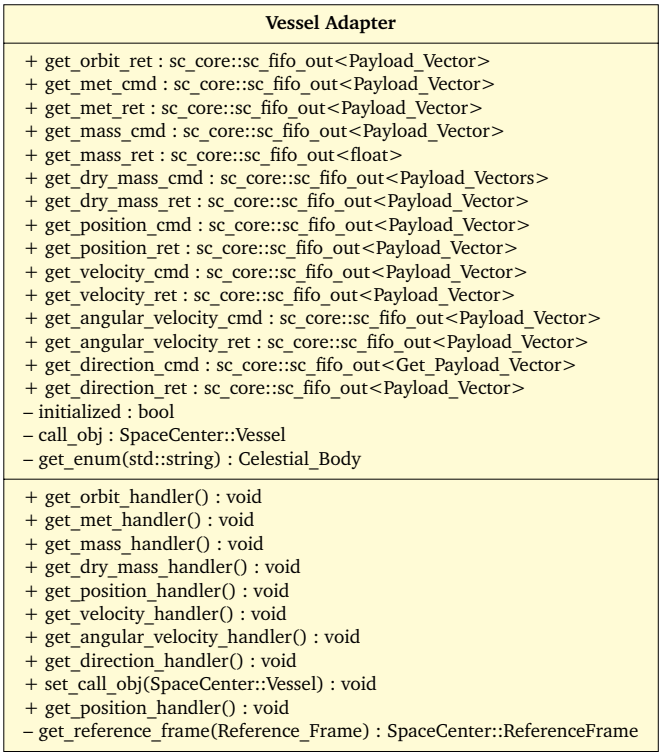


Abbildung D.10.: Klasse Vessel Adapter

D.1.2. Klassendiagramme der Satellitensteuerung

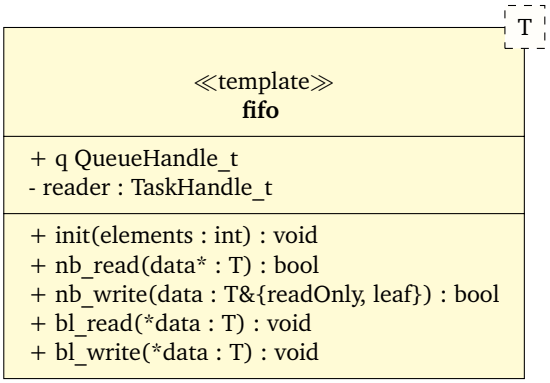


Abbildung D.11.: Fifo-Abstraktionsklasse für FreeRTOS.

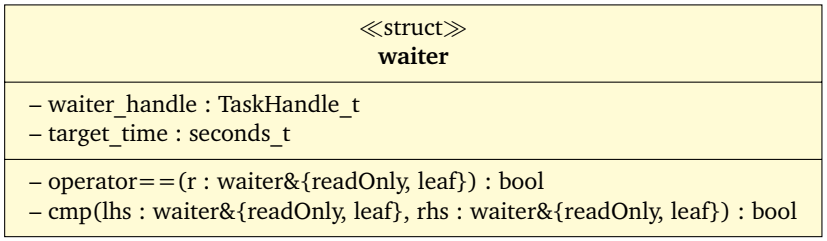


Abbildung D.12.: Waiter Struktur zur Einreihung der wartenden Tasks.

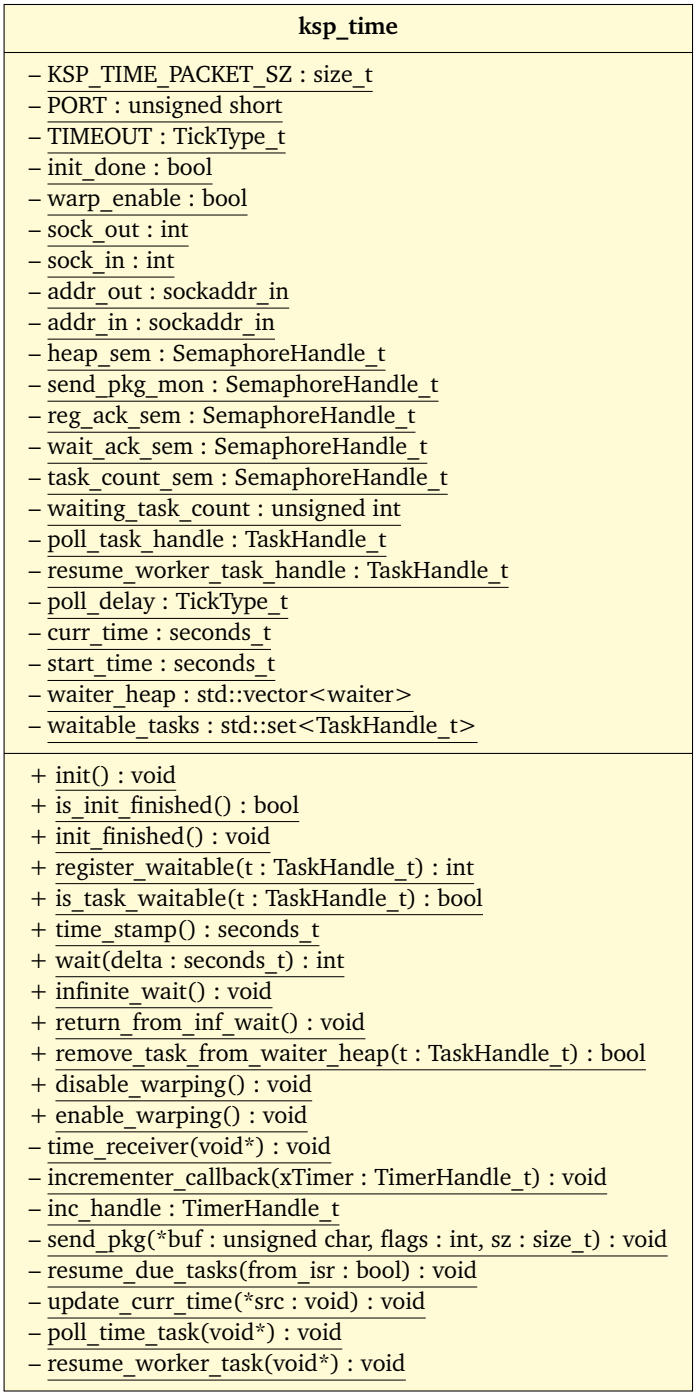


Abbildung D.13.: ksp_time: Klasse zur Zeitsynchronisation mit der Simulation.

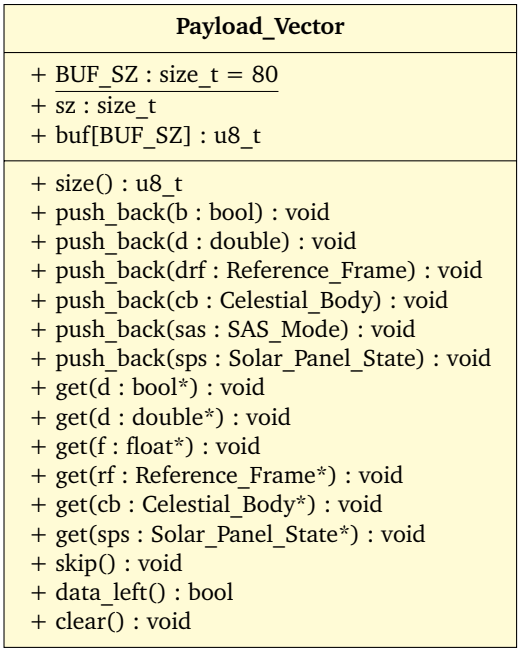


Abbildung D.14.: Variation der Payload_Vector-Klasse für das Zedboard.

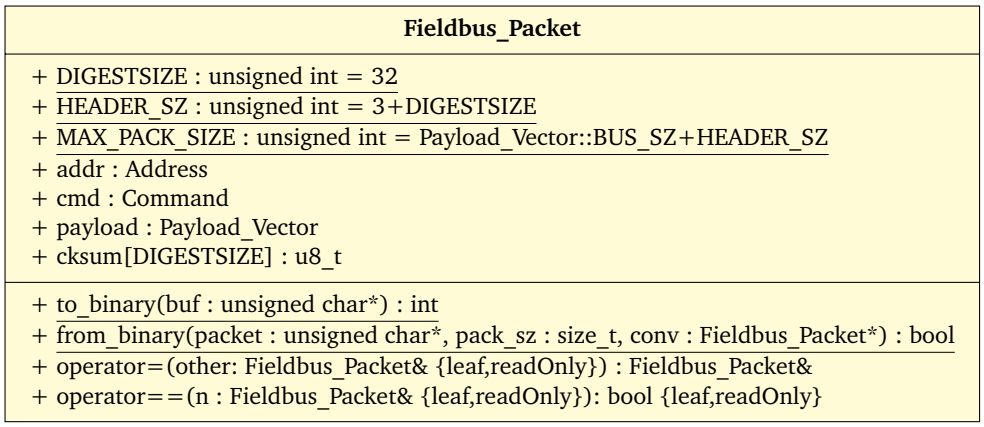


Abbildung D.15.: Variation der Fieldbus_Packet-Klasse für das Zedboard.

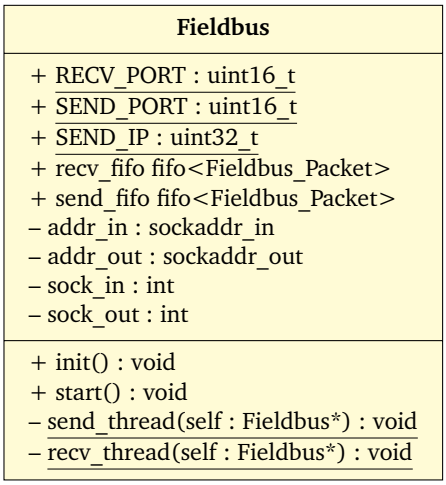


Abbildung D.16.: Fieldbus-Klasse.

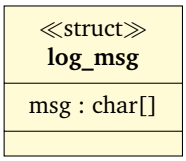


Abbildung D.17.: Datenstruktur für Logmessages.

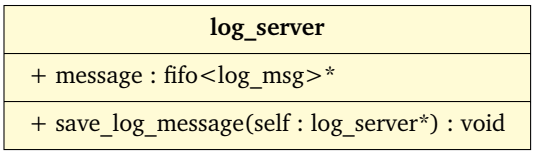


Abbildung D.18.: Logserver Sendermodul.

