Interner Bericht

DLR-IB-FT-BS-2025-157

Versionskontrolle und Kollaboration in MBSE: Untersuchung der Git-Integration mit SysML v2

Masterarbeit

Zohair Sheikh Suleiman

Deutsches Zentrum für Luft- und Raumfahrt

Institut für Flugsystemtechnik
Braunschweig

Deutsches Zentrum
DLR für Luft- und Raumfahrt

Institutsbericht
DLR-IB-FT-BS-2025-157

Versionskontrolle und Kollaboration in MBSE: Untersuchung der Git-Integration mit SysML v2

Zohair Sheikh Suleiman

Institut für Flugsystemtechnik
Braunschweig

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Institut für Flugsystemtechnik
Abteilung Sichere Systeme & Systems Engineering

Braunschweig, den 10.10.2025

Unterschriften:

Institutsleitung:	Dr.-Ing. Andreas Bierig
Abteilungsleitung:	Dr.-Ing. Andreas Bierig
Betreuer:in:	Dr.-Ing. Oliver Bertram
Verfasser:in:	Zohair Sheikh Suleiman, B.Eng.

Masterarbeit

zur Erlangung des akademischen Grades Master of Science an der
Technischen Universität Berlin im Studiengang
Luft- und Raumfahrttechnik

Versionskontrolle und Kollaboration in MBSE: Untersuchung der Git-Integration mit SysML v2

vorgelegt von

Zohair Sheikh Suleiman, B.Eng.
Matr.-Nr. 476167

Erstprüfer:in
Prof. Dr.-Ing. Lydia Kaiser
Zweitprüfer:in
Dr.-Ing. Oliver Bertram

Betreuer:in
Dr.-Ing. Oliver Bertram

Berlin, den 17.07.25

Technische Universität Berlin
Institut für Werkzeugmaschinen und
Fabrikbetrieb (IWF)

Fachgebiet Digitales Engineering 4.0 (DE4)
Prof. Dr.-Ing. Lydia Kaiser

Pascalstraße 8-9
10587 Berlin

Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Institut für Flugsystemtechnik

Abteilung Sichere Systeme & Systems Engineering
Dr.-Ing. Oliver Bertram

Lilienthalplatz 7
38108 Braunschweig

Masterarbeit Nr. MA-0035

Versionskontrolle und Kollaboration in MBSE: Untersuchung der Git-Integration mit SysML v2

am 17.07.25

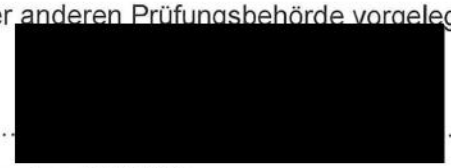
Eidesstattliche Erklärung nach § 60 Abs. 8 AllgStuPO

Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen die den benutzten Quellen und Hilfsmitteln unverändert oder sinngemäß entnommen sind, habe ich als solche kenntlich gemacht.

Sofern generative KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die jeweiligen Einsatzzwecke (z.B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 15. Februar 2023. https://www.static.tu.berlin/fileadmin/www/10002457/K3-AMBl/Amtsblatt_2023/Amtliches_Mitteilungsblatt_Nr._16_vom_30.05.2023.pdf habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.



.....

Berlin, den 17.07.25

Abstract

This thesis investigates the integration of Git with SysML v2 models to support version control and collaborative model-based systems engineering (MBSE) in safety-critical domains. Motivated by the increasing complexity of systems and the need for traceable, team-oriented development processes, the work examines how principles from software engineering – such as branching strategies, CI/CD pipelines, and text-based workflow – can be transferred to MBSE practice. The research follows a Design Science Research (DSR) approach and combines methodological analysis with practical implementation and evaluation.

The findings demonstrate that the integration of Git and SysML v2 is both technically feasible and methodologically beneficial. However, it requires clear organizational structures, role definitions, and tool configurations. The developed workflow provides a scalable foundation for collaborative modeling in interdisciplinary teams and highlights areas for future research and tool improvement.

Zusammenfassung

Diese Arbeit untersucht die Integration von Git mit SysML v2-Modellen zur Versionskontrolle und Kollaboration im modellbasierten Systems Engineering (MBSE) sicherheitskritischer Systeme. Ausgangspunkt ist die wachsende Komplexität technischer Systeme sowie der Bedarf nach nachvollziehbaren und teamorientierten Entwicklungsprozessen. Ziel war es, Prinzipien aus der Softwareentwicklung – etwa *Branching*-Strategien, CI/CD-Pipelines und textbasierte Workflows – auf das MBSE zu übertragen. Die Arbeit folgt dem Design Science Research (DSR)-Ansatz und kombiniert methodische Analyse mit praktischer Umsetzung und Evaluation.

Die Untersuchung belegt, dass die Kombination von Git und SysML v2 sowohl technisch umsetzbar als auch methodisch sinnvoll ist. Sie erfordert jedoch klare organisatorische Spielregeln. Der entwickelte Ansatz bietet eine skalierbare Grundlage für kollaboratives Modellieren in interdisziplinären Teams und zeigt Perspektiven für künftige Forschung und Werkzeugentwicklung auf.

Inhaltsverzeichnis

Verzeichnis der verwendeten Abkürzungen

AFHA	Aircraft Functional Hazard Assessment
AMBSE	Agile Model-Based Systems Engineering
AMBSE	Agile Model-Based Systems Engineering
AP	Agreement Processes
API	Application Programming Interface
ARP	SAE Aerospace Recommended Practices
ASA	Aircraft Safety Assessment
ASDP	Aircraft and System Development Process
BDD	Block-Definition-Diagramm
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
CMBSE	Collaborative Model-Based Systems Engineering
CSM	Cameo Systems Modeler
CVS	Concurrent Versions System
D&D	Data & Documentation
DAP	Development Assurance Planning
DevOps	Development and Operations
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DO	RTCA Document
DSR	Design Science Research
ED	EUROCAE Document
EUROCAE	European Organisation for Civil Aviation Equipment
FDD	Feature Driven Development

FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
FT-SSY	DLR Institut für Flugsystemtechnik, Abteilung Sichere Systeme & Systems Engineering
IBD	Internal-Block-Diagramme
IEC	International Electrotechnical Commission
INCOSE	International Council on Systems Engineering
IP	Integral Processes
IPDT	Integrated Product Development Team
ISO	International Organization for Standardization
JN	Jupyter Notebook
KerML	Kernel Modeling Language
MBSA	Model-Based Safety Analysis
MBSE	Model-Based Systems Engineering
MR	Merge Request
MTTR	Mean Time to Recovery
OEM	Original Equipment Manufacturer
OMG	Object Management Group
OOSEM	Object-Oriented Systems Engineering Method
OPEP	Organizational Project-Enabling Processes
OWL	Web Ontology Language
PASA	Preliminary Aircraft Safety Assessment
PSSA	Preliminary System Safety Assessment
RACI	Responsible, Accountable, Consulted, and Informed
RFP	Request for Proposal
RTCA	Radio Technical Commission for Aeronautics
SAE	Society of Automobile Engineers
SE	Systems Engineering

SFHA	System Functional Hazard Assessment
SoI	System of Interest
SoS	System of Systems
SSA	System Safety Assessment
SST	SysML v2 Submission Team
SysML	Systems Modeling Language
SysML v1	Systems Modeling Language Version 1
SysML v2	Systems Modeling Language Version 2
TMP	Technical Management Processes
TP	Technical Processes
UAV	Unmanned Aerial Vehicle
UI	User Interface
UML	Unified Modeling Language
V&V	Verifizierung und Validierung
XP	Extreme Programming

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Entwicklung sicherheitskritischer Flugzeugsysteme erfordert eine hohe Genauigkeit, Nachverfolgbarkeit sowie Konsistenz über den gesamten Entwicklungsprozess hinweg. *Model-Based Systems Engineering* (MBSE) adressiert diese Anforderungen durch einen modellzentrierten Ansatz, der eine konsistente und strukturierte Abbildung komplexer Systeme ermöglicht (Haberfellner et al., 2019).

Trotz dieser methodischen Vorteile bestehen weiterhin Herausforderungen in der teamübergreifenden Zusammenarbeit, insbesondere hinsichtlich der Versionskontrolle und der Rückverfolgbarkeit von Modelländerungen (Li et al., 2024; Wouters et al., 2017). Während in der Softwareentwicklung Versionskontrollsysteme wie Git etabliert sind, ist deren Anwendung im modellbasierten Systementwurf bislang nur begrenzt verbreitet.

Mit der Einführung der neuen Modellierungssprache *Systems Modeling Language* (SysML) Version 2 ergeben sich neue Potenziale zur Verbesserung der Kollaboration in MBSE-Projekten. Besonders die textuelle Notation von SysML v2 ermöglicht eine tiefere Integration mit Versionskontrollsystemen wie Git und bietet damit eine vielversprechende Grundlage für die Entwicklung kollaborativer, versionskontrollierter MBSE-Prozesse.

Die vorliegende Arbeit wurde im Rahmen einer Forschungsaktivität am Deutschen Zentrum für Luft- und Raumfahrt (DLR), Institut für Flugsystemtechnik, durchgeführt. Sie adressiert die Entwicklung und Bewertung eines Git-basierten Kollaborationsprozesses für MBSE unter Einsatz von SysML v2.

1.1 Problemstellung

Die Verwaltung von Modellversionen sowie die Nachverfolgbarkeit von Änderungen stellen zentrale Anforderungen in der Entwicklung sicherheitskritischer Systeme dar. In der Softwareentwicklung sind Versionskontrollsysteme wie Git längst etabliert und bilden dort eine essenzielle Grundlage für kollaborative, nachvollziehbare Entwicklungsprozesse. Im Kontext des MBSE hingegen fehlt bislang eine standardisierte, weit verbreitete Lösung zur strukturierten Versionsverwaltung von Systemmodellen. Die mangelnde Integration entsprechender Mechanismen in gängige MBSE-Werkzeuge erschwert die modellbasierte Zusammenarbeit und führt zu Unsicherheiten hinsichtlich der Konsistenz und Gültigkeit von Modellversionen (May & Zerwas, 2025).

Insbesondere bei der teamübergreifenden Zusammenarbeit über verteilte Entwicklungsgruppen hinweg entstehen zusätzliche Herausforderungen. Es ist häufig unklar, welche Version eines Systemmodells den aktuellen Stand der Entwicklung

darstellt (Wouters et al., 2017). In der Praxis kann es vorkommen, dass ein veralteter Modellstand bearbeitet oder weiterverwendet wird, obwohl bereits eine aktualisierte Version existiert. Dies kann erhebliche Auswirkungen auf die Qualität und Konsistenz der Systemdokumentation sowie auf die spätere Validierung haben.

Auch für externe Stakeholder besteht ein Bedarf an Transparenz und Nachvollziehbarkeit (prostep ivip Association, 2023). Beispielsweise müssen bei Audits oder Meilensteinfreigaben stets aktuelle und konsistente Modellversionen des *System of Interest* (SoI) zur Verfügung stehen. Darüber hinaus ist es entscheidend, dass der gesamte Änderungsverlauf lückenlos dokumentiert und eindeutig nachvollziehbar ist.

Erst durch die Veröffentlichung von SysML v2 mit ihrer textuellen Notation wird die tiefere Integration bestehender Versionskontrollsysteme wie Git in modellbasierte Entwicklungsprozesse überhaupt technisch möglich. Da die Systemmodelle nun als textbasierter „Quellcode“ behandelt werden können, bietet sich die Nutzung von Git als bewährtem Werkzeug für Versionsverwaltung und kollaborative Entwicklung besonders an.

1.2 Zielsetzung und Forschungsfragen

Ziel dieser Arbeit ist die Untersuchung der Integration von Git mit SysML v2-Modellen zur Versionskontrolle und Kollaboration im MBSE. Dabei sollen die Potenziale und Herausforderungen einer Git-basierten Verwaltung von modellbasierten Systementwürfen systematisch analysiert werden.

Damit sollen die folgenden vier Forschungsfragen beantwortet werden:

1. Wie lässt sich die bestehende Kollaborationsstruktur im MBSE unter Berücksichtigung relevanter Luftfahrtstandards und Teamstrukturen analysieren?
2. Welche Werkzeug-Konfigurationsrichtlinien sind erforderlich, um eine effiziente Nutzung von Git mit SysML v2 zu gewährleisten?
3. Wie kann ein Git-basierter Arbeitsablauf für SysML v2 gestaltet und implementiert werden?
4. Inwiefern ermöglicht dieser Arbeitsablauf eine verbesserte Nachverfolgbarkeit und Effizienz in der modellbasierten Entwicklung sicherheitskritischer Systeme?

Zur Beantwortung dieser Fragestellungen kommen im Rahmen der Arbeit folgende Technologien zum Einsatz:

- *Cameo Systems Modeler* (CSM) mit SysML v2-Plugin zur Modellierung eines Beispielsystems

- Git in Kombination mit GitLab zur Versionsverwaltung der Modelle
- *Jupyter Notebook* (JN) zur Visualisierung und Bearbeitung der Modelle sowie zur Automatisierung von Aufgaben mittels Python

Die Ergebnisse der Arbeit sollen eine praxisorientierte Grundlage schaffen, um die Einführung und Nutzung einer Git-basierten Versionskontrolle im MBSE zu erleichtern. Dabei steht insbesondere die Optimierung der Zusammenarbeit, Nachvollziehbarkeit und Effizienz in der modellbasierten Entwicklung sicherheitskritischer Systeme im Fokus. Die Erkenntnisse tragen somit sowohl zur methodischen Weiterentwicklung als auch zur direkten Anwendung im Kontext des DLR bei.