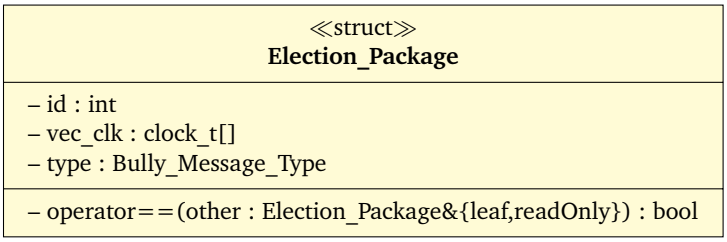


Abbildung D.19.: Election_Package: Datenstruktur, mit der sich die Master_Selector-Klassen austauschen.

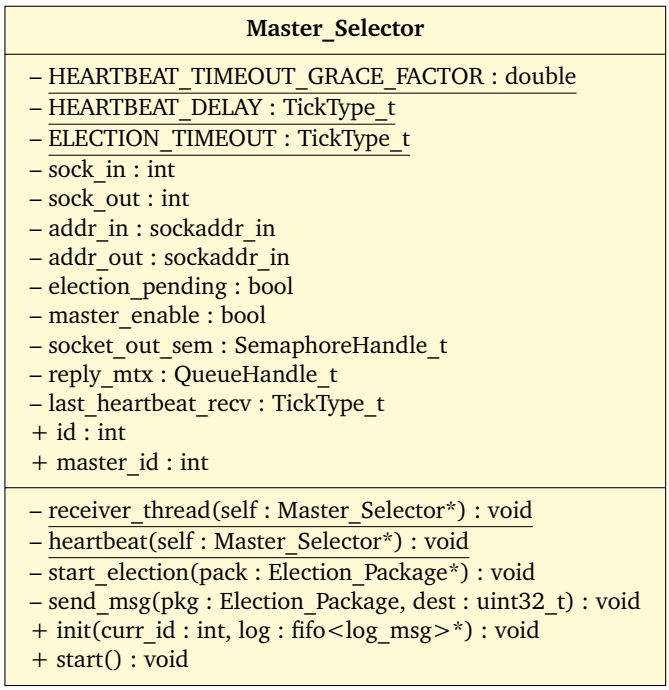


Abbildung D.20.: Master_Selector: Implementierung des Bully-Algorithmus.

Redundancy
+ NUM_INSTANCES : int = 4 + UDP_RECV_PORT : unsigned short = 50011
+ id2ip(id : char) : uint32_t

Abbildung D.21.: Hilffunktionen für Redundanz-Maßnahmen, u.a. Gesamtanzahl der Satellitensteuerungsinstanzen und Funktion der bestimmung der IP anhand der ID.

D.1.3. Fehlerinjektion

User Interface
fi_controller : std::shared_ptr<Fault_Injector_Controller> # receiver : FI_UI_Commands_Receiver # uicp_thread : boost::thread* # uic_thread : boost::thread*
+ User_Interface() + User_Interface() + set_fi_controller(std::shared_ptr<Fault_Injector_Controller> fi_controller) : void + send_fault_state_change(Fault_State_Change fault_state_change) : virtual void + send_fault_experiment_state_change(int fault_experiment_id, int new_completed_value) : virtual void + send_new_fault_injection_state(bool fi_state) : virtual void + send_mission_phase_change(int phase) : virtual void + send_environment_signal(Fieldbus_Packet& pkg) : virtual void + send_control_signal(Fieldbus_Packet& pkg) : virtual void + send_satellite_log_message(std::string message) : virtual void + send_faultinjection_log_message(std::string message) : virtual void + send_faultinjection_log_message(Logging_Message message) : virtual void + receive_ui_command_packet_thread : void + receive_ui_command(Fault_Injection_Command_Packet& pkg) : virtual void + receive_ui_command_thread : void + receive_command(UI_Command_ID cmd) : virtual void + send_command_response() : virtual void + send_all_fault_states() : virtual void

Abbildung D.22.: Klasse User Interface

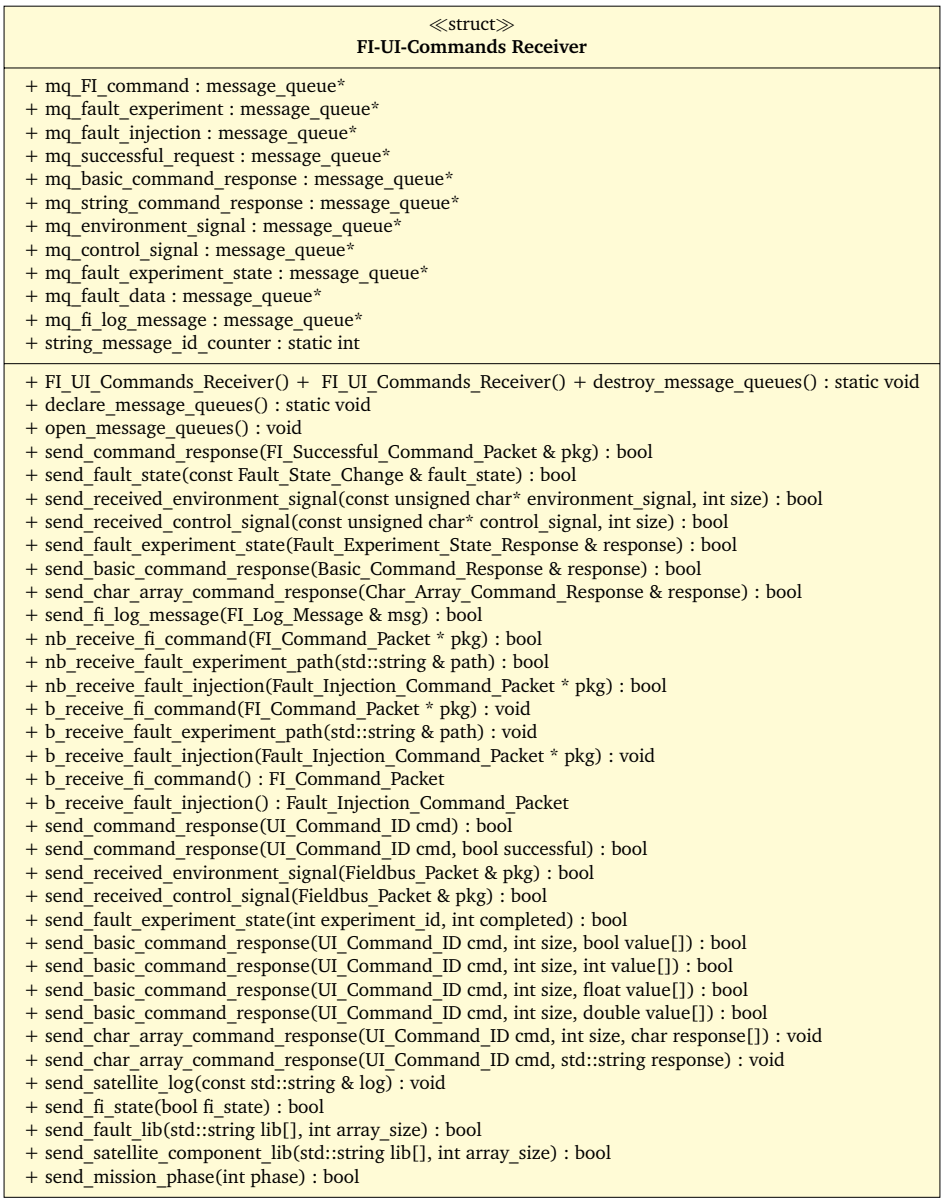


Abbildung D.23.: Klasse FI_UI_Commands_Receiver

≪struct≫
FI-UI-Commands Sender

+ mq_FI_command : message_queue*
+ mq_fault_experiment : message_queue*
+ mq_fault_injection : message_queue*
+ mq_successful_request : message_queue*
+ mq_basic_command_response : message_queue*
+ mq_string_command_response : message_queue*
+ mq_environment_signal : message_queue*
+ mq_control_signal : message_queue*
+ mq_fault_experiment_state : message_queue*
+ mq_fault_data : message_queue*
+ mq_fi_log_message : message_queue*

+ FI_UI_Commands_Sender() + FI_UI_Commands_Sender() + destroy_message_queues() : static void
+ open_message_queues() : void
+ send_fi_command(FI_Command_Packet & pkg) : bool
+ send_fault_experiment_path(char path[],int size) : bool
+ send_fault_injection(Fault_Injection_Command_Packet pkg) : bool
+ nb_receive_command_response(FI_Successful_Command_Packet * pkg) : bool
+ nb_receive_fault_state(Fault_State_Change * fault_state) : bool
+ nb_receive_received_environment_signal(unsigned char environment_signal[200]) : bool
+ nb_receive_received_control_signal(unsigned char control_signal[200]) : bool
+ nb_receive_fault_experiment_state(Fault_Experiment_State_Response * response) : bool
+ nb_receive_basic_command_response(Basic_Command_Response * response) : bool
+ nb_receive_char_array_command_response(Char_Array_Command_Response * response) : bool
+ nb_receive_fi_log_message(FI_Log_Message * msg) : bool
+ nb_receive_received_control_signal(bool& success) : Fieldbus_Packet
+ nb_receive_received_environment_signal(bool& success) : Fieldbus_Packet
+ b_receive_command_response(FI_Successful_Command_Packet * pkg) : void
+ b_receive_fault_state(Fault_State_Change * fault_state) : void
+ b_receive_received_environment_signal(unsigned char environment_signal[200]) : void
+ b_receive_received_control_signal(unsigned char control_signal[200]) : void
+ b_receive_fault_experiment_state(Fault_Experiment_State_Response * response) : void
+ b_receive_basic_command_response(Basic_Command_Response * response) : void
+ b_receive_char_array_command_response(Char_Array_Command_Response * response) : void
+ b_receive_fi_log_message(FI_Log_Message * msg) : void
+ b_receive_received_control_signal() : Fieldbus_Packet
+ b_receive_received_control_fieldbus_packet(Fieldbus_Packet * pkg) : void
+ b_receive_received_environment_signal() : Fieldbus_Packet
+ b_receive_received_environment_fieldbus_packet(Fieldbus_Packet * pkg) : void
+ send_fi_command(UI_Command_ID cmd) : bool
+ send_fi_command(UI_Command_ID cmd, bool fi_state) : bool
+ send_fi_command(UI_Command_ID cmd, mission_phase new_phase) : bool
+ send_fi_command(UI_Command_ID cmd, int aborted_experiment_id) : bool
+ send_fault_experiment_path(std::string & path) : bool
+ send_fault_injection(Fault<bool> fault) : bool
+ send_fault_injection(Fault<bool> fault, Activation_Pattern<bool> activation) : bool
+ send_fault_injection(Fault<bool> fault, Activation_Pattern<int> activation) : bool
+ send_fault_injection(Fault<bool> fault, Activation_Pattern<float> activation) : bool
+ send_fault_injection(Fault<bool> fault, Activation_Pattern<double> activation) : bool
+ send_fault_injection(Fault<int> fault) : bool
+ send_fault_injection(Fault<int> fault, Activation_Pattern<bool> activation) : bool
+ send_fault_injection(Fault<int> fault, Activation_Pattern<int> activation) : bool
+ send_fault_injection(Fault<int> fault, Activation_Pattern<float> activation) : bool
+ send_fault_injection(Fault<int> fault, Activation_Pattern<double> activation) : bool
+ send_fault_injection(Fault<float> fault) : bool
+ send_fault_injection(Fault<float> fault, Activation_Pattern<bool> activation) : bool
+ send_fault_injection(Fault<float> fault, Activation_Pattern<int> activation) : bool
+ send_fault_injection(Fault<float> fault, Activation_Pattern<float> activation) : bool
+ send_fault_injection(Fault<float> fault, Activation_Pattern<double> activation) : bool
+ send_fault_injection(Fault<double> fault) : bool
+ send_fault_injection(Fault<double> fault, Activation_Pattern<bool> activation) : bool
+ send_fault_injection(Fault<double> fault, Activation_Pattern<int> activation) : bool
+ send_fault_injection(Fault<double> fault, Activation_Pattern<float> activation) : bool
+ send_fault_injection(Fault<double> fault, Activation_Pattern<double> activation) : bool
+ send_abort_fault_injection(Fault<bool> fault) : bool
+ send_abort_fault_injection(Fault<int> fault) : bool
+ send_abort_fault_injection(Fault<float> fault) : bool
+ send_abort_fault_injection(Fault<double> fault) : bool
+ send_abort_fault_experiment(int experiment_id) : bool
+ send_print_all_fault_experiments() : bool
+ send_print_all_fault_states() : bool
+ send_set_fi_state(bool fi_state) : bool
+ send_print_fi_state() : bool
+ send_print_fault_lib() : bool
+ send_print_satellite_component_lib() : bool
+ send_switch_mission_phase(mission_phase phase) : bool
+ send_print_mission_phase() : bool
+ send_reset_fi() : bool

Abbildung D.24.: Klasse FI_UI_Commands_Sender

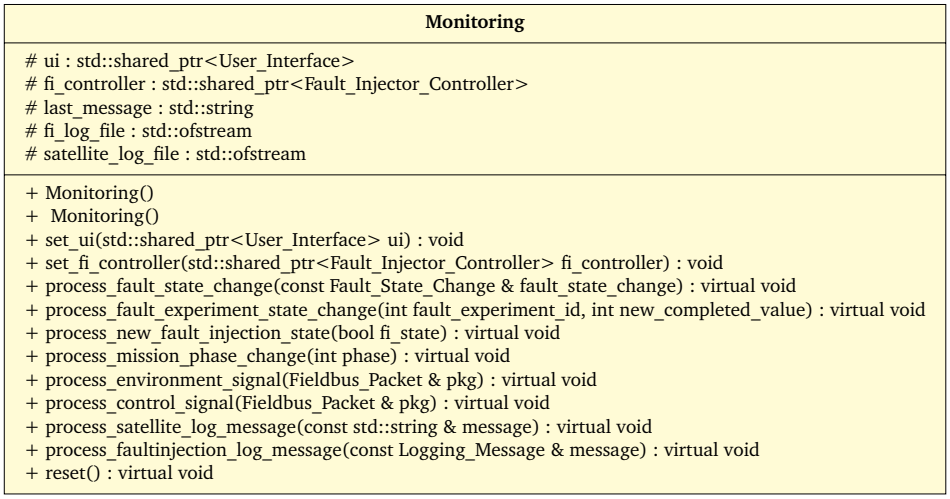


Abbildung D.25.: Klasse Monitoring

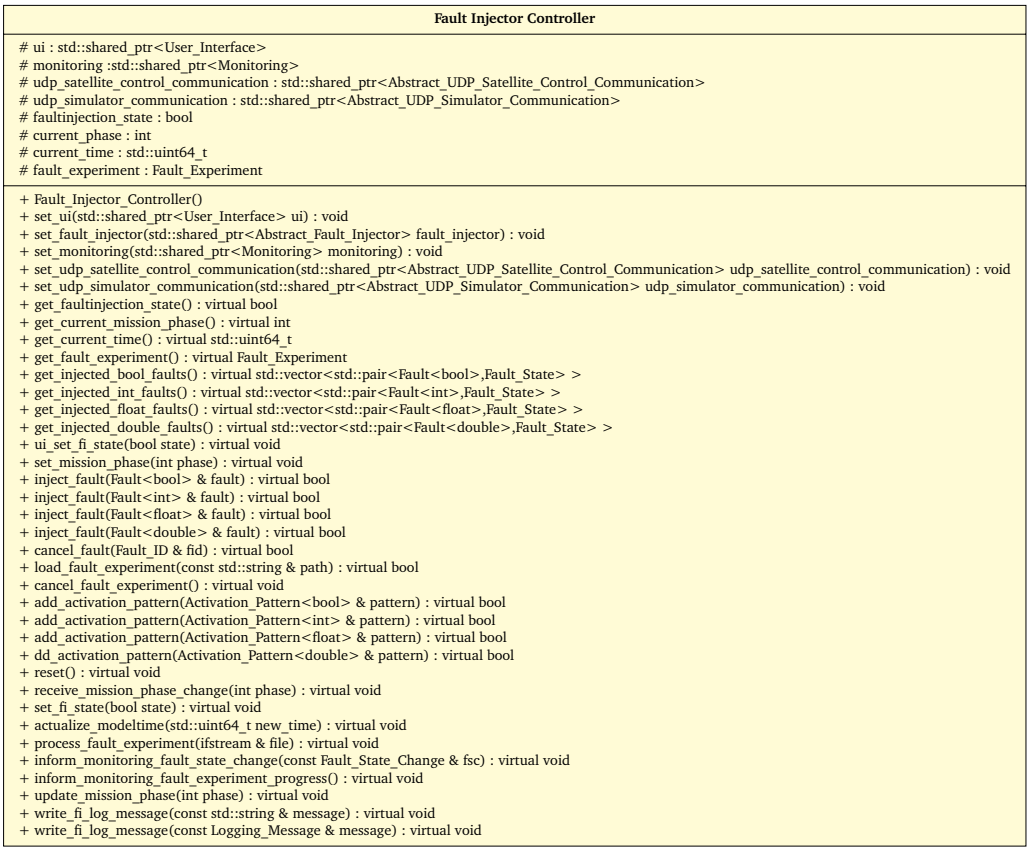


Abbildung D.26.: Klasse Fault Injector Controller

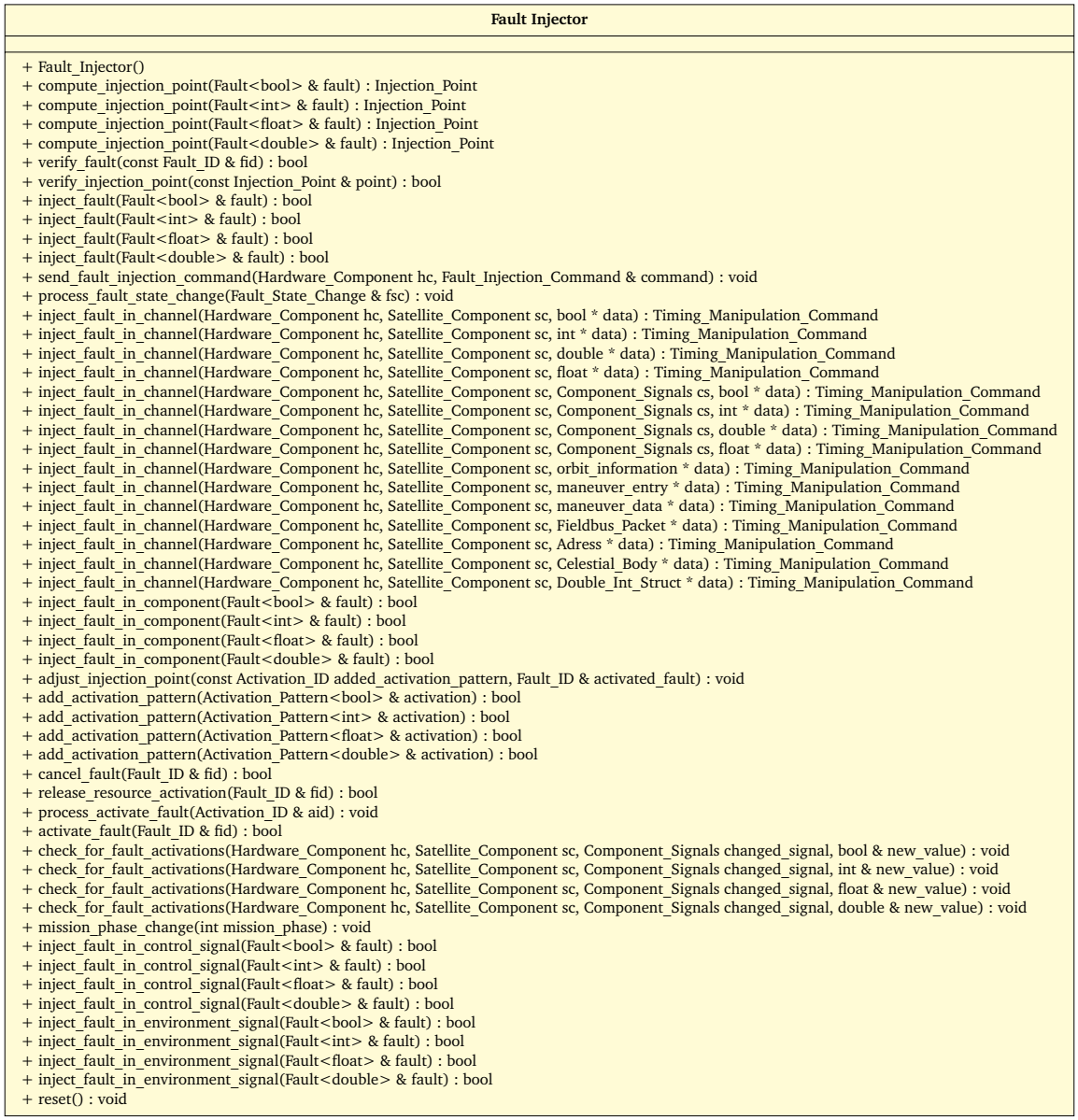


Abbildung D.27.: Klasse Fault_Injector

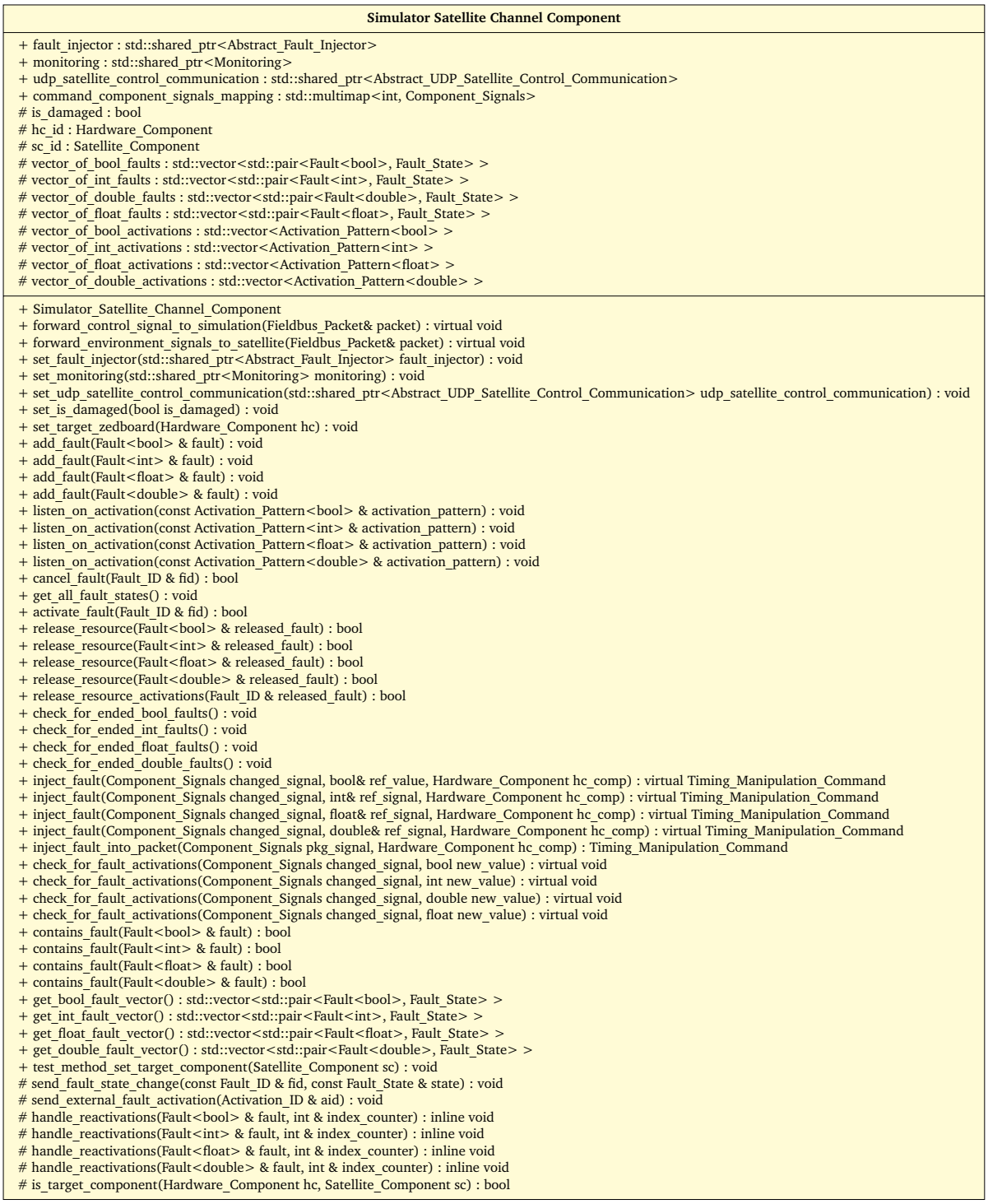


Abbildung D.28.: Klasse Simulator_Satellite_Channel_Component

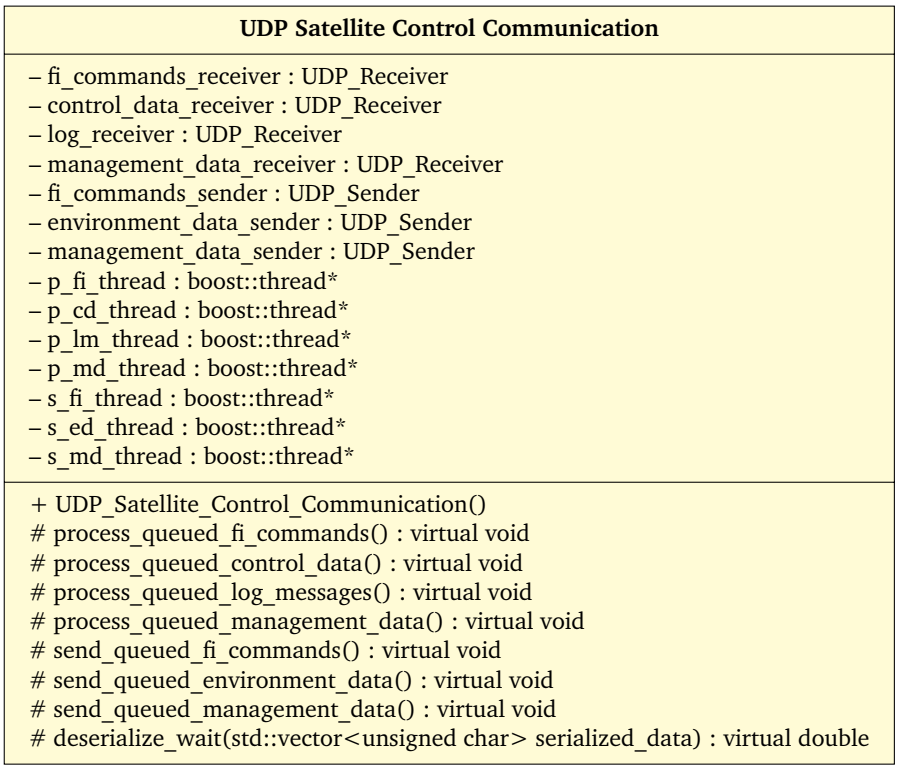


Abbildung D.29.: Klasse UDP_Satellite_Control_Communication

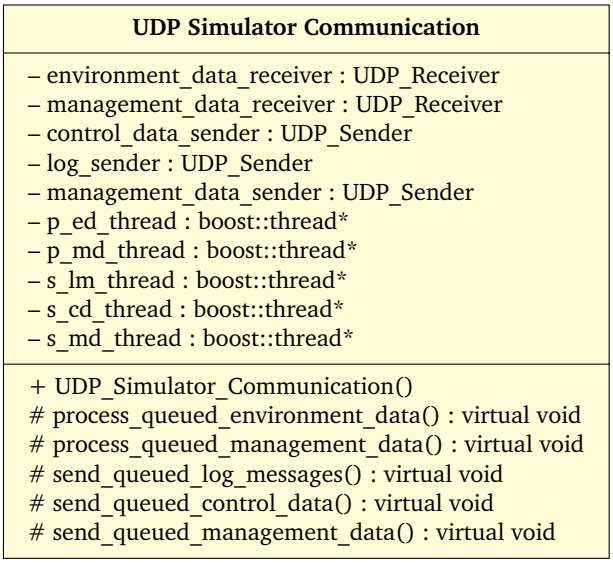


Abbildung D.30.: Klasse UDP_Simulator_Communication

D.2. Sequenzdiagramme

D.2.1. Simulation

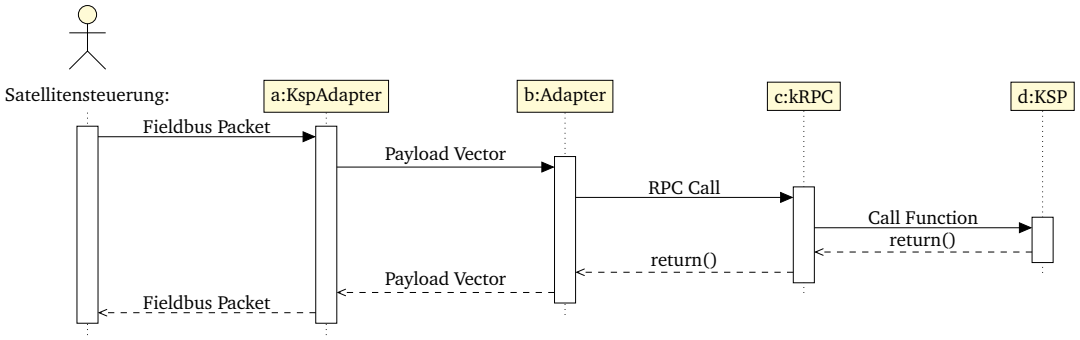


Abbildung D.31.: Sequenzdiagramm Sim

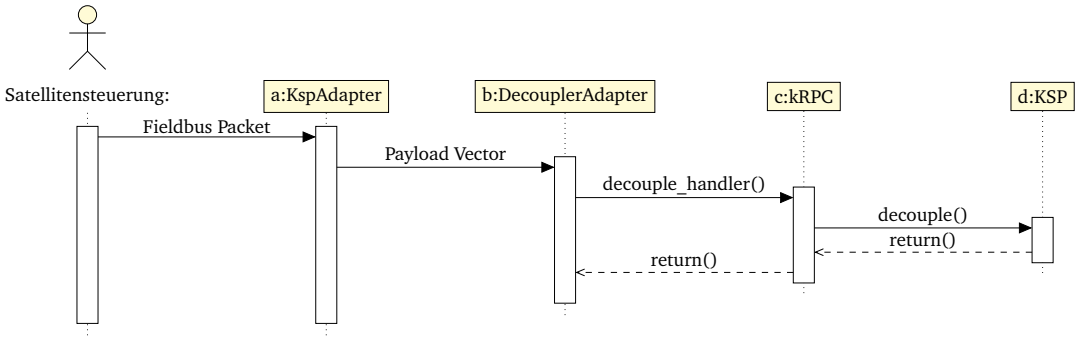


Abbildung D.32.: Sequenzdiagramm Decouple

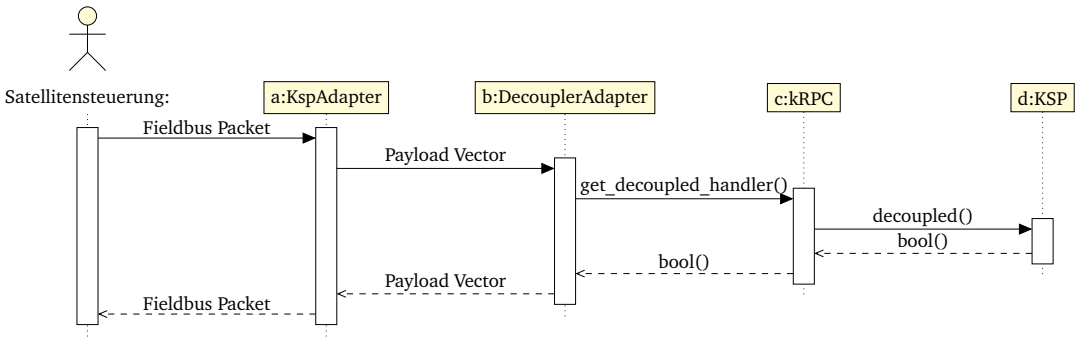


Abbildung D.33.: Sequenzdiagramm Sim

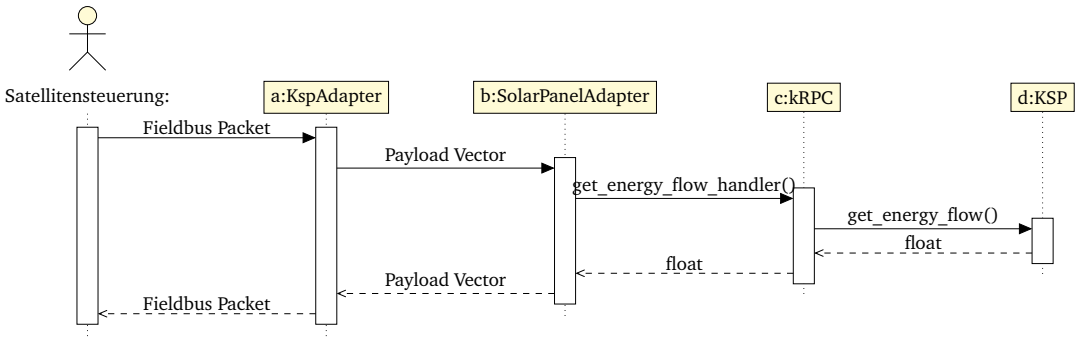


Abbildung D.34.: Sequenzdiagramm Sim

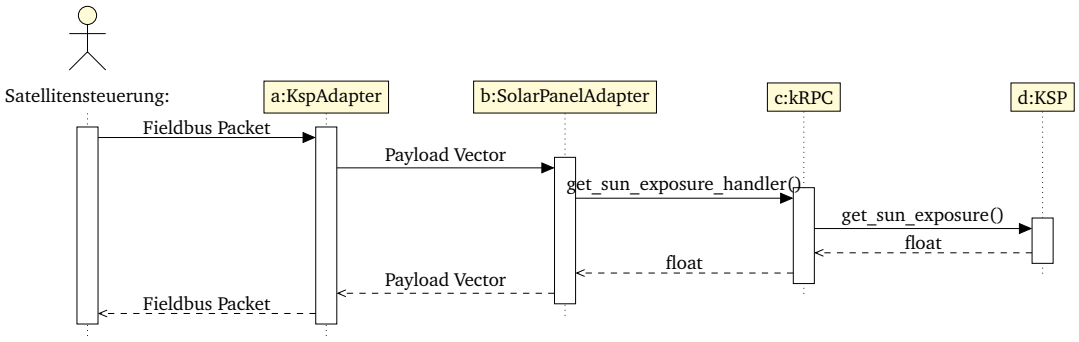


Abbildung D.35.: Sequenzdiagramm Sim

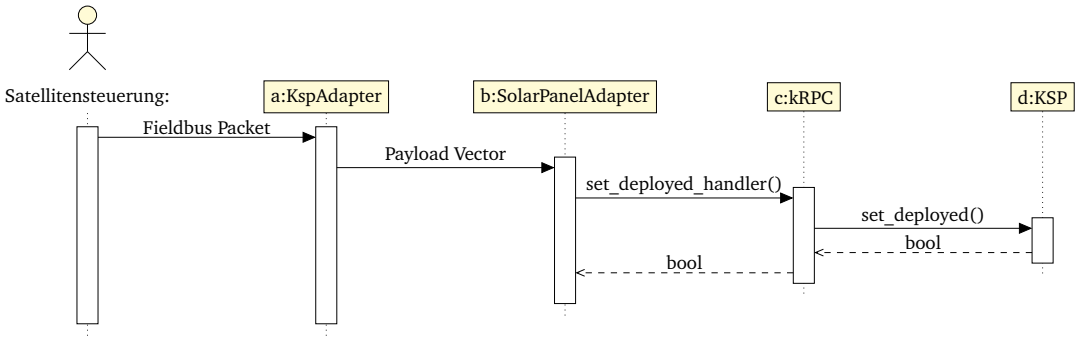


Abbildung D.36.: Sequenzdiagramm Sim

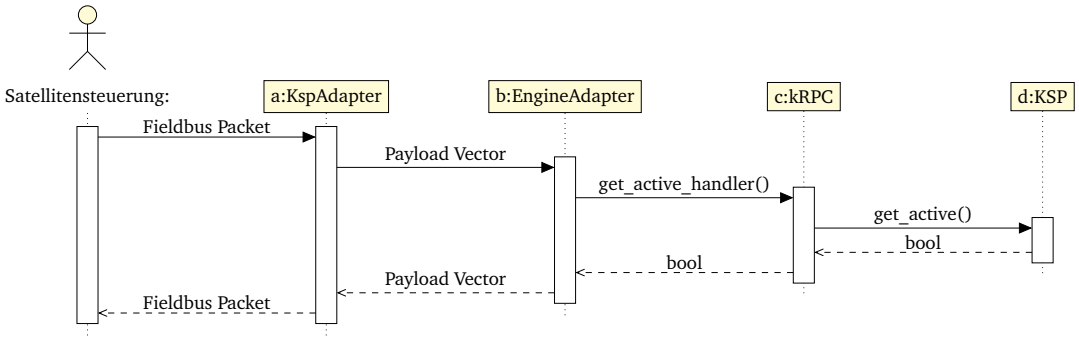


Abbildung D.37.: Sequenzdiagramm Sim

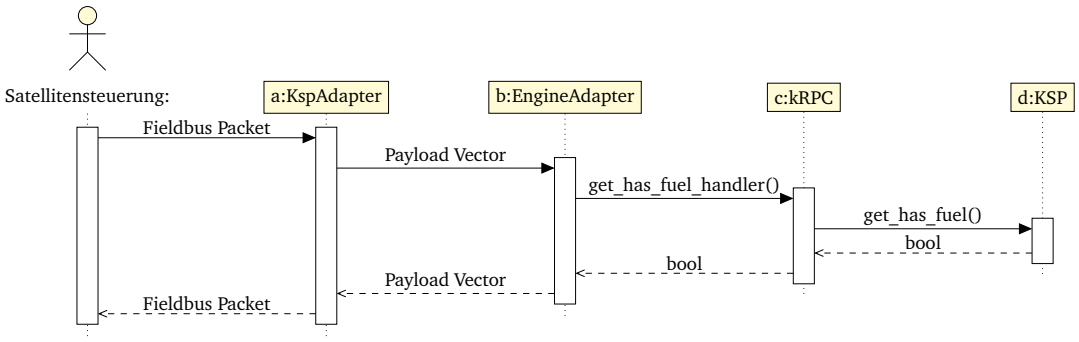


Abbildung D.38.: Sequenzdiagramm Sim

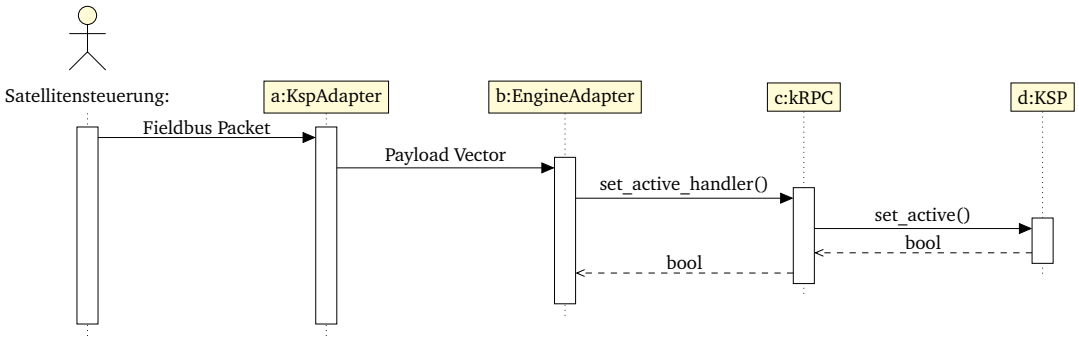


Abbildung D.39.: Sequenzdiagramm Sim

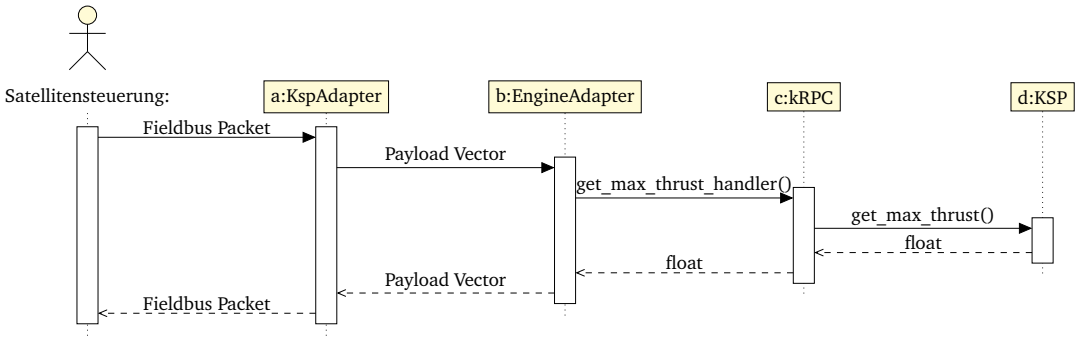


Abbildung D.40.: Sequenzdiagramm Sim

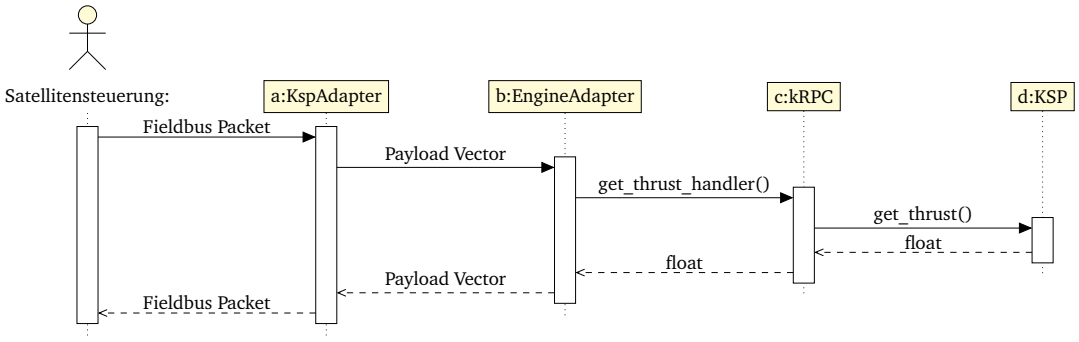


Abbildung D.41.: Sequenzdiagramm Sim

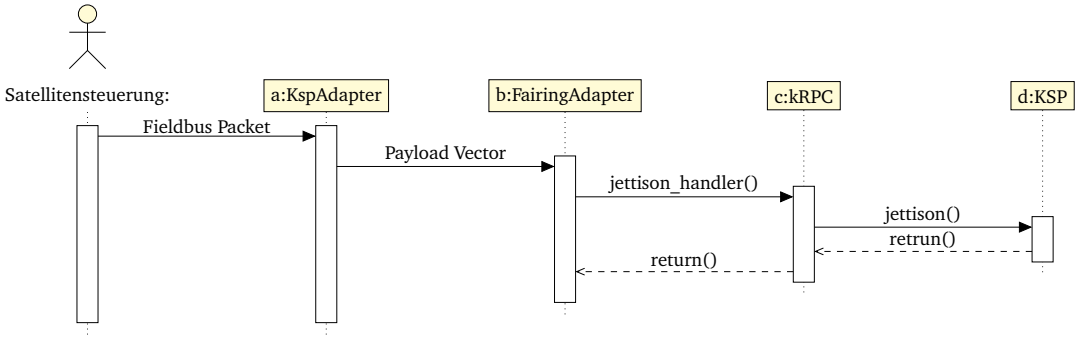


Abbildung D.42.: Sequenzdiagramm Sim

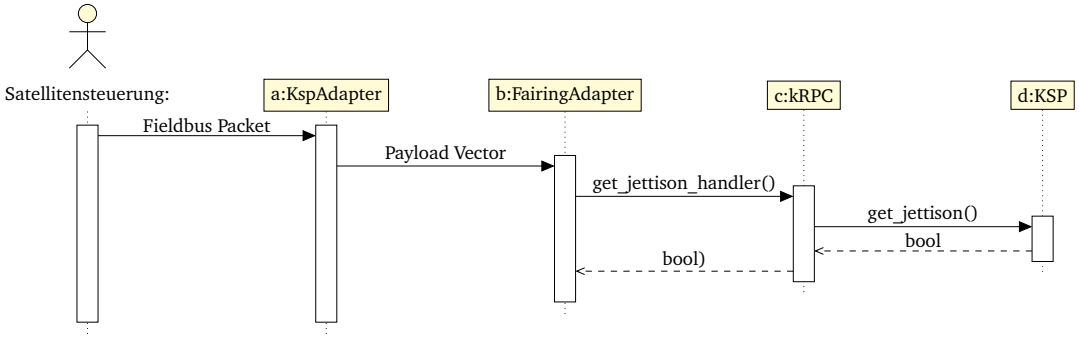


Abbildung D.43.: Sequenzdiagramm Sim

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

ACS Attitude Control Subsystem. 6

AMBA Advanced Microcontroller Bus Architecture. 59