1.3 Forschungsmethode

Als Forschungsansatz wird die *Design Science Research* (DSR)-Methodologie nach Hevner et al., (2004) verwendet. DSR kombiniert konstruktive Forschung mit einer wissenschaftlichen Evaluierung des entwickelten Artefakts. Dabei werden drei zentrale Forschungszyklen unterschieden: Der *Relevance Cycle*, der *Rigor Cycle* und der *Design Cycle*. Diese Zyklen stehen in einer wechselseitigen Beziehung zueinander und bilden den Rahmen für die iterative Entwicklung, Validierung und Integration des Artefakts in die Praxis. Bild 1-1 stellt diese Zyklen und ihre Interaktionen grafisch dar.

Der ***Relevance Cycle*** verbindet die Forschungsaktivitäten mit der praktischen Anwendungsumgebung. Er stellt sicher, dass das entwickelte Artefakt reale Probleme adressiert und einen Mehrwert für die Praxis bietet. Die Anforderungen für die Forschung werden aus dem Anwendungskontext abgeleitet, während die Ergebnisse des Forschungsprojekts in die Praxis zurückgeführt und evaluiert werden. Iterationen dieses Zyklus sind notwendig, wenn sich im Feldtest neue Anforderungen oder Verbesserungspotenziale ergeben.

Der ***Rigor Cycle*** stellt die wissenschaftliche Fundierung sicher, indem er auf bestehende Theorien, Methoden und bewährte Praktiken zurückgreift. Die wissenschaftliche Wissensbasis liefert sowohl theoretische Grundlagen als auch bestehende Artefakte und Prozesse, die zur Entwicklung des neuen Artefakts herangezogen werden. Gleichzeitig trägt die Forschung durch neue Erkenntnisse und Erweiterungen der bestehenden Theorien zur Wissensbasis bei.

Der ***Design Cycle*** bildet das Herzstück der DSR. Er beschreibt die iterative Entwicklung und Evaluation des Artefakts. Basierend auf den Anforderungen aus dem *Relevance Cycle* und den wissenschaftlichen Grundlagen aus dem *Rigor Cycle* werden Designalternativen entwickelt, getestet und optimiert. Dieser Prozess wiederholt sich, bis das Artefakt eine zufriedenstellende Lösung für das identifizierte Problem darstellt. Eine umfassende Evaluierung des Artefakts ist essenziell, um die Qualität und Anwendbarkeit der Lösung sicherzustellen.

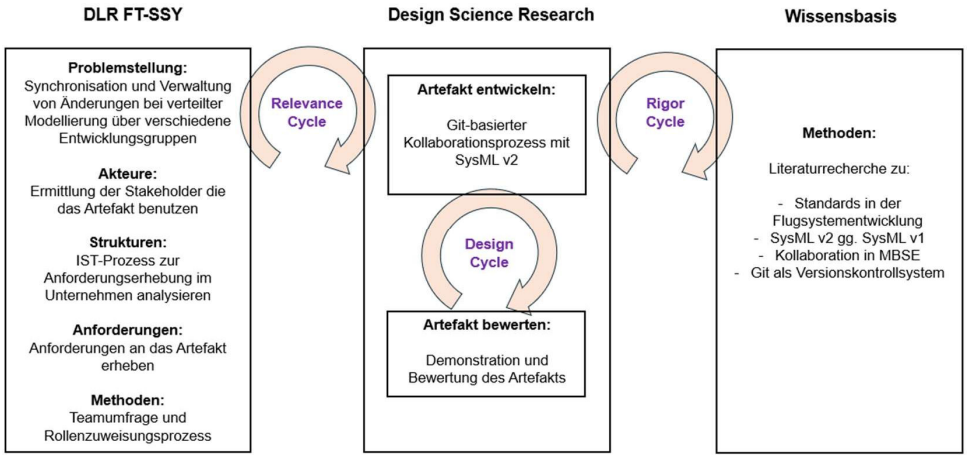


Bild 1-1 Design Science Research-Methodologie (eigene Darstellung nach Hevner et al., (2004))

1.4 Struktur der Arbeit

Kapitel 1 stellt die grundlegende Motivation der Arbeit dar, indem die Problemstellung erläutert und die Forschungsfragen formuliert werden. Zudem wird die gewählte Forschungsmethodik vorgestellt, um den wissenschaftlichen Rahmen der Untersuchung abzustecken.

Kapitel 2 ist dem *Rigor Cycle* zugeordnet und analysiert den bestehenden Wissensstand, um eine fundierte theoretische Grundlage zu schaffen. Hierzu werden relevante Normen und Standards für die Flugsystementwicklung betrachtet, darunter ARP4754B, ARP4761A und ISO/IEC 15288. Darüber hinaus werden MBSE, die SysML, sowie die Kollaborationsherausforderungen in MBSE untersucht, um eine fundierte Basis für die weiteren Analysen zu schaffen.

Kapitel 3 widmet sich der Untersuchung der Forschungsumgebung und ordnet diese in den *Relevance Cycle* der DSR-Methodologie ein. Basierend auf einer Teamumfrage und einem Rollenzuweisungsprozess wird analysiert, welche Kompetenzen innerhalb des Teams bestehen und wie die Entwicklungsprozesse im Kontext der Flugsystementwicklung organisiert sind. Ein besonderer Fokus liegt dabei auf der Identifikation der Herausforderungen in der Zusammenarbeit, die als Kriterien für die spätere Entwicklung eines Git-basierten Kollaborationsprozesses dienen. Zudem wird die Arbeitsweise innerhalb der Forschungsgruppe des DLR beschrieben und evaluiert.

Aufbauend auf den Erkenntnissen der vorherigen Kapitel wird in **Kapitel 4** ein Artefakt entwickelt: ein Git-basierter Kollaborationsprozess für die Flugsystementwicklung mit SysML v2. Dieser Prozess wird so gestaltet, dass er den identifizierten Anforderungen gerecht wird. Ergänzend werden Konfigurationsrichtlinien für

CSM und Git definiert. Die Funktionsweise wird anhand eines UAV-Beispielsystems demonstriert, das in SysML v2 modelliert und mit Git integriert wird.

Kapitel 5 behandelt die Validierung des entwickelten Prozesses als Teil des *Design Cycle* der DSR-Methodologie anhand definierter Testszenarien. Diese Tests zielen darauf ab, sowohl technische als auch methodische Aspekte des Kollaborationsprozesses zu überprüfen. Nach Durchführung der Tests werden die Ergebnisse ausgewertet, um eine abschließende Bewertung des Artefakts vorzunehmen.

Kapitel 6 fasst die zentralen Erkenntnisse der Arbeit zusammen und reflektiert die gewonnenen Ergebnisse. Zudem werden mögliche Limitationen der Untersuchung aufgezeigt und Perspektiven für zukünftige Forschung sowie Weiterentwicklungen des Artefakts diskutiert.

2 Theoretische Grundlagen und Stand der Technik

Dieses Kapitel ist dem *Rigor Cycle* der DSR-Methodik zugeordnet. Ziel ist es, die theoretische und methodische Fundierung der Arbeit sicherzustellen, indem bestehende Standards, Methoden und Werkzeuge aus dem Bereich MBSE sowie der Luftfahrttechnik systematisch aufgearbeitet und analysiert werden. Die hier erarbeiteten Grundlagen bilden eine fundierte Wissensbasis für die Artefakt-Entwicklung im *Design Cycle* und tragen zugleich zum vertieften Verständnis der Forschungsumgebung im *Relevance Cycle* bei.

Zunächst werden in **Kapitel 2.1** zentrale Prozessstandards der Flugsystementwicklung vorgestellt, darunter ARP4754B, ARP4761A und ISO/IEC 15288. Diese Normen definieren wesentliche Anforderungen an sicherheitskritische Entwicklungsprozesse und dienen als Referenzrahmen für die spätere Gestaltung des Git-basierten Kollaborationsprozesses.

Kapitel 2.2 führt in das Konzept des MBSE ein, das als methodisches Fundament der Arbeit fungiert. Darauf aufbauend werden in **Kapitel 2.3** die beiden Versionen der SysML miteinander verglichen, wobei der Fokus auf der Weiterentwicklung von SysML v1 zu v2 liegt.

In **Kapitel 2.4** wird CSM als in dieser Arbeit eingesetztes Modellierungswerkzeug vorgestellt, inklusive seiner Unterstützung für SysML v2.

Anschließend widmet sich **Kapitel 2.5** dem Thema Kollaboration in MBSE. Hier werden bestehende Herausforderungen analysiert und Ansätze wie Agile MBSE (AMBSE) beleuchtet. Abschließend wird das Versionskontrollsystem Git betrachtet, das als zentrales Werkzeug für den in dieser Arbeit entwickelten Kollaborationsprozess dient.

2.1 Standards in der Flugsystementwicklung

Die Entwicklung moderner Flugsysteme unterliegt besonders hohen Anforderungen an Struktur, Nachvollziehbarkeit und Systemsicherheit. Um diesen gerecht zu werden, greift die zivile Luftfahrtindustrie auf bewährte Standards und empfohlene Vorgehensweisen zurück, die fest in den Entwicklungsprozessen verankert sind. Im Rahmen dieser Arbeit werden insbesondere die SAE *Aerospace Recommended Practices* ARP4754B und ARP4761A sowie die internationale Norm ISO/IEC 15288 betrachtet.

Diese Regelwerke strukturieren die Systementwicklung und begleiten den gesamten Lebenszyklus technischer Systeme – von der Anforderungsdefinition über Entwicklung, Integration und Verifikation bis hin zu Betrieb, Wartung und

Außerdienststellung. Ihre Anwendung ist zentral, um die funktionale Eignung, Sicherheit und Zulassungsfähigkeit eines Luftfahrtsystems nachweisbar sicherzustellen.

Da sich diese Arbeit auf die modellbasierte Systementwicklung konzentriert, dienen die genannten Standards als verbindlicher Rahmen für die Gestaltung konformer Entwicklungsprozesse. Sie legen Anforderungen an Rückverfolgbarkeit, Dokumentation und systematische Analyse fest, die auch bei der Gestaltung kollaborativer MBSE-Ansätze mit SysML v2 und Git berücksichtigt werden müssen.

2.1.1 ARP4754B und ARP4761A Prozesse

Die SAE ARP4754B und ARP4761A bilden gemeinsam ein abgestimmtes Rahmenwerk zur sicheren Entwicklung von Luftfahrtsystemen. Während ARP4754B den allgemeinen Entwicklungsprozess für Luftfahrzeuge und deren Systeme beschreibt, ergänzt ARP4761A diesen durch spezifische Methoden zur Sicherheitsbewertung (SAE Aerospace Recommended Practice, 2023a).

ARP4754B stellt Empfehlungen für die Entwicklung von Luftfahrzeugen und Systemen bereit, wobei insbesondere die Luftfahrzeugfunktionen und deren Betriebsumgebung berücksichtigt werden. Es enthält Vorgehensweisen zur Sicherstellung der Gesamtsicherheit des Entwurfs, zur Einhaltung behördlicher Vorgaben sowie zur Unterstützung firmeninterner Standards. Dabei umfasst es sowohl die Validierung der Anforderungen als auch die Verifikation der Umsetzung, insbesondere in Bezug auf Sicherheit, Zertifizierbarkeit und Produktqualität.

Die Zielsetzung der ARP4754B besteht in der Bereitstellung bewährter industrieller Praktiken zur strukturierten Entwicklung integrierter, oft von verschiedenen Organisationen entwickelter Systeme. Diese Systeme müssen diszipliniert und systematisch entwickelt werden, um sicherzustellen, dass sicherheitsrelevante und funktionale Anforderungen erfüllt und nachgewiesen werden können. Die Empfehlungen sind nicht als regulatorische Anforderungen zu verstehen, sondern als industrieübliche Vorgehensweisen. Abweichende Methoden können zulässig sein, sofern sie eine gleichwertige Nachweisführung ermöglichen.

Die ARP4761A bietet ergänzend dazu Richtlinien zur Durchführung von Sicherheitsanalysen für zivile Luftfahrzeuge, Systeme und Ausrüstungen. Diese können zur Einhaltung behördlicher Zertifizierungsanforderungen oder firmeninterner Sicherheitsstandards herangezogen werden. Auch wenn der Fokus auf Neuentwicklungen liegt, sind die Methoden ebenfalls für bestehende Systeme bei Änderungen oder Derivatanwendungen anwendbar. Sicherheitsbewertungen von in Betrieb befindlichen Produkten sind hingegen nicht Gegenstand dieses Dokuments (SAE Aerospace Recommended Practice, 2023b).

Die Zielsetzung der ARP4761A besteht darin, ein akzeptiertes Verfahren zur Sicherheitsbewertung bereitzustellen. Es beinhaltet unter anderem die folgenden methodischen Schritte:

- *Aircraft Functional Hazard Assessment* (AFHA)
- *Preliminary Aircraft Safety Assessment* (PASA)
- *System Functional Hazard Assessment* (SFHA)
- *Preliminary System Safety Assessment* (PSSA)
- *System Safety Assessment* (SSA)
- *Aircraft Safety Assessment* (ASA)

AFHA, PASA und ASA werden auf der Ebene des Flugzeugs durchgeführt, während SFHA, PSSA und SSA auf der Systemebene durchgeführt werden. Zur Durchführung dieser Bewertungen werden unterschiedliche Analysemethoden empfohlen, darunter *Fault Tree Analysis* (FTA), *Failure Modes and Effects Analysis* (FMEA), *Model-Based Safety Analysis* (MBSA) und andere.

ARP4754B ist als Hauptstandard zu verstehen, während ARP4761A eine komplementäre Funktion übernimmt. Die Beziehung zwischen ARP4754B und seinen unterstützenden Standards ist in Bild 2-1 dargestellt.