API Application Programming Interface. 15, 60, 157, 158, 231

AXI Advanced eXtensible Interface. 59

EBNF Erweiterte Backus-Naur-Form. 147, *Glossar:* EBNF

FI Fehlerinjektion. 40

FPGA Field Programmable Gate Array. 6, 16, 21, 53, 57–59, 86, 92, 96–98, 116, 119–121, 135–140, 144–151, 188, 190–193, 195, 196, 199, 209, 211–213, 215, 218, 219, 225, 230, 242, 264, 272

FSBL First Stage Boot Loader. 242

HLTP High-Level Test Plan. 69, 71

IDE Integrated Development Environment. 60

IF Interface. 87–89, 155

IP Intelectual Property. 59

KSP Kerbal Space Program.. ii, 2, 12–15, 21, 22, 55, 60, 87, 89, 112, 119, 155, 157–159, 161, 162, 176, 222, 227, 228, 230, 231, 239, 241, 274, *Glossar:* KSP

LEO Low Earth Orbit. 239, *Glossar:* LEO

MDVE Model-based Development & Verification Environment. 6

MMU Memory Management Unit. 121

MPU Memory Protection Unit. 121

PG Guardian Projektgruppe Guardian. ii, 2, 7, 64, 65, 77, 230, 235, 263

PL Programmable Logic. 272

RTL Register Transfer Level. 59

SLD System Level Design. ii, 62, 232

SoC System on Chip. 272

SOI Sphere of Influence.. 83, 123, 243, *Glossar:* Sphere of Influence

TLM Transaction Level Model. 113

TLM Transaction Level Modelling. 57

TMR Triple Modular Redundancy. 130

UI User Interface. 153

VHDL Very High Speed Integrated Circuit Hardware Description Language. 16, 19–21, 57, 59, 186, 274

ZedBoard Xilinx Zynq-7000. ii, 2–5, 18, 21, 25, 40, 42, 43, 54–57, 59, 60, 100, 101, 113, 118, 128–130, 135, 136, 138–140, 144, 145, 147, 153–155, 161, 167, 180, 181, 185, 194, 196, 200, 206, 218, 222, 223, 230, 233, 239, 241, 242, 272

Glossar

ZedBoard Das Xilinx Zynq-7000 basiert auf der Xilinx All Programmable System on Chip (SoC)-Architektur. Es ist mit einem Dual-core ARM Coretex-A9, sowie einem Artix-7 FPGA als Programmable Logic (PL) ausgestattet.. ii, 2–5, 18, 21, 25, 40, 42, 43, 54–57, 59, 60, 100, 101, 113, 118, 128–130, 135, 136, 138–140, 144, 145, 147, 153–155, 161, 180, 200, 206, 218, 222, 223, 230, 233, 239, 241, 242, 272