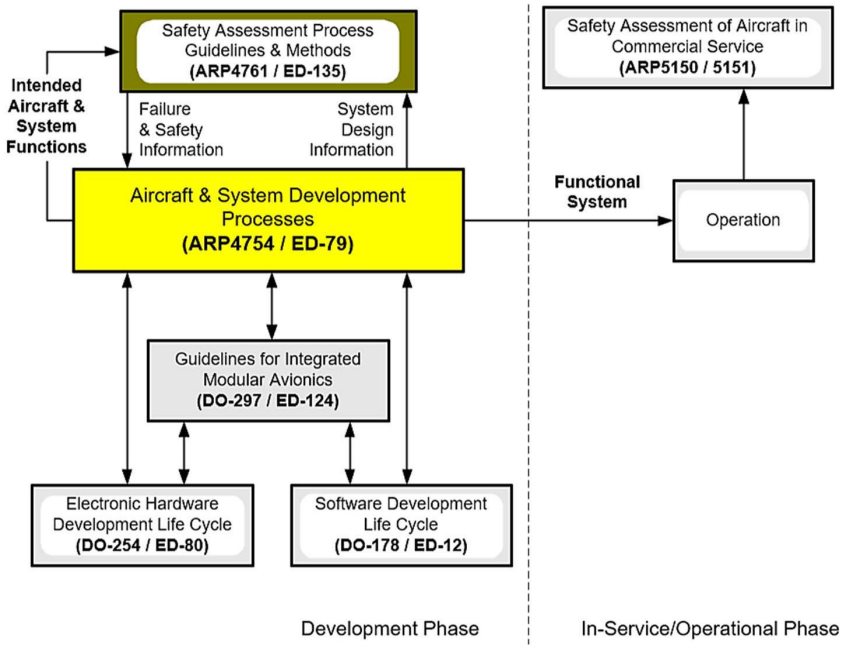


Bild 2-1 Grundsatzdokumente für die Entwicklungs- und die Betriebsphase (SAE Aerospace Recommended Practice, 2023a)

Ziel der Systementwicklungsphase ist die Bereitstellung eines funktionalen Systems, das in den Betrieb überführt werden kann. Um dies zu erreichen, müssen innerhalb dieser Phase mindestens fünf verschiedene Standards bzw. Richtlinien berücksichtigt werden. In dieser Arbeit werden ausschließlich die Standards ARP4754B und ARP4761A behandelt.

Der Flugzeug-/Systementwicklungsprozess umfasst insgesamt 16 Teilprozesse, welche in vier Hauptkategorien gegliedert sind:

- *Development Assurance Planning* (DAP) – 3 Prozesse
- *Aircraft and System Development Process* (ASDP) – 5 Prozesse
- *Data & Documentation* (D&D) – 1 Prozess
- *Integral Processes* (IP) – 7 Prozesse

Tabelle 2-1 zeigt die zentralen Prozesse gemäß ARP4754B, die im Rahmen dieser Arbeit betrachtet werden. Die Spalte „ARP-ID #“ dient der schnellen Orientierung innerhalb des Standards.

Tabelle 2-1 Gesamtüberblick ARP4754B-Prozesse (SAE Aerospace Recommended Practice, 2023a)

Cat.	ARP ID #	Prozess (Engl.)
DAP	3.1	Development Assurance Planning Process
DAP	3.2	Development Assurance Plan
DAP	3.3	Certification Authority Coordination
ASDP	4.2	Aircraft Function and Requirement Development
ASDP	4.3	Development of Aircraft Architecture and Allocation of Aircraft Functions to Systems
ASDP	4.4	Development of System Functions and Requirements
ASDP	4.5	Development of System Architecture and Allocation of System Requirements to Items
ASDP	4.6	Implementation
D&D	4.7	Summary of Development Assurance Process Outputs
IP	5.1	Safety Assessment (ARP4761A)
IP	5.2	Development Assurance Level Assignment
IP	5.3	Requirements Capture
IP	5.4	Requirements Validation
IP	5.5	Implementation Verification
IP	5.6	Configuration Management
IP	5.7	Process Assurance

Die **DAP** definieren den Entwicklungsrahmen für alle nachgelagerten Prozesse – ASDP, D&D und IP. Mit der Genehmigung des *Development Assurance Plans* beginnt die eigentliche Entwicklungsphase.

Die **ASDP** stellen das Kernstück der Systementwicklung dar. Sie enthalten Richtlinien zur Entwicklung ziviler Luftfahrzeuge und Systeme und orientieren sich am klassischen V-Modell.

Der **D&D-**Prozess sorgt dafür, dass alle Ergebnisse gemäß den Vorgaben des DAP dokumentiert werden. Abweichungen vom Plan müssen begründet und dokumentiert werden.

Die **IP** werden iterativ innerhalb jedes ASDP-Schritts durchlaufen. Sie spiegeln die Realität moderner Entwicklungsprozesse wider, in denen Tätigkeiten oft parallel und zyklisch erfolgen. Der Einstiegspunkt in das Entwicklungsmodell kann variieren, je nachdem, ob ein neues Funktionskonzept eingeführt oder eine bestehende Funktion modifiziert wird. In jedem Fall ist eine Bewertung der Auswirkungen auf andere Funktionen und Anforderungen erforderlich.

Bild 2-2 zeigt den vollständigen Flugzeug-/Systementwicklungsprozess nach ARP4754B inklusive der Interaktionen zwischen den vier Prozesskategorien.

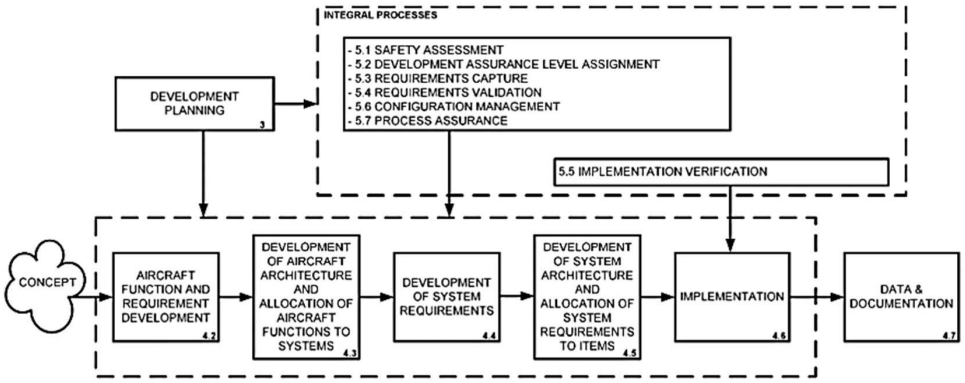


Bild 2-2 Luftfahrzeug-/Systementwicklungsprozess nach ARP4754B (SAE Aerospace Recommended Practice, 2023a)

Für Systemingenieure sind insbesondere die ASDP-Prozesse relevant, da in diesen der eigentliche Systementwicklungsprozess abgebildet wird. Das V-Modell in Bild 2-3 bildet den strukturellen Rahmen für die Entwicklung und Validierung sicherheitskritischer Systeme in der Luftfahrt. Der linke Ast des V-Modells repräsentiert die Phase, in der die Anforderungen an das Luftfahrzeug bzw. Systeme sowie Funktionen und Systemarchitekturen entwickelt und auf Subsysteme zugewiesen werden.

Im unteren Bereich des V-Modells erfolgt die detaillierte Entwicklung und Verifikation von Hardware und Software. Diese Aktivitäten werden gemäß den in der

Luftfahrt etablierten Standards durchgeführt – insbesondere DO-178C / ED-12C für Software sowie DO-254 / ED-80 für Hardware.

Der rechte Ast des V-Modells beschreibt die stufenweise Integration von Systemen und Subsystemen sowie die zugehörigen Verifikations- und Validierungsaktivitäten (V&V). Ziel dieser Phase ist es, nachzuweisen, dass die entwickelten Systeme die gestellten Anforderungen erfüllen und für den vorgesehenen Betrieb geeignet sind.

Die Interaktion zwischen den Prozessen der ARP4754B und ARP4761A erfolgt insbesondere im Rahmen der Sicherheitsanalyse. Diese ist im IP der ARP4754B unter der ARP-ID #5.1 „*Safety Assessment*“ verankert. In Bild 2-3 sind die verschiedenen Arten von Sicherheitsanalysen dargestellt, die in den jeweiligen Phasen des Entwicklungsprozesses durchzuführen sind. Dadurch wird verdeutlicht, wie sicherheitsrelevante Erkenntnisse systematisch in die funktionale und technische Entwicklung einfließen.

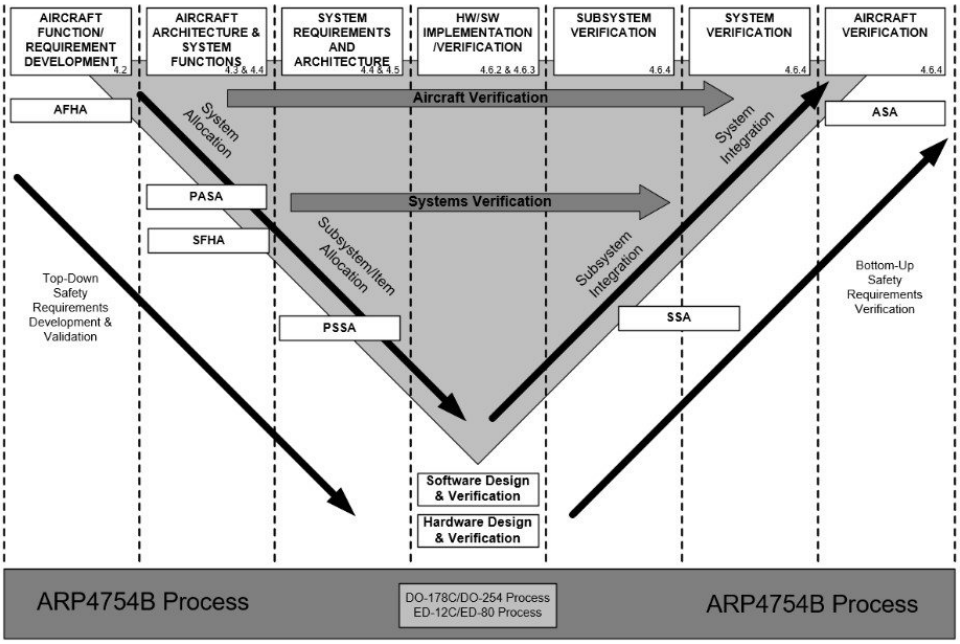


Bild 2-3 Interaktion zwischen ARP4754B und ARP4761A Prozesse (SAE Aerospace Recommended Practice, 2023a)

2.1.2 ISO/IEC 15288 Prozesse

Die Norm ISO/IEC 15288:2023 definiert einen einheitlichen Rahmen zur Beschreibung des Lebenszyklus technischer Systeme. Ziel ist es, durch standardisierte Prozesse und Begriffe die Kommunikation zwischen allen Beteiligten eines Projekts – einschließlich Erwerber, Lieferant und weiterer Stakeholder – zu erleichtern (ISO/IEC & IEEE, 2023). Diese Prozesse können auf jeder Ebene eines Systems

angewendet werden und decken sämtliche Lebenszyklusphasen ab: von der Konzeption über die Entwicklung und Nutzung bis hin zur Außerdienststellung. Darüber hinaus ermöglicht die Norm Organisationen die gezielte Definition, Steuerung und Verbesserung ihrer Lebenszyklusprozesse. Die Anwendung erfolgt sowohl in interner als auch externer Projektdurchführung und kann auf Einzelorganisationen oder kooperierende Parteien ausgeweitet werden.

ISO/IEC 15288 adressiert eine Vielzahl an Systemen unterschiedlichster Komplexität, Größe, Zweckbestimmung oder Lebensdauer. Systeme können dabei aus beliebigen Kombinationen von Hardware, Software, Daten, Personen, Verfahren, Anleitungen oder Einrichtungen bestehen. Für softwarebasierte Systemelemente verweist ISO/IEC 15288 auf die ergänzende Norm ISO/IEC 12207, mit der sie harmonisiert wurde (ISO/IEC & IEEE, 2023).

Kapitel 6 der Norm beschreibt ein Prozessreferenzmodell, das sich aus insgesamt 30 Prozessen zusammensetzt. Diese sind in vier Hauptkategorien gegliedert:

- *Agreement Processes* (AP) – 2 Prozesse
- *Organizational Project-Enabling Processes* (OPEP) – 6 Prozesse
- *Technical Management Processes* (TMP) – 8 Prozesse
- *Technical Processes* (TP) – 14 Prozesse

Eine vollständige Übersicht dieser Prozesse findet sich in Anhang A1.

Jeder Prozess wird in mehreren standardisierten Abschnitten beschrieben. Hierzu gehören: Zweck (*Purpose*), Beschreibung (*Description*), Eingaben/Ausgaben (*Inputs/Outputs*), Prozessaktivitäten (*Process Activities*) sowie Erläuterung (*Elaboration*). Diese Struktur erleichtert sowohl die praktische Anwendung in Projekten als auch die Bewertung im Sinne der Prozessreife, wie sie beispielsweise in ISO/IEC 15504 vorgesehen ist (Walden & International Council on Systems Engineering, 2023).

Die **AP** umfassen den Erwerbs- und den Lieferprozess. Sie beschreiben die Aktivitäten, die erforderlich sind, um Vereinbarungen zwischen internen und externen Organisationseinheiten zu etablieren, etwa bei der Beschaffung oder Lieferung von Produkten und Dienstleistungen.

Die **OPEP** stellen sicher, dass eine Organisation über die notwendigen Fähigkeiten, Ressourcen und die Infrastruktur verfügt, um Projekte wirksam zu initiieren, zu unterstützen und zu kontrollieren. Diese Prozesse dienen nicht der umfassenden strategischen Unternehmensführung, sondern unterstützen gezielt projektbezogene Aktivitäten.

Die **TMP** befassen sich mit der Planung, Durchführung, Bewertung und Steuerung von Projekten. Sie können je nach Bedarf in unterschiedlichen Phasen und Ebenen

eines Projekthierarchiebaums angewendet werden und werden abhängig von Risiko und Komplexität unterschiedlich formalisiert.

Die **TP** definieren jene Aktivitäten, die notwendig sind, um Systemanforderungen zu definieren, Produkte entsprechend umzusetzen und deren Nutzung, Wartung sowie Ausmusterung zu gewährleisten. Diese Prozesse fördern technische Entscheidungen, die die Produktqualität sowie die Einhaltung gesellschaftlicher Anforderungen (z. B. Sicherheit, Umweltverträglichkeit) sicherstellen.

Im Vergleich zu den Prozessen der ARP4754B zeigt sich, dass die ISO/IEC 15288-Prozesse deutlich umfassender sind. Während sich ARP4754B primär auf die Entwicklungsphase konzentriert, deckt ISO/IEC 15288 den vollständigen Systemlebenszyklus ab. Dies resultiert in einem erhöhten Planungsaufwand, zusätzlicher Ressourcennutzung und einem erweiterten Kompetenzbedarf über Fachdisziplinen hinweg.

Ein genauerer Vergleich der technische Prozessen (TP) verdeutlicht, dass drei spezifische Prozesse – der *Operation Process*, der *Maintenance Process* sowie der *Disposal Process* – in ARP4754B nicht abgebildet sind. Dies liegt daran, dass ARP4754B die Betriebsphase nicht adressiert (siehe Bild 2-1). Die in ARP4761A definierten Sicherheitsprozesse sind hingegen nicht Bestandteil der ISO/IEC 15288, da sich letztere auf allgemeine Lebenszyklusprozesse konzentriert und keine spezialisierte Sicherheitsanalyse bereitstellt.

2.2 Model-Based Systems Engineering

Das *Model-Based Systems Engineering* (MBSE) wurde 2007 im *„Systems Engineering Vision 2020“* des *International Council on Systems Engineering* (INCOSE) als ein formalisierter Ansatz beschrieben, der die Modellierung zur Unterstützung der Anforderungen, der Systemarchitektur, der Analyse sowie der Verifikation und Validierung über den gesamten Lebenszyklus eines Systems hinweg einsetzt (International Council on Systems Engineering, 2007). Ziel ist es, das bisher dokumentenzentrierte Vorgehen durch ein modellzentriertes Paradigma zu ersetzen, das eine tiefere Integration in bestehende Systementwicklungsprozesse erlaubt. MBSE ist Teil einer disziplinübergreifenden Entwicklung hin zu modellbasierten Vorgehensweisen, wie sie auch in der Mechanik, Elektronik und Softwareentwicklung zu beobachten ist.

Durch den Einsatz modellbasierter Techniken wird erwartet, dass MBSE wesentliche Vorteile gegenüber der traditionellen Dokumentation bietet – darunter eine höhere Produktivität, verbesserte Qualität, reduzierte Entwicklungsrisiken sowie eine effektivere Kommunikation im Entwicklerteam (Haberfellner et al., 2019).

Während Modelle seit jeher ein zentrales Hilfsmittel in der Systementwicklung darstellen, zeichnet sich MBSE dadurch aus, dass das Systemmodell zur verbindlichen

und durchgängigen Repräsentation (*„single source of truth“*) wird. Es integriert Informationen zu Anforderungen, Funktionen, Struktur, Verhalten sowie zur Verifikation und Validierung in einer konsistenten, maschineninterpretierbaren Form (Hick et al., 2019). Das Systemmodell dient somit nicht nur als technische Dokumentation, sondern als aktives Steuerungsinstrument des Entwicklungsprozesses.

Zentrale Prinzipien des MBSE umfassen die disziplinübergreifende Modellintegration, die Wiederverwendbarkeit von Modellelementen, die durchgängige Rückverfolgbarkeit sowie die Nutzung formalisierter Sichten zur Reduktion von Komplexität. Letztere ergibt sich im Systementwurf nicht nur aus der Anzahl von Komponenten, sondern auch aus deren Interaktionen und der Dynamik über Systemgrenzen hinweg. Madni et al., (2023) zeigen, dass durch Praktiken wie Abstraktion, Trennung von Belangen und strukturbasierte Dekomposition verschiedene Komplexitätsarten – z. B. funktionale, strukturelle oder emergente Komplexität – gezielt beherrscht werden können.

Ein wesentliches Artefakt in diesem Kontext ist das Systemmodell selbst. Es fungiert als integratives Bindeglied zwischen unterschiedlichen Disziplinen und Entwicklungsphasen. Gemäß Gräßler et al., (2022) sowie Hick et al., (2019) verbindet das Modell Anforderungen, logische und physische Systemelemente sowie Verifikations- und Validierungsaktivitäten. Damit wird es zu einem zentralen Kommunikationsmittel für alle Beteiligten – sowohl innerhalb des Entwicklerteams als auch gegenüber externen Stakeholdern.

Die Kopplung disziplinübergreifender Modelle unterstützt ein konsistentes und ganzheitliches Systemverständnis, ohne die erforderliche Detailtiefe einzelner Fachdomänen zu vernachlässigen. Gleichzeitig können durch die Reduktion auf aufgabenspezifische Sichten sowie eine zielgerichtete Navigation durch die Modellhierarchie die Zugänglichkeit und Akzeptanz modellbasierter Methoden erhöht werden.

Ein zentrales Element des MBSE ist das sogenannte MBSE-Dreieck (siehe Bild 2-4), welches das Zusammenspiel von Modellierungssprache, Modellierungsmethode und Werkzeug verdeutlicht. Zur Erstellung eines konsistenten und praxisrelevanten Systemmodells reicht eine grafische Sprache allein nicht aus. Vielmehr bedarf es einer abgestimmten Kombination aus Sprache, Methode und unterstützendem Softwarewerkzeug, um die Vorteile modellbasierter Systementwicklung in der industriellen Praxis nutzbar zu machen (Kaiser, 2013).

Die **Modellierungssprache** stellt dabei lediglich das Ausdrucksmittel dar. Sie definiert die formale Struktur, mit der Systeme beschrieben und analysiert werden können. Erst durch die **Modellierungsmethode** wird festgelegt, wie und zu welchem Zweck die Sprache angewendet wird. Die Methode definiert, welche Systemaspekte modelliert und in welcher Reihenfolge die Modellinhalte erzeugt werden

sollen. Sie dient damit als Bindeglied zwischen Sprache und Werkzeug und stellt sicher, dass das Modell den Anforderungen der jeweiligen Domäne entspricht.

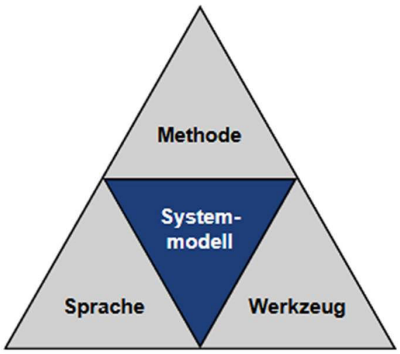


Bild 2-4 MBSE-Dreieck (Kaiser, 2013)

Das verwendete **Werkzeug** wiederum bildet die technische Grundlage zur Anwendung der Sprache und Methode. Es ermöglicht die Erstellung, Pflege und Analyse komplexer Systemmodelle und stellt Funktionen wie Versionskontrolle, Modellvalidierung und Kollaborationsunterstützung bereit (Kaiser, 2013).

Im Rahmen dieser Arbeit wird SysML v2 als Modellierungssprache verwendet, während der CSM als zentrales Modellierungswerkzeug zum Einsatz kommt. Einer vordefinierten Modellierungsmethode wird dabei nicht gefolgt, da das Ziel dieser Arbeit die Entwicklung eines Git-basierten Kollaborationsprozesses ist. Bei der Modellierung des UAV-Beispielsystems werden daher werkzeugspezifische sowie umgebungsspezifische Anforderungen berücksichtigt (siehe Kapitel 3.5). Ziel ist es, auf Basis dieser Komponenten einen strukturierten Kollaborationsansatz zu schaffen, der die Zusammenarbeit innerhalb von SE-Teams unterstützt.

2.3 Systems Modeling Language

Die *Systems Modeling Language* (SysML) ist eine von der *Object Management Group* (OMG) entwickelte, domänenunabhängige Modellierungssprache für die Spezifikation, Analyse, das Design und die Verifikation komplexer technischer Systeme. Die Sprache basiert auf einer reduzierten und erweiterten Version der *Unified Modeling Language* (UML) und wurde spezifisch entwickelt, um die Anforderungen des *Systems Engineerings* zu adressieren. Die Modellierung mit SysML ermöglicht die integrierte Darstellung von Anforderungen, Verhalten, Struktur und Parametern eines Systems und schafft so eine gemeinsame Grundlage für den interdisziplinären Austausch zwischen verschiedenen Ingenieurdisziplinen (Friedenthal et al., 2009).

Die initiale Version SysML v1.0 wurde im September 2007 veröffentlicht. Die aktuell veröffentliche stabile Version ist SysML v1.7 (Stand: Juni 2024). Parallel dazu

verlagert sich der Fokus der OMG zunehmend auf die Entwicklung der nächsten Sprachgeneration, SysML v2. Eine erste Demoversion von SysML v2 wurde im April 2024 veröffentlicht (Friedenthal, 2024). Der letzte veröffentlichte Entwurf stammt vom April 2025 und stellt den bislang aktuellsten Stand der Sprache dar (OMG Systems Modeling Community, n.d.). Obwohl SysML v2 bereits öffentlich zugänglich ist, befindet sich die Sprache weiterhin in der Finalisierungsphase. Die grundlegenden Unterschiede zwischen den Versionen sowie der aktuelle Entwicklungsstand von SysML v2 werden im Folgenden näher erläutert.

2.3.1 SysML v1

Die erste Version der *Systems Modeling Language* (SysML v1) wurde als Erweiterung der UML für die Systementwicklung konzipiert und im Jahr 2007 von der OMG standardisiert. Sie etablierte sich schnell als weit verbreiteter Standard zur modellbasierten Beschreibung technischer Systeme. SysML v1 basiert konzeptuell auf vier Modellkategorien – den sogenannten Vier Säulen von SysML: **Struktur**, **Verhalten**, **Anforderungen** und **Parametrik** (Friedenthal et al., 2009).

Bild 2-5 veranschaulicht die vier zentralen Modellkategorien und ordnet ihnen spezifische Diagrammtypen zu.

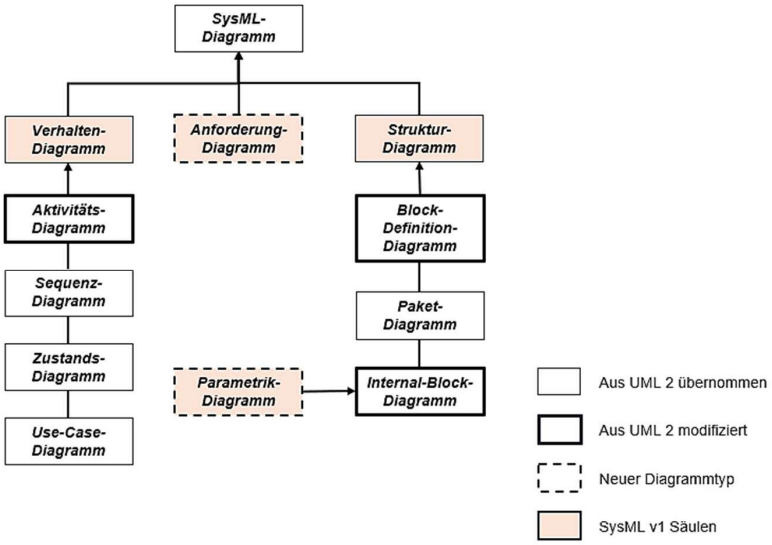


Bild 2-5 Die vier Säulen der SysML v1 mit Diagrammtypen (eigene Darstellung nach Friedenthal et al., (2009))

Die erste Säule, **Struktur**, bildet die statische Organisation und Zusammensetzung des Systems ab. Das zentrale Element dieser Kategorie ist der „Block“, der als Basiseinheit sowohl physische als auch logische Systemelemente wie Hardware, Software, Personen oder Einrichtungen darstellen kann. Die Struktur eines Systems wird dabei mithilfe von Blockdefinitionsdiagrammen (BDD) modelliert, die die

hierarchische Gliederung und Klassifizierung von Systemkomponenten beschreiben. Die interne Struktur eines Blocks wird durch Interne Blockdiagramme (IBD) dargestellt, welche die Zusammensetzung aus Teilen, Schnittstellen (*Ports*) und Verbindungen zeigen. Zur strukturellen Organisation auf höherer Ebene dient zudem das Paketdiagramm, mit dem Modellelemente logisch gruppiert werden können.

Die zweite Säule, **Verhalten**, beschreibt die dynamischen Aspekte des Systems – also, wie sich Systemelemente verhalten, interagieren und auf Ereignisse reagieren. Hierfür stehen mehrere Diagrammtypen zur Verfügung: *Use-Case*-Diagramme geben eine abstrakte Übersicht über die Funktionalitäten des Systems und deren Interaktionen mit externen Akteuren. Aktivitätsdiagramme zeigen den Ablauf von Aktionen und den Fluss von Daten und Kontrolle. Sequenzdiagramme stellen die zeitliche Abfolge von Nachrichten zwischen interagierenden Systemteilen dar, während Zustandsdiagramme das zustandsbasierte Verhalten eines Systems oder Systemteils modellieren (Friedenthal et al., 2009).