Aktivierungsmuster Aktivierungsmuster geben an, zu welchem Zeitpunkt ein Fehler injiziert werden soll. Sie beschreiben die Menge A des FARM Konzepts.. 16, 17, 144–151, 188, 190, 194–196, 199, 200, 203, 209, 211–219

Apoapsis Punkt eines Orbits mit dem größten Abstand zum zentralen Körper.. 83, 110, 123, 128

Bodenstation Von der Bodenstation aus könnte der Satellit ferngesteuert werden. Bisher wurde noch keine Entscheidung getroffen, ob und in welchem Umfang diese in diesem Projekt berücksichtigt wird. Siehe auch Kerbal Space Center. 50, 57

Co-Simulation Bei einer Co-Simulation werden zwei Subsysteme simuliert, sodass diese Subsysteme als Blackbox agieren und untereinander Daten austauschen.. v, 186, 239

EBNF Erweiterte Backus-Naur-Form. Das ist eine formale Metasyntax zur Beschreibung von kontextfreien Grammatiken. 147

exzentrische Anomalie Phasenwinkel bezüglich eines Orbits. In der Himmelsmechanik unterscheidet man zwischen wahrer, exzentrischer und mittlerer Anomalie. . 124

FARM Das FARM Konzept ist die Datenklassifizierung eines Fehlerinjektionssystems.. 17, 144

Fehlerexperiment Ein Fehlerexperiment ist ein sequenziell abgearbeitetes Script von Fehlerinjektionen.. 40, 42, 142, 153, 190

Fehlerinjektion Die Fehlerinjektion hat die Aufgabe, Fehler in das System einzuspeisen, um die Fehlertoleranz der Satellitensteuerung zu testen.. 15, 16, 21, 23, 48, 51–54, 131, 135, 137, 138, 153, 188, 197, 203, 230

Fehlerinjektionskomponente Hiermit ist – je nach Kontext – eine Komponente im System oder auf dem ZedBoard gemeint, die jeweils die dortige Kommunikation beeinflussen kann.. 135–139