Die dritte Säule, **Anforderungen**, ermöglicht die Modellierung textbasierter Anforderungen sowie deren Verknüpfung mit anderen Modellelementen. Das Anforderungsdiagramm erlaubt die Darstellung von Anforderungshierarchien und -ableitungen und verknüpft Anforderungen über die Beziehungen *„satisfy“* und *„verify“* mit strukturellen oder verhaltensbezogenen Elementen. Dadurch wird eine konsistente Nachverfolgbarkeit zwischen Anforderungen und Systementwurf sichergestellt und eine Brücke zum klassischen Anforderungsmanagement geschaffen.

Die vierte Säule, **Parametrik**, erweitert die Modellierung um mathematische Randbedingungen und Beziehungen zwischen physikalischen Eigenschaften. Parametrikdiagramme verwenden sogenannte *„constraint blocks“*, um beispielsweise Leistung-, Masse- oder Zuverlässigkeitsanforderungen als Gleichungssysteme im Modell zu verankern. Diese Diagramme ermöglichen die Integration von Analysemodellen in die Modellierungsumgebung und fördern so die Verbindung von Systementwurf und Simulation.

Ergänzend zu den genannten Diagrammen stellt SysML v1 das *Allocation*-Konstrukt bereit, um verschiedene Zuweisungsbeziehungen abzubilden – etwa zwischen Funktionen und Komponenten, logischen und physischen Elementen oder Software und Hardware (Friedenthal et al., 2009).

2.3.2 SysML v2

Die *Systems Modeling Language* Version 2 (SysML v2) wurde durch das SysML v2 *Submission Team* (SST) als Antwort auf das im Dezember 2017 durch die OMG veröffentlichte *Request for Proposal* (RFP) entwickelt. Die daraus resultierenden Spezifikationen – bestehend aus der *Kernel Modeling Language* (KerML), der

grafischen und textuellen Notation von SysML v2 sowie der *Systems Modeling API & Services* – befinden sich derzeit in der Finalisierungsphase (Stand: April 2025).

Ziel von SysML v2 ist es, die Akzeptanz und Effektivität modellbasierter Systementwicklung durch Verbesserungen in mehreren Bereichen deutlich zu steigern (Friedenthal, 2024):

- Präzision und Ausdrucksstärke der Sprache,
- Konsistenz und Integration zwischen Sprachkonzepten,
- Interoperabilität mit anderen Ingenieurmodellen und Werkzeugen,
- Benutzerfreundlichkeit für Modellentwickler:innen und -nutzer:innen,
- Erweiterbarkeit zur Unterstützung domänenspezifischer Anwendungen,
- sowie die Bereitstellung eines Migrationspfades für Anwender:innen und Tool-Hersteller von SysML v1.

Trotz des noch laufenden Finalisierungsprozesses sind bereits öffentlich zugängliche Demonstrationsversionen verfügbar. Diese können über die OMG offizielle GitHub-Seite (OMG Systems Modeling Community, n.d.) heruntergeladen und in Umgebungen wie *Jupyter Notebook* oder *Eclipse* ausgeführt werden. Auch weiterführende Informationen und aktuelle Entwicklungen werden dort veröffentlicht.

Im Gegensatz zu SysML v1, das auf sogenannten Säulen (engl. *pillars*) beruht, verfolgt SysML v2 einen modulareren Ansatz. Die Sprache wurde entlang definierter Fähigkeiten (engl. *capabilities*) strukturiert, die sich auf bestimmte Modellierungsaspekte konzentrieren. Diese Umstellung unterstützt die Erweiterbarkeit und modulare Weiterentwicklung der Sprache. Bild 2-6 zeigt einen Überblick über die zentralen Sprachfähigkeiten von SysML v2.

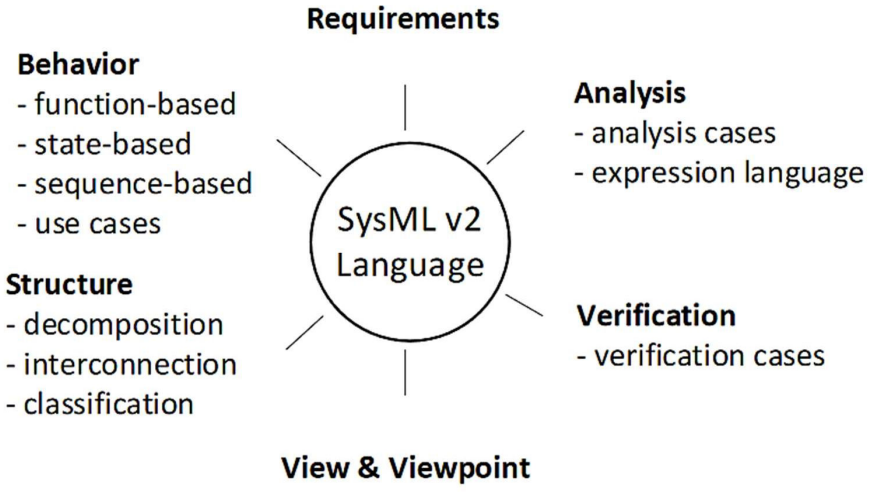


Bild 2-6 SysML v2 Sprachfähigkeiten (Friedenthal, 2024)

Drei der zentralen neuen Sprachfähigkeiten sind:

1. ***View*- und *Viewpoint*:** Diese ermöglicht die flexible Darstellung von Systeminformationen aus unterschiedlichen Stakeholder-Perspektiven.
2. **Analysefähigkeit:** Erlaubt die Integration und Ausführung analytischer Modelle zur Unterstützung von Systembewertungen und Entscheidungsprozessen.
3. **Verifikationsfähigkeit:** Durch formale Verifikationsmechanismen kann sichergestellt werden, dass Systeme korrekt spezifiziert und die Anforderungen erfüllt werden.

Die Architektur von SysML v2 basiert auf einer mehrschichtigen Sprachstruktur, die in Bild 2-7 dargestellt ist.

Das zugrundeliegende KerML definiert die abstrakte Syntax, auf der SysML v2 aufbaut. Die *Systems Library* erweitert diese Basisspezifikation um Systembezogene Konstrukte. Zusätzlich ermöglichen sogenannte *Domain Libraries* die Einbindung domänenspezifischer Referenzmodelle, beispielsweise zur Modellierung von physikalischen Größen oder Analysemodellen (OMG Systems Modeling Language, 2024).

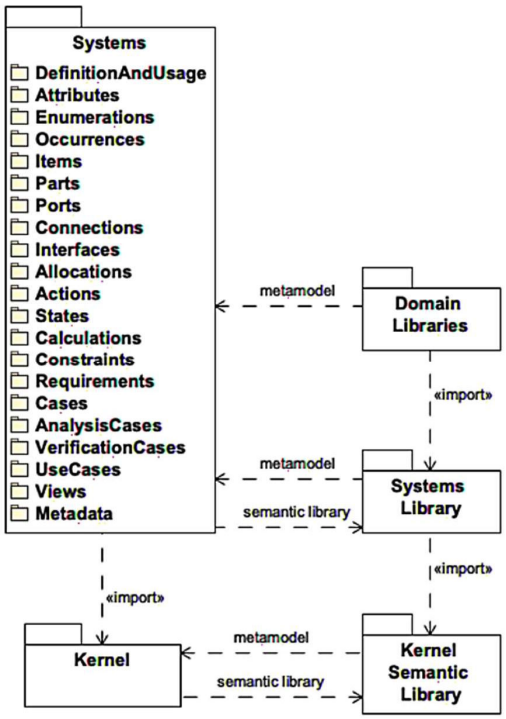


Bild 2-7 SysML v2 Spracharchitektur (OMG Systems Modeling Language, 2024)

2.3.3 Weiterentwicklung der Terminologie und Struktur von SysML v1 zu SysML v2

SysML v2 wurde mit dem Ziel entwickelt, bestehende Schwächen von SysML v1 gezielt zu adressieren. Im Vergleich zur Vorgängerversion SysML v1 bringt SysML v2 nicht nur strukturelle und terminologische Änderungen mit sich, sondern adressiert auch zentrale Schwächen und Herausforderungen der bisherigen Modellierungspraxis (Friedenthal, 2024). Wesentliche Neuerungen bestehen in folgenden Punkten:

- Einführung einer textuellen Notation, die parallel zur grafischen Notation verwendet werden kann,
- Unterstützung von Systemvarianten, Analysefällen, Verifikationsfällen und Sichten,
- Integration über eine standardisierte *Application Programming Interface* (API) und ein zentrales Modellserver-Konzept,
- und die Möglichkeit zur domänenspezifischen Erweiterung durch Bibliothekskonzepte.

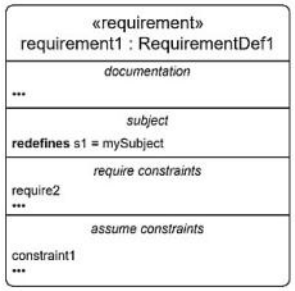
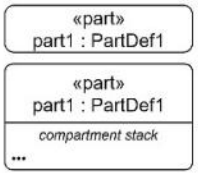
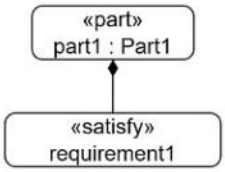
Ein zentrales Unterscheidungsmerkmal liegt in der technologischen Basis: Während SysML v1 auf UML aufbaut, basiert SysML v2 auf KerML, einer speziell entwickelten Metamodellierungssprache. KerML selbst wurde auf Grundlage von UML und der *Web Ontology Language* (OWL) konzipiert und bietet eine stärkere formale Grundlage für die Modellierung. Diese Neuausrichtung zielt darauf ab, die Semantik der Sprache zu vereinheitlichen und damit die Interpretierbarkeit von Modellen zu verbessern (Friedenthal, 2024).

Ein weiteres wesentliches Merkmal von SysML v2 ist die optionale textuelle Notation, die als Ergänzung zur grafischen Sichtweise dient. Beide Darstellungen beruhen auf demselben zugrunde liegenden Modell und sind vollständig synchronisierbar. Dadurch ergeben sich für Anwender flexible Möglichkeiten der Modellbearbeitung und -dokumentation, angepasst an individuelle Präferenzen oder Werkzeugeinsatz (OMG Systems Modeling Language, 2024). Die neue Struktur von SysML v2 erleichtert außerdem die Integration mit modernen Werkzeugketten und Arbeitsweisen, wie beispielsweise Git-basierten Kollaborationsprozessen (Ahlbrecht et al., 2024). Tabelle 2-2 umfasst ein Beispiel, das sowohl die grafischen als auch die textuellen Notationen der SysML v2 verdeutlicht.

Die standardisierte API und das Modellserver-Konzept eröffnen neue Möglichkeiten zur Werkzeugintegration sowie zur Entwicklung spezialisierter MBSE-Tools – auch durch kleinere Anbieter, akademische Einrichtungen oder gemeinschaftsbasierte *Open-Source*-Projekte. Die Unterstützung durch etablierte Toolhersteller wird ebenfalls erwartet, da SysML v2 mit seinen erweiterten Fähigkeiten und

Interoperabilitätsansätzen einen zukunftsfähigen Standard darstellt (Ahlbrecht et al., 2024).

Tabelle 2-2: Beispiel für die grafischen und textuellen Notationen von SysML v2 (OMG Systems Modeling Language, 2024)

Elementname	Grafische Notation	Textuelle Notation
Requirement		`requirement requirement1 : RequirementDef1 { doc /* ... */ subject redefines s1 = mySubject; require require2; assume constraint1; }`
Part		`part part1 : PartDef1;` `part part1 : PartDef1 { /* members */ }`
Satisfy		`requirement requirement1 : Requirement1;` `part part1 : Part1 { satisfy requirement1; }`

Die sprachliche Vereinfachung spielt ebenfalls eine zentrale Rolle in SysML v2. Wie in Tabelle 2-3 dargestellt, wurden Begriffe wie *„part property“* oder *„block“* durch klarere und intuitivere Bezeichnungen wie *„part“* und *„part def“* ersetzt. Diese sprachliche Konsistenz erleichtert nicht nur das Erlernen der Sprache, sondern verbessert auch die Verständlichkeit und Anwendbarkeit im praktischen Einsatz (Friedenthal, 2024).

Die textuelle Notation von SysML v2 wird in dieser Arbeit bevorzugt betrachtet, da sie eng mit dem in den nachfolgenden Kapiteln entwickelten methodischen Ansatz verknüpft ist.

Tabelle 2-3 Vergleich der Terminologie zwischen SysML v2 und SysML v1 (Ausschnitt) (Friedenthal, 2024)

SysML v2	SysML v1
part / part def	part property / block
attribute / attribute def	value property / value type
port / port def	proxy port / interface block
action / action def	action / activity
state / state def	state / state machine
constraint / constraint def	constraint property / constraint block
requirement / requirement def	requirement
connection / connection def	connector / association block
view / view def	view

2.4 Cameo Systems Modeler mit SysML v2

Cameo Systems Modeler (CSM) ist eine plattformübergreifende Modellierungsumgebung für das MBSE. Sie wird von *Dassault Systèmes* (ehemals *No Magic*) entwickelt und gilt als eines der führenden Werkzeuge in der MBSE-Praxis. Die Software ermöglicht die standardkonforme Modellierung gemäß SysML und unterstützt Ingenieurteams bei der Modellierung technischer Systeme (Dassault Systèmes, 2023). Laut Herstellerbeschreibung unterstützt CSM insbesondere:

- Die Durchführung von Analysen zur Bewertung von Entwurfsentscheidungen und zur Verifikation von Anforderungen,
- die kontinuierliche Konsistenzprüfung von Modellen sowie
- die Fortschrittsüberwachung anhand definierter Metriken.