FI-PC Auf dem Fehlerinjektions-PC wird in diesem Projekt die Fehlerinjektionskomponente ausgeführt.. 154, 188–191, 195, 204, 205, 207–209, 217, 264

First In, First Out FIFO bezeichnet jegliche Verfahren der Speicherung, bei denen diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden.. 121, 156, 176

Golden Run Ein Referenzlauf bei der Fehlerinjektion der das gewünschte Verhalten liefert. 17

High Level Synthese High Level Synthesen sind Übersetzungen von einem abstraktem Modell in eine konkrete Implementierung. Zum Beispiel kann eine High Level Synthese die Übersetzung eines SystemC Modells in konkrete VHDL - Komponenten bedeuten. 232

HIL Das Hardwaremodell eines eingebetteten Systems wird in einer Simulation des Umweltsystems in einer Schleife ausgeführt.. 197, 222

Host-PC Als Host-PC wird ein Rechnersystem bezeichnet, auf dem sowohl die Simulations-, als auch die Fehlerinjektionskomponente ausgeführt werden. In diesem Projekt geschieht das nur auf einer abstrakten Ebene; beide Komponenten haben aus Performanzgründen ihr eigenes Rechnersystem.. 16, 25, 54, 56, 60, 92, 100, 135, 138–141, 144, 148–151, 153, 154, 239, 241–243, 264

Inklination Winkel der Orbitbahnebene im Vergleich zu der Referenzebene des zentralen Himmelskörpers.. 83, 123

Keplerbahnen Keplerbahnen sind Lösungen des Zweikörperproblems der klassischen Himmelsmechanik, bei dem zwei Massepunkte sich um den gemeinsamen Schwerpunkt bewegen. Die Formen der Keplerbahnen sind Kegelschnitte: Kreis, Ellipse, Parabel und Hyperbel.. 110

Kerbal Space Center Ausgangspunkt der Trägerrakete auf Kerbin. 272

Kerbin Name der Erde in KSP.. ii, 22, 230

Kerbol Name der Sonne in KSP.. 107

kRPC Remote Procedure Calls for Kerbal Space Program. kRPC ermöglicht die Kontrolle über KSP, welche mit Hilfe von Skripten außerhalb des Spiels realisiert wird [44].. 60, 87, 102, 103, 155–158, 173, 239, 241