Zudem betont der Hersteller, dass CSM speziell auf hochkomplexe, regulierte Industrien wie Luft- und Raumfahrt, Verteidigung oder Automobil ausgelegt ist. Es unterstützt die Einhaltung strenger Normen und erleichtert die Dokumentation für Projektmanagement und Zertifizierungsprozesse (Dassault Systèmes, 2023).

Die im Rahmen dieser Arbeit verwendete Version von CSM wurde vom DLR bereitgestellt und enthält das SysML v2-Plugin (Bild 2-8).

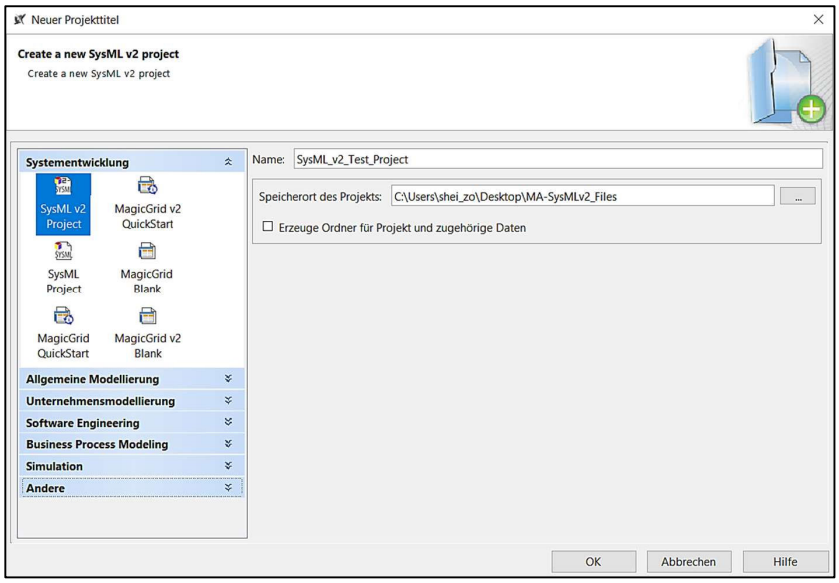


Bild 2-8 CSM SysML v2-Plugin (eigener Screenshot aus CSM, SysML v2-Plugin)

Dabei handelt es sich ausdrücklich um eine Vorabversion, die sich noch in der Entwicklungsphase befindet. Nach dem Start der Software erscheint ein Warnhinweis (Bild 2-9):

„Not for Use in Production:

The SysML v2 Plugins are pre-released version and cannot be used for production. Please use them only for testing and evaluation purposes.

Note that this version does not support project migration, so projects created may not be compatible with future releases.

While support through the official Dassault Systèmes Support channel is not available, we welcome your feedback in the 3DSwym SysMLv2 community."

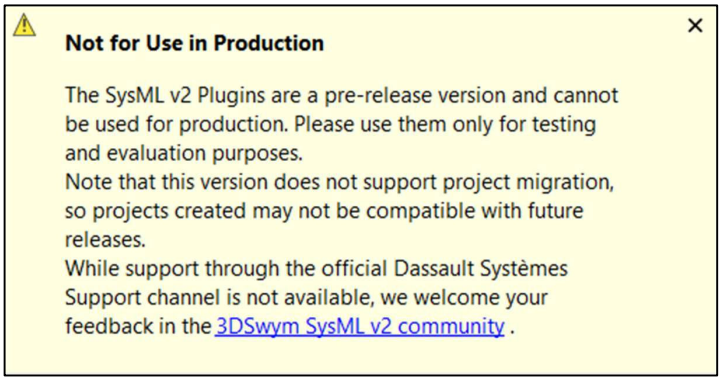


Bild 2-9 Warnhinweis beim Start des SysML v2-Plugins in CSM (eigener Screenshot aus CSM, Warnhinweis)

Diese Einschränkungen machen deutlich, dass die aktuell verfügbare Version noch nicht für produktive Zwecke vorgesehen ist. Die Unterstützung offizieller Supportkanäle entfällt und eine Rückwärtskompatibilität zukünftiger Versionen ist nicht gewährleistet. Dennoch bietet diese Vorabversion bereits die wesentlichen Funktionen, um ein Beispielsystem zu modellieren und erste Erfahrungen mit SysML v2 zu sammeln.

Im Rahmen dieser Arbeit wird ein einfaches UAV-Beispielsystem in CSM modelliert, wobei besonderes Augenmerk auf die Verwendung der textuellen Notation gelegt wird. SysML v2 erlaubt die parallele Nutzung beider Darstellungsformen – graphisch und textuell, was in der Benutzeroberfläche von CSM direkt sichtbar ist.

Bild 2-10 zeigt ein Beispiel, in dem die textuelle und grafische Darstellung eines modellierten Teils nebeneinander angezeigt werden. Dieses Beispiel verdeutlicht die synchrone Visualisierung beider Notationen im SysML v2-Plugin von CSM. Weitere Details zur Projekterstellung und zu den spezifischen Einschränkungen der SysML v2-Integration in CSM werden in Kapitel 4.2 beschrieben.

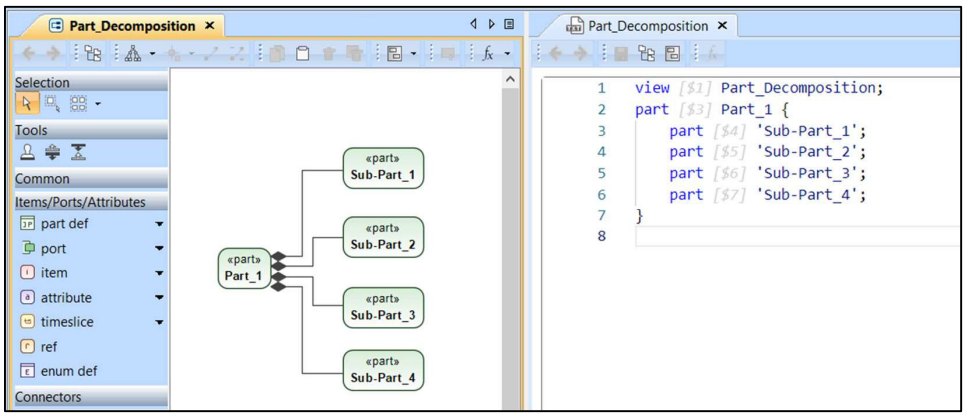


Bild 2-10 Beispiel für die Darstellung einer Teilzerlegung in SysML v2 (eigener Screenshot aus CSM, grafische und textuelle Notation)

2.5 Kollaboration in MBSE

Die modellbasierte Systementwicklung erfordert die enge Zusammenarbeit interdisziplinärer Teams über den gesamten Systemlebenszyklus hinweg. Im Gegensatz zu dokumentenbasierten Ansätzen bietet MBSE eine zentrale Wissensrepräsentation, die als Referenz für Anforderungen, Architektur, Verifikation und Validierung dient (Haberfellner et al., 2019). Damit diese Vorteile jedoch wirksam werden, müssen alle Beteiligten effizient und konsistent auf ein gemeinsames Modell zugreifen und daran mitarbeiten können (May & Zerwas, 2025).

In der Praxis zeigt sich, dass Kollaboration im MBSE mit einer Vielzahl von Herausforderungen verbunden ist – sowohl organisatorischer als auch technischer

Natur. Unterschiedliche Begriffsverständnisse, fehlende Synchronisation zwischen Teilmodellen und unzureichende Versionskontrolle sind nur einige Beispiele. Diese Problematik wird in Kapitel 2.5.1 detailliert betrachtet.

In Kapitel 2.5.2 wird anschließend aufgezeigt, wie agile Prinzipien und *DevOps*-Ansätze (*Development & Operations*) in der Softwareentwicklung als Antwort auf ähnliche Herausforderungen entstanden sind und wie diese Denkweise in Form von Agile MBSE auf die modellbasierte Systementwicklung übertragen werden kann. In Kapitel 2.5.3 wird Git als Versionsverwaltungssystem vorgestellt und darauf eingegangen, weshalb es für die modellbasierte Kollaboration in diesem Kontext ausgewählt wurde und welche Vorteile sich daraus ergeben können.

2.5.1 Herausforderungen der Kollaboration in MBSE

Die kollaborative modellbasierte Systementwicklung (CMBSE) steht vor vielfältigen Herausforderungen, die in der Fachliteratur umfangreich diskutiert werden. Auf Basis einer systematischen Literaturrecherche lassen sich zehn wiederkehrende Problemfelder identifizieren, die nachfolgend zusammengefasst werden.

Konsistenz und Aktualität im Informationsaustausch:

Ein zentrales Problem stellt die konsistente Versionierung und der Austausch von Modellen dar. Insbesondere in verteilten Entwicklungsnetzwerken ist es erforderlich, dass Partner wie OEMs und Zulieferer modellbasierte Beschreibungen, Anforderungen und Lösungen aufeinander abgestimmt und in nachvollziehbarer Version austauschen (prostep ivip Association, 2023). Die hohe Dynamik in frühen Entwicklungsphasen führt zudem zu ständigen Anpassungen der Modelle, die zeitnah mit allen Beteiligten synchronisiert werden müssen (Li et al., 2024).

Zugänglichkeit und Schutz sensibler Daten:

MBSE-Plattformen wie 3DX von *Dassault Systèmes* etablieren Modelle als *Single Point of Truth* und verbessern dadurch die Zusammenarbeit über Domänengrenzen hinweg. Dennoch bestehen Einschränkungen bei der Zugänglichkeit für externe Stakeholder, da Schnittstellen nur innerhalb der Plattform aktiviert werden können (May & Zerwas, 2025). Parallel dazu erfordert die Sensibilität der Entwicklungsdaten eine selektive Offenlegung, bei der klar definiert ist, welche Informationen in welchem Umfang zugänglich sind (Li et al., 2024; Wouters et al., 2017).

Semantische Interoperabilität und gemeinsame Sprache:

Ein häufig unterschätztes Hindernis ist die mangelnde semantische Konsistenz bei der Verwendung zentraler Begriffe. Unterschiedliche Disziplinen interpretieren Begriffe wie „Funktion“ oder „System“ teils divergierend, was Missverständnisse bei der Modellintegration begünstigt (Wouters et al., 2017).

Koordination verteilter Modellierungsaktivitäten:

Die Orchestrierung von Modellierungsprozessen über Fach- und Organisationsgrenzen hinweg erfordert eine präzise Abstimmung. Dabei sollen sowohl Kohärenz als auch individuelle Arbeitsweisen berücksichtigt werden (Wouters et al., 2017). Unterstützt werden muss dies durch rollenbasierte Zugriffskonzepte, die sowohl Datenschutz als auch Informationsrelevanz berücksichtigen (Wouters et al., 2017).

Modellvalidität und Regelkonformität:

Ein weiteres zentrales Thema betrifft die Einhaltung domänenspezifischer Regeln und Normen innerhalb der Modelle. Diese müssen explizit formuliert und maschinenlesbar validierbar gemacht werden, um Inkonsistenzen frühzeitig zu erkennen (Wouters et al., 2017).

Rückverfolgbarkeit und Änderungsmanagement:

Die Nachvollziehbarkeit von Anforderungen, Entwurfsentscheidungen und Änderungen über disziplinäre Grenzen hinweg ist essenziell für Qualität und Auditierbarkeit. Dies setzt geeignete Mechanismen zur Dokumentation, Kommentierung und Versionierung voraus (Wouters et al., 2017).

Systemisches Denken als kulturelle Herausforderung:

Abschließend sei auf ein übergreifendes Problem hingewiesen: Die Umstellung von einer komponentenbasierten hin zu einer systemischen Denkweise. MBSE erfordert, dass Beteiligte funktional und nutzerorientiert denken, anstatt sich auf vorhandene Produktstrukturen zu stützen. Dies impliziert tiefgreifende organisatorische Veränderungen sowie entsprechende Schulungs- und Transformationsprozesse (prostep ivip Association, 2023).

Zur besseren Nachvollziehbarkeit werden in Tabelle 2-4 zentrale Herausforderungen der kollaborativen modellbasierten Systementwicklung (CMBSE) zusammengefasst. Die dargestellten Problemfelder basieren auf einer systematischen Literaturrecherche, sind thematisch gruppiert und jeweils mit Quellenangaben versehen.

Die in diesem Abschnitt identifizierten Herausforderungen bilden eine zentrale Grundlage für die Analyse der konkreten Umsetzung kollaborativer MBSE-Praktiken im Rahmen dieser Arbeit. Sie dienen als Referenzrahmen, um die in der Forschungsumgebung des DLR beobachteten Problemstellungen systematisch einordnen und bewerten zu können. Ein entsprechender Vergleich erfolgt in Kapitel 3.5, das die Herausforderungen der Zusammenarbeit im Kontext der betrachteten Forschungsumgebung beschreibt.

Tabelle 2-4: Übersicht der Herausforderungen in CMBSE mit Zuordnung zu Quellen und thematischen Kategorien

Herausforderung	**Kategorie**	**Quelle**
Dynamik & Aktualisierung	Änderungsmanagement	Li et al., (2024)
Rückverfolgbarkeit	Änderungsmanagement	Wouters et al., (2017)
Informationssicherheit	Zugriff & Sicherheit	Li et al., (2024)
Kontrollierte Offenlegung	Zugriff & Sicherheit	Wouters et al., (2017)
Konsistenter Austausch	Informationsmanagement	prostep ivip Association, (2023)
Regelkonformität im Modell	Domänenspezifische Validierung	Wouters et al., (2017)
Zugriffsbeschränkung in 3DX	Plattformtechnische Grenzen	May & Zerwas, (2025)
Begriffsdivergenz	Semantik & Interoperabilität	Wouters et al., (2017)
Koordination über Rollen hinweg	Orchestrierung	Wouters et al., (2017)
Systemisches Denken	Organisation & Kultur	prostep ivip Association, (2023)

2.5.2 Agile MBSE

In der Softwareentwicklung haben sich agile Methoden als wirkungsvolle Antwort auf Herausforderungen der Zusammenarbeit etabliert. Das Agile Manifest nach Beck et al. (2001) betont Werte wie Zusammenarbeit, Reaktionsfähigkeit auf Veränderung sowie funktionsfähige Ergebnisse statt umfassender Dokumentation. Diese Prinzipien spiegeln sich in verschiedenen agilen Methoden wider, wie z. B. *Scrum*, *Extreme Programming* (XP) oder *Feature Driven Development* (FDD), die allesamt auf inkrementeller Entwicklung und iterativen Zyklen basieren (Alsaqqa et al., 2020). Ein zentrales Element vieler agiler Methoden sind sogenannte *Sprints* – kurze, festgelegte Iterationen, in denen Planung, Entwicklung und Überprüfung stattfinden (Nyembe et al., 2023). *Scrum*, als eine der populärsten agilen Methoden, definiert klare Rollen, Verantwortlichkeiten und Abläufe zur kontinuierlichen Verbesserung und Anpassung des Produkts (Schwaber & Sutherland, 2020).

Parallel zur agilen Bewegung entwickelte sich *DevOps* als erweiterter organisatorischer Ansatz, der insbesondere die Integration von Entwicklung und Betrieb fokussiert. Ziel ist eine beschleunigte, zuverlässige und kontinuierliche

Softwareauslieferung. Zentrale *DevOps*-Prinzipien wie *Continuous Integration* (CI), *Continuous Deployment* (CD), automatisierte Prüfverfahren, Systemüberwachung und Rückmeldezyklen tragen maßgeblich zur Effizienzsteigerung bei (Jayaraman & Rastogi, 2025; Kim et al., 2016). Darüber hinaus fördert *DevOps* eine Kultur geteilter Verantwortung, in der funktionsübergreifende Teams gemeinsam für Qualität und Geschwindigkeit der Lieferung verantwortlich sind. Diese Herangehensweise reduziert sowohl die mittlere Wiederherstellungszeit (MTTR) als auch die Fehlerquote in Produktionsumgebungen und verbessert die Reaktionsfähigkeit auf Marktveränderungen (Jayaraman & Rastogi, 2025).

Ein typischer agiler Entwicklungsprozess besteht aus den Schritten: Planung, Entwurf, Entwicklung, Erprobung, Bereitstellung, Überprüfung und Veröffentlichung. Diese Phasen werden zyklisch durchlaufen, sodass in jeder Iteration Feedback einfließt und Anpassungen möglich sind. Bild 2-11 zeigt diesen iterativen Ablauf exemplarisch (exapp.ca, 2024).

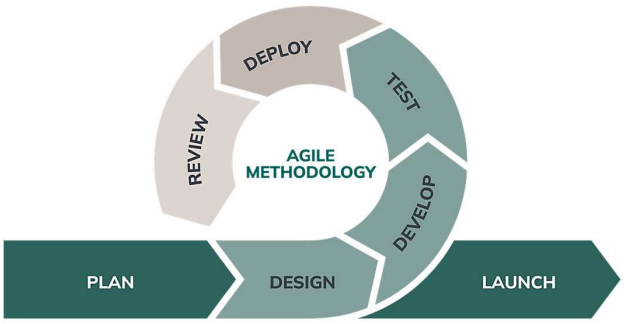


Bild 2-11 Agile Methodologie (exapp.ca, 2024)

Auch in der modellbasierten Systementwicklung hat sich inzwischen ein agiler Denkansatz etabliert: *Agile Model-Based Systems Engineering* (AMBSE). AMBSE kombiniert die Prinzipien agiler Softwareentwicklung mit modellbasierten Systementwicklungsprozessen. Im Bereich der Luftfahrtsystementwicklung zeigen Fallstudien, wie agile Prinzipien erfolgreich auf frühe Phasen der Systementwicklung angewendet werden können. Krupa (2019) beschreibt z. B. die Kombination von OOSEM (*Object-Oriented Systems Engineering Method*) mit SysML v1 zur iterativen Konzeptentwicklung (Bild 2-12). Dabei entsteht ein kontinuierlicher Entwicklungsprozess, der Flexibilität in der Modellierung mit hoher Nachvollziehbarkeit verbindet und gleichzeitig die Anforderungen regulatorischer Standards wie ARP4754A und ARP4761 unterstützt.

Darüber hinaus wird der AMBSE-Prozess in Bild 2-13 als SysML-Aktivitätsdiagramm dargestellt. Die Aktivitäten innerhalb des Diagramms repräsentieren zentrale Aufgaben im Rahmen des MBSE. Nachdem eine konkrete System- bzw. Subsystemarchitektur entworfen wurde, erfolgt ein Vergleich mit alternativen Konzepten. Sobald eine Architekturvariante als geeignet eingestuft wird, wird sie an die integrierten Produktentwicklungsteams (IPDT) zur weiteren Ausarbeitung im

Rahmen des vorläufigen Designs übergeben. Während des gesamten Entwicklungsprozesses dienen die Luftfahrtstandards ARP4754A und ARP4761 als Grundlage zur sys-tematischen Anforderungserhebung und Sicherheitsanalyse (Krupa, 2019).

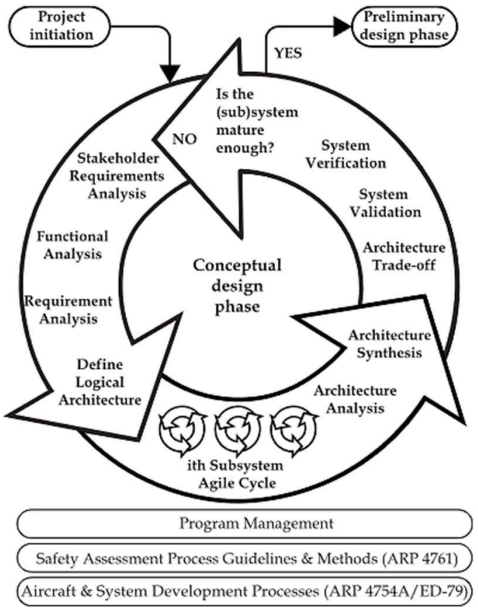


Bild 2-12 Agiler Systementwicklungsprozess für die Flugzeugkonzeption nach Krupa, (2019)

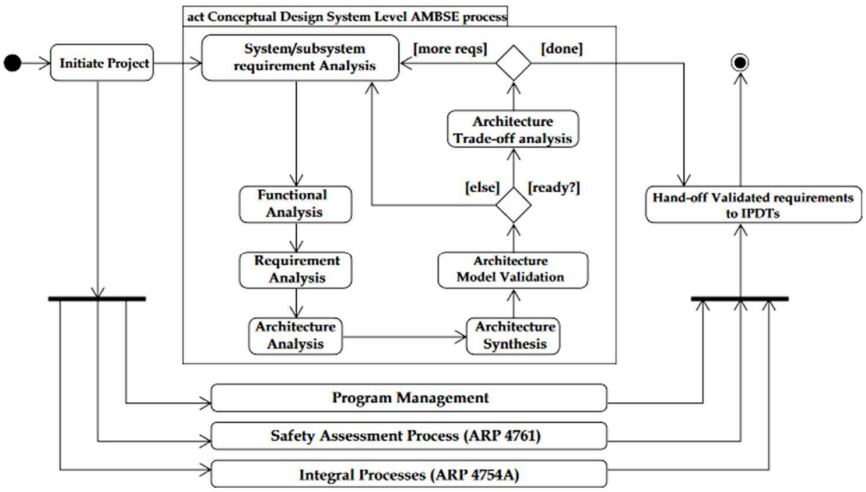


Bild 2-13 AMBSE Lieferprozess während der Konzeptionsphase nach Krupa, (2019)

2.5.3 Git als Versionskontrollsystem

Git ist ein verteiltes Versionskontrollsystem, das ursprünglich zur Koordination der Entwicklung des *Linux*-Kernels entwickelt wurde und sich seither als führendes Werkzeug für die Verwaltung von Quellcodeänderungen etabliert hat (Spinellis, 2012). Im Gegensatz zu zentralisierten Systemen wie *Subversion* oder CVS erlaubt Git es mehreren Entwicklern, parallel an lokalen Ablagen (engl. *„Repository“* pl. *„Repositories“*) zu arbeiten, die bei Bedarf mit einem zentralen entfernten *Repository* synchronisiert werden. Jeder *„Commit“* stellt dabei eine Momentaufnahme des Projekts dar, die durch einen eindeutigen Hash identifiziert wird. Durch die Nutzung von sogenannten Zweigen (engl. *„Branch“* pl. *„Branches“*) können Funktionen, Fehlerbehebungen oder experimentelle Änderungen unabhängig voneinander entwickelt und später selektiv zusammengeführt werden (Ghodke & Chavan, 2024).

Diese dezentrale Struktur ermöglicht nicht nur eine flexible Arbeitsweise, sondern unterstützt auch komplexe Workflows, wie sie in modernen Softwareentwicklungsprojekten üblich sind. Entwickler können lokale Änderungen vor dem *Commit* im sogenannten *„Staging“*-Bereich vorbereiten und die Historie eines Projekts jederzeit vollständig lokal nachvollziehen – selbst ohne Internetverbindung. Relevante Git-Befehle – wie *´git init´*, *´git add´*, *´git commit´*, *´git status´*, *´git log´* oder *´git push´* ermöglichen eine granulare Kontrolle über den Entwicklungsprozess (Ghodke & Chavan, 2024). Eine vollständige Übersicht zentraler Git-Befehle findet sich in Anhang A2.

Im Kontext von *DevOps* spielt Git eine zentrale Rolle als verbindendes Element zwischen Entwicklung und Betrieb. Als Teil einer integrierten Werkzeugkette lässt sich Git nahtlos mit weiteren *DevOps*-Werkzeugen wie GitLab, Jenkins oder Docker kombinieren, um CI/CD und automatisierte Testprozesse zu ermöglichen. Diese Integration ist essenziell, um eine durchgängige Rückverfolgbarkeit, eine konsistente Entwicklungsumgebung und eine hohe Automatisierung zu gewährleisten (Jayaraman & Rastogi, 2025).

Neben der lokalen Nutzung von Git bieten Plattformen wie GitHub und GitLab erweiterte Funktionen zur Verwaltung und Kollaboration von Git-*Repositories*. Wie Spinellis (2012) beschreibt, übernimmt GitHub als Drittanbieter zentrale Aufgaben wie das Hosting, die Versionskontrolle, die Benutzerverwaltung sowie Sicherheits- und Zugriffsrichtlinien. Dadurch wird insbesondere die Zusammenarbeit in *Open-Source*-Projekten erleichtert, etwa durch die Möglichkeit, *Merge Requests* (MR) einzureichen oder Änderungen direkt über eine webbasierte Oberfläche vorzunehmen. GitHub fördert so die kollaborative Entwicklung durch ein integriertes Ökosystem mit Funktionen wie der Nachverfolgung von Aufgaben, Projektdokumentationen und Quellcodeüberprüfungen. GitLab bietet ähnliche Funktionalitäten und integriert Werkzeuge für CI/CD direkt in die Plattform. Dadurch eignet sich

GitLab besonders gut für die Umsetzung von *DevOps*-Prinzipien in einer konsolidierten Entwicklungsumgebung. Beide Plattformen haben sich somit als zentrale Bausteine in modernen Softwareentwicklungsprozessen etabliert und tragen wesentlich zur effizienten Anwendung von Git in verteilten Teams bei (Spinellis, 2012).

Im Gegensatz zu SysML v1, dessen Modelle primär in binären Formaten gespeichert werden, ermöglicht SysML v2 eine Serialisierung in menschenlesbaren Textdateien. Dadurch lassen sich Modelle mit Git auf dieselbe Weise versionieren wie Quellcode in der Softwareentwicklung. Dies eröffnet neue Potenziale für verteilte Zusammenarbeit, Modellnachverfolgbarkeit und Wiederverwendung von Modellartefakten – ohne auf proprietäre Datenformate angewiesen zu sein.

Im Rahmen dieser Arbeit wird Git in Verbindung mit GitLab als Versionskontrollsystem eingesetzt, um modellbasierte Systementwicklungsartefakte zu verwalten. Dabei liegt ein besonderer Fokus auf der Integration der textuellen SysML v2-Notation, wie sie durch das CSM SysML v2-Plugin unterstützt wird. Die Möglichkeit zur Nutzung einer klar strukturierten, textuellen Repräsentation der Modelle erlaubt es, diese in Versionskontrollsysteme einzubinden, wie es bei Quellcode üblich ist.

3 Umgebungsanalyse und Anforderungen an den Ansatz

Dieses Kapitel widmet sich der Analyse der Forschungsumgebung und der Ableitung relevanter Anforderungen an den zu entwickelnden Git-basierten Kollaborationsprozess. Ziel ist es, die Relevanz und Anwendbarkeit des Ansatzes im Kontext der modellbasierten Flugsystementwicklung zu fundieren. Methodisch ist dieses Kapitel dem *Relevance Cycle* der DSR-Methodologie nach Hevner et al. (2004) zuzuordnen. Dieser Zyklus betont die systematische Erfassung praxisrelevanter Problemstellungen sowie die Ableitung von Anforderungen, die als Input für die Gestaltung und Evaluation von Artefakten dienen.

Im ersten Schritt wird das methodische Vorgehen beschrieben, mit dem die Forschungsumgebung erhoben und analysiert wurde (**Kapitel 3.1**). Darauf folgt eine Beschreibung der organisatorischen Rahmenbedingungen des Projektteams am DLR (**Kapitel 3.2**).