KSP Kerbel Space Program ist ein Computerspiel sowie eine Simulation eines fiktiven Sternsystems mit natürlichen und erbauten Satelliten.. ii, 241

LEO Low Earth Orbit, zu deutsch: Niedrige Erdumlaufbahn. 239

MIL Das Modell eines eingebetteten Systems wird in einer Simulation des Umweltsystems in einer Schleife ausgeführt.. 222

mittlere Anomalie Phasenwinkel bezüglich eines Orbits. In der Himmelsmechanik unterscheidet man zwischen wahrer, exzentrischer und mittlerer Anomalie.. 123, 124

orbitales Referenzsystem Referenzsystem, das als Bezugspunkt die aktuelle Position des Satelliten hat und deren Koordinatenachsen am Orbit des Satelliten ausgerichtet sind.. 124

Periapsis Allgemeine Bezeichnung für denjenigen Punkt auf einer Umlaufbahn um einen zentralen Körper, in dem der Begleiter dem zentralen Objekt am nächsten steht [72].. 83, 123

prograde Ausrichtung in aktuelle Flugrichtung im orbitalen Referenzsystem.. 124

RCS Reaction Control System, ist ein System aus Schubdüsen die zur Navigation im Vakuum gedacht sind.. 79, 107, 108, 112, 113, 128

retrograde Ausrichtung entgegengesetzt der aktuellen Flugrichtung im orbitalen Referenzsystem.. 124

SAS Stability Augmentation System, ist ein automatisches Flugkontrollsystem.. 106

Satellit Objekt in einem Orbit.. 4, 5, 22, 48, 51, 107

Satellitensteuerung Die Satellitensteuerung umfasst in diesem Projekt die gesamte Soft- und Hardware, die der Steuerung des Satelliten dient, unabhängig davon, wie und worauf diese ausgeführt wird.. 23, 33, 48, 50, 54, 57, 165, 180, 220, 230

SIL Das Softwaremodell eines eingebetteten Systems wird in einer Simulation des Umweltsystems in einer Schleife ausgeführt.. 222

Sim-PC Auf dem Simulations-PC wird in diesem Projekt die Simulationsumgebung KSP ausgeführt.. 154

Simulationsumgebung Das Programm, das die Umgebung des Satelliten simuliert.. 7, 24

Simulator Das Teilprojekt Simulator stellt die Schnittstelle im System zur Simulationsumgebung zur Verfügung und ermöglicht so die Anbindung der Satellitensteuerung an das System.. 21, 23, 24, 48, 51, 52, 54, 57, 59, 60, 92, 137, 153, 155, 160, 230

Single Point Of Failure Eine Komponente, dessen Ausfall zum Versagen des gesamten Systems führt.. 101, 129

Sphere of Influence Gravitationaler Einflussbereich eines Himmelskörpers.. 83

Steuerbefehle Befehle, die an Aktuatoren gesendet werden, um den Satelliten zu steuern.. 57, 150, 153, 165

Steuerkurs Ein Ausrichtungswinkel im orbitalen Referenzsystem. Dieser beschreibt die Ausrichtung orthogonal zur Orbitebene. Das heißt, das ein Steuerkurs(engl. heading) von 0 immer zur Orbitebene ausgerichtet liegt. Ein Steuerkurs von 90 ist eine orthogonale Ausrichtung zur Orbitebene. Dieser Winkel geht von -180 bis 180.. 125

System In diesem Projekt der gesamte Aufbau, bestehend aus den Teilsystemen Simulator, Fehlerinjektion und Satellitensteuerung.. 23, 24, 48, 70, 71, 74, 138, 140, 153, 230, 272

SystemC SystemC ist eine Modellierungssprache auf höherem Abstraktionsniveau als herkömmliche Hardwaremodellierungssprachen, wie VHDL oder Verilog.. 7, 19–21, 57, 59, 60, 87, 155, 159, 161–163, 176, 177, 180, 181, 183, 184, 186, 220, 232, 239, 240

UDP User Datagram Protocol. Für die Übertragung von Daten definiertes Netzwerkprotokoll auf der Transportschicht.. 195, 217

Umweltsignale Umweltsignale sind vorverarbeitete Unweltinformationen, wie zum Beispiel Gegenstandserkennung oder unverarbeitete Sensorwerte.. 57, 94, 139, 147, 149–151, 153, 165

wahre Anomalie Phasenwinkel bezüglich eines Orbits. In der Himmelsmechanik unterscheidet man zwischen wahrer, exzentrischer und mittlerer Anomalie unterscheiden. In der Orbitplanung, in diesem Dokument wird die wahre Anomalie benutzt.. 83, 123, 127, 175

Literatur

[1] *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*. System C Standardization Working Group. 2011.

[2] European Space Agency. *SpaceWire - ESA Missions*. URL: http://spacewire.esa.int/content/Missions/ESA.php (besucht am 02. 03. 2017).

[3] Sandeep S. Kulkarni Ali Ebnenasir Reza Hajisheykhi. „Facilitating the design of fault tolerance in transaction level SystemC programs“. In: *Theoretical Computer Science* 496 (2013), 50(19).

[4] KSP API. *KSP API Documentation*. URL: http://anatid.github.io/XML-Documentation-for-the-KSP-API/index.html (besucht am 13. 12. 2016).

[5] KSP API. *KSP API: Planetarium Class Reference*. URL: http://anatid.github.io/XML-Documentation-for-the-KSP-API/class_planetarium.html (besucht am 03. 10. 2017).

[6] KSP API. *KSP API: Vessel Class Reference*. URL: http://anatid.github.io/XML-Documentation-for-the-KSP-API/class_vessel.html (besucht am 26. 09. 2016).

[7] Jean Arlat u. a. „Fault injection for dependability validation: A methodology and some applications“. In: *IEEE Transactions on Software Engineering* 16.2 (1990), S. 166–182.

[8] Luca Benini, Robin Hodgson und Polly Siegel. „System-level power estimation and optimization“. In: *Proceedings of the 1998 international symposium on Low power electronics and design - ISLPED '98* (1998). DOI: 10.1145/280756.280881. URL: http://dx.doi.org/10.1145/280756.280881.

[9] Karavul Berekat. *Projektplanung*. URL: http://www.projektmanagementhandbuch.de/projektplanung/ (besucht am 13. 12. 2016).

[10] Jochen Bergmann. *Funktionsprüfung der Steuerungssoffware intelligenter technischer Produkte*. Prof. Dr. Klaus Bender, 1998.

[11] David C. Black u. a. *SystemC: From the Ground Up*. 2. Aufl. Springer, 2010.

[12] Robert A. Braeunig. *Rocket and Space Technology*. URL: http://www.braeunig.us/space/index.htm (besucht am 08. 12. 2016).

[13] Michel Capderou. *Definition of the mean anomaly*. Springer, 2005.

[14] Pierluigi Civera u. a. „Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits“. In: *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*. IEEE. 2001, S. 250–258.

[15] Stephan Clamage. *Mixing C and C++ Code in the Same Program*. 2011. URL: http://www.oracle.com/technetwork/articles/servers-storage-dev/mixingcandcpluspluscode-305840.html.

[16] Howard Curtis. *Orbital mechanics for engineering students*. Butterworth-Heinemann, 2013.

[17] djungelorm. *Communication Protocol*. URL: https://krpc.github.io/krpc/communication-protocol.html (besucht am 27. 03. 2017).

[18] Klaus Echtle. *Fehlertoleranzverfahren*. Springer, 1990. URL: http://dc.informatik.uni-essen.de/Echtle/all/buch_ftv/index.html.

[19] Klaus Echtle. *Fehlertoleranzverfahren*. Springer, 1990. URL: http://dc.informatik.uni-essen.de/Echtle/all/buch_ftv/index.html.

[20] B612 Foundation. *B612 Senitel Mission*. URL: https://b612foundation.org/sentinel/ (besucht am 13. 12. 2016).

[21] Daniel D. Gajski u. a. *Embedded System Design*. Springer Nature, 2009. DOI: 10.1007/978-1-4419-0504-8. URL: http://dx.doi.org/10.1007/978-1-4419-0504-8.

[22] Ian J. Hayes. „Dynamically Detecting Faults via Integrity Constraints“. In: *Methods, Models and Tools for Fault Tolerance*. Hrsg. von Michael Butler Cliff Jones Alexander Romanovsky Elena Troubitsyna. Springer, 2009.

[23] Werner Horn. *The Three Anomaliesl*. URL: https://www.csun.edu/~hcmth017/master/node14.html (besucht am 31. 08. 2017).

[24] Mei-Chen Hsueh, Timothy K Tsai und Ravishankar K Iyer. „Fault Injection Techniques and Tools“. In: *Computer* 30.4 (1997), S. 75–82.

[25] F Huber u. a. „FPGA based on-board computer system fort he Flying Laptop microsatellite“. In: *Proceedings of the Data System in Aerospace Conference, SP-638, ESA, Naples*. 2007.

[26] Digilent Inc. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. 2012. URL: https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/ (besucht am 14. 06. 2018).

[27] Department of Information Technology at Uppsala University und Department of Computer Science at Aalborg University. *UPPAAL*. 2017. URL: http://www.uppaal.org/.

[28] Michael Jugovac. „Ausarbeitung zum Seminar Softwarebasierte Fehlertoleranz im Wintersemester 2012/2013: Grundlagen der Fehlerinjektion“. In: ().

[29] Frank Kesel. *Modellierung von digitalen System mit SystemC*. Oldenbourg Verlag München, 2012.

[30] Israel Koren und C. Mani Krishna. *Fault-Tolerant Systems*. Elsevier, 2007.

[31] Israel Koren und C. Mani Krishna. *Fault-Tolerant Systems*. Elsevier, 2007.

[32] Kenneth A. LaBel. *Radiation Effects on Electronics 101*. 2004. URL: https://nepp.nasa.gov/DocUploads/392333B0-7A48-4A04-A3A72B0B1DD73343/Rad_Effects_101_WebEx.pdf.

[33] Jet Propulsion Laboratory. *How Many Solar System Bodies*. URL: http://ssd.jpl.nasa.gov/?body_count (besucht am 13. 12. 2016).

[34] Marcello Lajolo, Mihai Lazarescu und Alberto Sangiovanni-Vincentelli. „A compilation-based software estimation scheme for hardware/software co-simulation“. In: *Hardware/Software Codesign, 1999.(CODES'99) Proceedings of the Seventh International Workshop on*. IEEE. 1999, S. 85–89.

[35] Leslie Lamport, Robert Shostak und Marshall Pease. „The Byzantine Generals Problem“. In: *ACM Transactions on Programming Languages and Systems* 4.3 (Juli 1982), S. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: http://dx.doi.org/10.1145/357172.357176.

[36] Wiley J Larson und James Richard Wertz. *Space mission analysis and design*. Techn. Ber. Microcosm, Inc., Torrance, CA (US), 1992.

[37] Real Time Engineers Ltd. *The FreeRTOS Reference Manual*. 2016. URL: http://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf (besucht am 29. 09. 2017).

[38] R. E. Lyons und W. Vanderkulk. „The Use of Triple-Modular Redundancy to Improve Computer Reliability“. In: *IBM Journal of Research and Development* 6.2 (Apr. 1962), S. 200–209. ISSN: 0018-8646. DOI: 10.1147/rd.62.0200. URL: http://dx.doi.org/10.1147/rd.62.0200.

[39] Jon Manning u. a. *The Kerbal Player's Guide*. O'Reilly, 2017.

[40] Kim K. de Groh Maria M Finckenor. *A Researcher's Guide to: Space Environmental Effects*. NASA, 2015. URL: https://www.nasa.gov/sites/default/files/files/NP-2015-03-015-JSC_Space_Environment-ISS-Mini-Book-2015-508.pdf.

[41] AVNet Electronics Marketing. *ZedBoard. (Zynq Evaluation and Development) Hardware User's Guide*. 2012. URL: https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf (besucht am 14. 06. 2018).

[42] Charles Morisset Miaomiao Zhang Zhiming Liu und Anders P. Ravn. „Design and Verification of Fault-Tolerant Components“. In: *Methods, Models and Tools for Fault Tolerance*. Hrsg. von Michael Butler Cliff Jones Alexander Romanovsky Elena Troubitsyna. Springer, 2009.

[43] Microsoft. *Type float*. URL: https://msdn.microsoft.com/en-us/library/hd7199ke.aspx (besucht am 27. 09. 2016).

[44] Curse Mods. *KSP Mods*. URL: https://mods.curse.com/ksp-mods/kerbal/220219-krpc-control-the-game-using-c-c-java-lua-python (besucht am 13. 12. 2016).

[45] OpenAMP. *Github: libmetal*. 2017. URL: https://github.com/OpenAMP/libmetal (besucht am 29. 09. 2017).

[46] Marcel Pockrandt. „Model Checking Memory-Related Properties of Hardware/Software Codesigns“. Dissertation. Technische Universität Berlin, 2014.

[47] Kerbal Space Program. *Addon - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Addon (besucht am 08. 12. 2016).

[48] Kerbal Space Program. *Asteroid - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Asteroid (besucht am 13. 12. 2016).

[49] Kerbal Space Program. *Biome - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Biome (besucht am 08. 12. 2016).

[50] Kerbal Space Program. *Career - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Career (besucht am 08. 12. 2016).

[51] Kerbal Space Program. *Category:Dwarf planets - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Category:Dwarf%5C_planets (besucht am 08. 12. 2016).

[52] Kerbal Space Program. *Gilly - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Gilly (besucht am 08. 12. 2016).

[53] Kerbal Space Program. *http://docuwiki-kspapi.rhcloud.com/#/about*. URL: http://docuwiki-kspapi.rhcloud.com/#/about (besucht am 08. 12. 2016).

[54] Kerbal Space Program. *Kerbal Space Center - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Kerbal_Space%5C_Center (besucht am 08. 12. 2016).

[55] Kerbal Space Program. *Kerbal Space Program*. URL: https://kerbalspaceprogram.com/en/?page_id=7 (besucht am 08. 12. 2016).

[56] Kerbal Space Program. *Kerbal Space Program | Made with Unity*. URL: https://madewith.unity.com/games/kerbal-space-program (besucht am 08. 12. 2016).

[57] Kerbal Space Program. *Kerbol - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Kerbol (besucht am 08. 12. 2016).

[58] Kerbal Space Program. *Kerbol System - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Kerbol_System (besucht am 08. 12. 2016).

[59] Kerbal Space Program. *KSP API: Class List*. URL: http://anatid.github.io/XML-Documentation-for-the-KSP-API/annotated.html (besucht am 08. 12. 2016).

[60] Kerbal Space Program. *Parts - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Parts (besucht am 08. 12. 2016).

[61] Kerbal Space Program. *Resource - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Resource (besucht am 08. 12. 2016).

[62] Kerbal Space Program. *Sandbox - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Sandbox (besucht am 08. 12. 2016).

[63] Kerbal Space Program. *Science - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Science (besucht am 08. 12. 2016).

[64] Kerbal Space Program. *Science mode - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Science_mode (besucht am 08. 12. 2016).

[65] Kerbal Space Program. *Sphere of influence - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Sphere_of_influence (besucht am 08. 12. 2016).

[66] Kerbal Space Program. *Technology tree - Kerbal Space Program Wiki*. URL: http://wiki.kerbalspaceprogram.com/wiki/Technology_tree (besucht am 08. 12. 2016).

[67] Wiki Kerbal Space Program. *Making an asset from start to finish*. URL: http://wiki.kerbalspaceprogram.com/wiki/Tutorial:Making_an_asset_from_start_to_finish (besucht am 13. 12. 2016).

[68] *Redmine*. URL: http://www.redmine.org/ (besucht am 13. 12. 2016).

[69] Santhosh Kumar Rethinagiri u. a. „Hybrid system level power consumption estimation for FPGA-based MPSoC“. In: *2011 IEEE 29th International Conference on Computer Design (ICCD)* (Okt. 2011). DOI: 10.1109/iccd.2011.6081403. URL: http://dx.doi.org/10.1109/ICCD.2011.6081403.

[70] Axel-Tobias Schreiner. „Object oriented programming with ANSI-C“. In: (1993).

[71] Joint Functional Component Command for Space. *SPACEFLIGHT SAFETY HANDBOOK FOR SATELLITE OPERATORS*. 2016. URL: https://www.space-track.org/documents/JSpOC_Spaceflight_Safety_Handbook_For_Operators.pdf.

[72] Spektrum.de. *Periapsis*. Spektrum Akademischer Verlag. 1998. URL: https://www.spektrum.de/lexikon/physik/periapsis/11021 (besucht am 11. 06. 2018).

[73] Squad. *Asteroid Day*. URL: https://kerbal.curseforge.com/projects/asteroid-day (besucht am 13. 12. 2016).

[74] Squad. *Time*. URL: https://wiki.kerbalspaceprogram.com/wiki/Time (besucht am 03. 10. 2017).

[75] Squad. *Time warp*. URL: https://wiki.kerbalspaceprogram.com/wiki/Time_warp (besucht am 03. 10. 2017).

[76] Universität Stuttgart. *The Flying Laptop*. URL: http://www.kleinsatelliten.de/flying_laptop/index.en.html (besucht am 13. 12. 2016).

[77] *SystemC to Timed Automata Transformation Engine*. URL: http://www.sese.tu-berlin.de/menue/ueber_uns/team/paula_herber/state/.

[78] Telemachus. *KSP Telemachus*. URL: https://github.com/KSP-Telemachus/Telemachus/wiki/User-Guide (besucht am 13. 12. 2016).

[79] Telemachus. *KSP Telemachus API-String*. URL: https://github.com/KSP-Telemachus/Telemachus/wiki/API-String (besucht am 13. 12. 2016).

[80] David Tribble. *Incompatibilities Between ISO C and ISO C++*. 5. Aug. 2001. URL: http://david.tribble.com/text/cdiffs.htm.

[81] TriggerAu. *[1.3.x] Kerbal Alarm Clock v3.8.5.0 (May 30)*. URL: https://forum.kerbalspaceprogram.com/index.php?/topic/22809-13x-kerbal-alarm-clock-v3850-may-30/ (besucht am 03. 10. 2017).

[82] Michael Butler Cliff Jones Alexander Romanovsky Elena Troubitsyna, Hrsg. *Methods, Models and Tools for Fault Tolerance*. Springer, 2009.

[83] *UG1144: PetaLinux Tools Documentation - Reference Guide*. Xilinx. Juni 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1144-petalinux-tools-reference-guide.pdf (besucht am 29. 09. 2017).

[84] *UG1156: PetaLinux Tools Documentation - Workflow Tutorial*. Xilinx. Mai 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1156-petalinux-tools-workflow-tutorial.pdf (besucht am 29. 09. 2017).

[85] David A. Vallado. *Fundamentals of Astrodynamics and Applications, Third Edition*. Hawthorne Press, 2007.

[86] westamastaflash. *[1.2.2] Time Control [2.5]*. URL: https://forum.kerbalspaceprogram.com/index.php?/topic/143763-122-time-control-25/ (besucht am 03. 10. 2017).

[87] Kerbal Space Program Wiki. *Atmosphere*. URL: http://wiki.kerbalspaceprogram.com/wiki/Atmosphere (besucht am 02. 01. 2017).

[88] Kerbal Space Program Wiki. *Kerbin*. URL: http://wiki.kerbalspaceprogram.com/wiki/Kerbin0 (besucht am 02. 01. 2017).

[89] Xilinx Wiki. *OpenAMP*. 2017. URL: http://www.wiki.xilinx.com/OpenAMP (besucht am 29. 09. 2017).

[90] Mahyar R. Malekpour Wilfredo Torres-Pomales und Paul S. Miner. *ROBUS-2: A Fault-Tolerant Broadcast Communication System*. Techn. Ber. Langley Research Center, Hampton, Virginia, 2005.

[91] William J. Wolfgong. „Outgassing - Gaseous Materals Leading to Failures“. In: *Handbook of Materials Falure analysis with case Studies from Aerospace and Automotive Industries*. 2016. Kap. 15.2.

[92] Xilinx. *Xilinx Wiki: Petalinux*. 2016. URL: http://www.wiki.xilinx.com/PetaLinux (besucht am 29. 09. 2017).

[93] Fernando Martinez Vallina (Xilinx). *Processor Control of Vivado HLS Designs*. URL: https://www.xilinx.com/support/documentation/application_notes/xapp745-processor-control-vhls.pdf (besucht am 05. 10. 2017).

[94] Hang Yin. „Adaptive Embedded Systems“. Magisterarb. Mälardalen University (MDH), 2010. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-10590.

[95] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco u. a. „A survey on fault injection techniques“. In: *Int. Arab J. Inf. Technol.* 1.2 (2004), S. 171–186.