Kapitel 3.3 stellt die Ergebnisse einer teaminternen Umfrage vor, mit der bestehende Kompetenzen, Werkzeuge und Prozesse identifiziert wurden. Diese Ergebnisse bilden die Grundlage für die Entwicklung einer Rollenzuweisung gemäß ARP4754B und ISO/IEC 15288, die in **Kapitel 3.4** behandelt wird. **Kapitel 3.5** schließt die Analyse mit einer strukturierten Darstellung der zentralen Kollaborationsherausforderungen ab. Diese bilden die Basis für die in Kapitel 4 zu entwickelnden Anforderungen an den Git-basierten Kollaborationsansatz.

3.1 Methodisches Vorgehen zur Umgebungsanalyse

Die Analyse der Forschungsumgebung basiert auf einer empirischen Untersuchung innerhalb der Forschungsgruppe am DLR. Ziel war es, bestehende Strukturen, Prozesse und Herausforderungen der modellbasierten Zusammenarbeit im Team systematisch zu erfassen.

Hierzu wurde eine standardisierte Umfrage mit 18 geschlossenen und offenen Fragen entwickelt und durchgeführt. Elf Teammitglieder nahmen daran teil. Die Umfrage zielte darauf ab, Kompetenzen, genutzte Werkzeuge, Rollenverständnisse sowie Probleme in der täglichen Zusammenarbeit zu identifizieren.

Ergänzend wurde ein strukturierter Rollenzuweisungsprozess durchgeführt. Grundlage bildete eine RACI-Matrix, in der die im Team identifizierten Rollen systematisch den Aktivitäten der Systementwicklungsprozesse gemäß ARP4754B und ISO/IEC 15288 zugeordnet wurden. Diese methodische Kopplung ermöglicht eine fundierte Einordnung von Verantwortlichkeiten in Bezug auf etablierte Entwicklungsstandards.

Die identifizierten Herausforderungen aus Umfrage und Rollenanalyse dienen in den nachfolgenden Kapiteln als Grundlage für die Ableitung funktionaler und nicht-funktionaler Anforderungen an den Kollaborationsprozess.

3.2 Beschreibung der Forschungsumgebung

Die vorliegende Masterarbeit wurde im Rahmen einer Forschungsaktivität am DLR Institut für Flugsystemtechnik, in der Abteilung „Sichere Systeme & *Systems Engineering*“ (FT-SSY) in Braunschweig durchgeführt. Die Abteilung beschäftigt sich mit der Entwicklung sicherheitskritischer Systeme im Bereich der Flugsystemtechnik sowie mit der Anwendung und Weiterentwicklung modellbasierter Methoden des *Systems Engineerings*.

Die Forschungsgruppe besteht aus einem Gruppenleiter und zehn wissenschaftlichen Mitarbeitenden, von denen sich die Mehrheit in der Promotionsphase befindet. Die Mitarbeitenden sind in verschiedene nationale und internationale Forschungsprojekte eingebunden. Inhaltlich liegen die Arbeitsschwerpunkte überwiegend in der frühen Phase der Systementwicklung, insbesondere in der Systemkonzeption, dem Entwurf und der Analyse von Flugsystemen. Diese Tätigkeiten lassen sich im V-Modell nach ARP4754B primär auf der linken Seite verorten, also im Bereich der Anforderungsdefinition, Konzeptausarbeitung und Architekturentwicklung. Darüber hinaus sind die Mitarbeitenden regelmäßig mit wissenschaftlichen Publikationen und Konferenzbeiträgen aktiv.

Neben den wissenschaftlichen Mitarbeitenden betreuen die Teammitglieder auch Bachelor- und Masterarbeiten. Die studentischen Hilfskräfte sowie Abschlussarbeitsstudierende sind jedoch nicht Teil der systematischen Analyse in dieser Arbeit, da der Fokus auf den dauerhaft im Projekt eingebundenen Fachkräften liegt.

In der täglichen Arbeit kommen verschiedene Softwarewerkzeuge zum Einsatz. Dazu zählen unter anderem modellbasierte Entwicklungstools wie CSM, mit dem auf Basis von SysML v1 modelliert wird. Darüber hinaus werden weitere Werkzeuge zur Dokumentation, Versionskontrolle, Simulation oder Kommunikation verwendet. Eine einheitliche, integrierte Toollandschaft für die kollaborative Systemmodellierung ist bislang jedoch nicht etabliert.

Ziel der im Rahmen dieser Arbeit durchgeführten Erhebung ist es daher, die bestehende Arbeitsweise hinsichtlich Kompetenzen, genutzter Werkzeuge und Formen der Zusammenarbeit zu erfassen. Die daraus gewonnenen Erkenntnisse bilden die Grundlage für die Ableitung von Anforderungen an einen Git-basierten Kollaborationsprozess im modellbasierten *Systems Engineering*.

3.3 Analyse der Teamumfrage zur Zusammenarbeit

Zur Untersuchung der aktuellen Modellierungs- und Kollaborationspraxis innerhalb der Forschungsgruppe wurde eine Teamumfrage durchgeführt. Ziel war es, ein fundiertes Verständnis über die Rollenverteilung, eingesetzte Modellierungsmethoden, etablierte Kollaborationsprozesse sowie bestehende Herausforderungen und Verbesserungspotenziale im Umgang mit Systemmodellen zu gewinnen.

3.3.1 Methodik der Umfrage

Die Umfrage wurde mithilfe von *Microsoft Forms* erstellt und im Rahmen eines Online-Meetings mit dem Team durchgeführt. Insgesamt nahmen elf Personen teil – acht davon während des Meetings und drei im Nachgang. Ein neues Teammitglied konnte nicht alle Fragen beantworten, sodass einige Fragen nur zehn statt elf Antworten aufweisen.

Der Fragebogen umfasste 18 Fragen, davon 17 geschlossene und eine offene Frage. Die Formulierungen wurden so gewählt, dass sie gezielt auf die erwarteten Erkenntnisse ausgerichtet sind. Zur systematischen Auswertung wurden die Ergebnisse in fünf thematische Kategorien eingeordnet:

1. Team- und Projektinformationen
2. Rollen und Verantwortlichkeiten in der Systementwicklung
3. Modellierungspraxis und SysML-Nutzung
4. Kollaboration und Versionskontrolle
5. Erwartungen und Verbesserungsideen

Die Auswertung der Ergebnisse erfolgte mithilfe von *Microsoft Excel*. Eine vollständige Auflistung aller Fragen samt Antwortverteilungen ist im Anhang A3 dokumentiert. Im folgenden Abschnitt (3.3.2) werden ausgewählte Ergebnisse vorgestellt und interpretiert.

3.3.2 Ergebnisse und Auswertung

Team- und Projektinformationen

Die Mehrheit der Teilnehmenden arbeitet aktuell an zwei bis drei Projekten parallel. Hinsichtlich der Zusammenarbeit zeigt sich ein heterogenes Bild: Während einige Teammitglieder eigenständig agieren, findet bei anderen ein regelmäßiger Austausch in kleinen oder größeren Gruppen statt. Die Abstimmung zu Systemmodellen erfolgt überwiegend auf monatlicher Basis; ein täglicher Austausch findet derzeit nicht statt.

Rollen und Verantwortlichkeiten

Die Antworten zeigen, dass viele Teammitglieder mehrere Aufgabenbereiche abdecken, insbesondere in den Bereichen Systemarchitektur, Sicherheit sowie Modellierung mit SysML. Gleichzeitig geben lediglich 10 % an, eine klar definierte Modellierungsverantwortung zu besitzen – was auf einen Mangel an Rollenklarheit hinweist. Die Abstimmung beim Aktualisieren von Modellen erfolgt vorwiegend innerhalb des jeweiligen Projektteams, was auf teaminterne Abhängigkeiten schließen lässt.

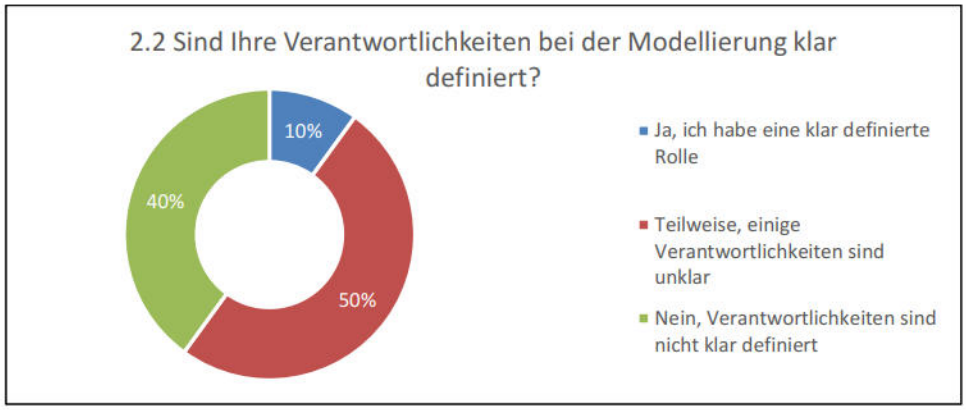


Bild 3-1 Klarheit über Modellierungsverantwortlichkeiten (Frage 2.2)

Modellierungspraxis und SysML-Nutzung

Die Modellierung mit SysML erfolgt in der Mehrheit monatlich, tägliche Modellierung ist im Team nicht etabliert. Als bevorzugtes Werkzeug wird CSM eingesetzt. Ein formell definierter Modellierungsprozess liegt jedoch nicht vor – die meisten orientieren sich nur grob an bestehenden Vorgaben. Die Rückverfolgbarkeit von Modellinhalten wird überwiegend manuell oder halbautomatisiert dokumentiert, wobei die eingesetzten Methoden stark variieren.

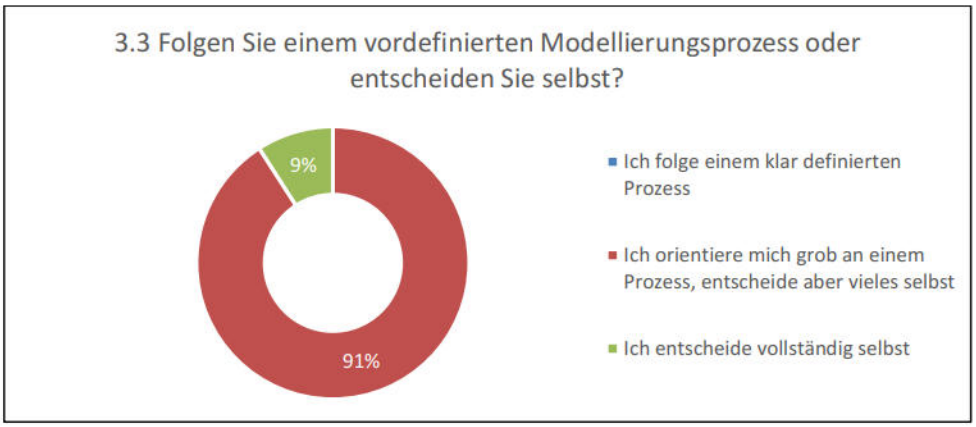


Bild 3-2 Existenz eines definierten Modellierungsprozesses (Frage 3.3)

Kollaboration und Versionskontrolle

Zu den größten Herausforderungen bei der modellbasierten Zusammenarbeit zählen fehlende Prozesse, inkonsistente Änderungen und veraltete Modellstände. Die aktuelle Änderungshistorie wird meist ohne strukturierte Strategie oder manuell gepflegt. Git wird in einzelnen Fällen eingesetzt, jedoch nicht durchgängig im Team verwendet. Positiv hervorzuheben ist, dass Git vielen Teammitgliedern bereits bekannt ist, was eine potenzielle Einführung Git-basierter Workflows erleichtern könnte. Bemerkenswert ist zudem, dass 33 % der Teilnehmenden bereits Versionskonflikte erlebt haben, die typischerweise manuell abgestimmt wurden.

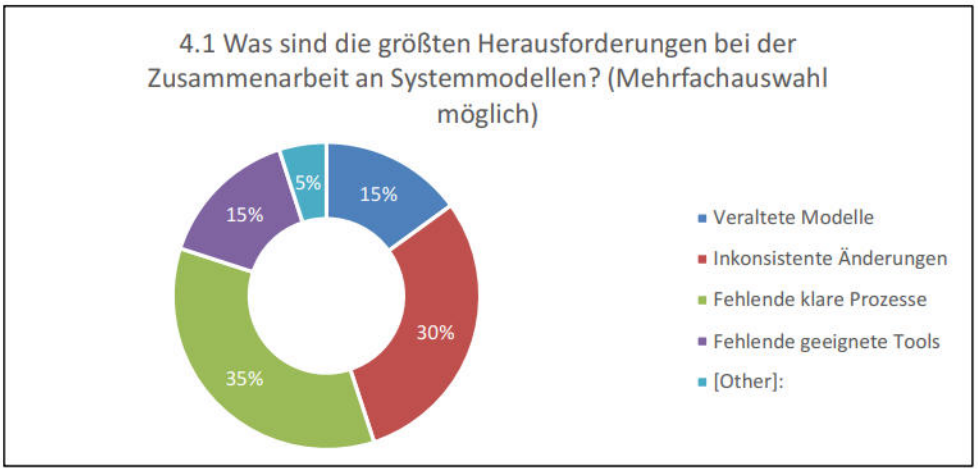


Bild 3-3 Herausforderungen bei der modellbasierten Zusammenarbeit (Frage 4.1)

Erwartungen und Verbesserungsideen

Viele Teammitglieder äußern den Wunsch nach klar definierten Rollen, strukturierter Versionierung und einer automatisierten Synchronisierung. Eine Git-basierte Versionskontrolle wird von der Mehrheit als sinnvoll eingeschätzt, sofern sie benutzerfreundlich implementiert ist.

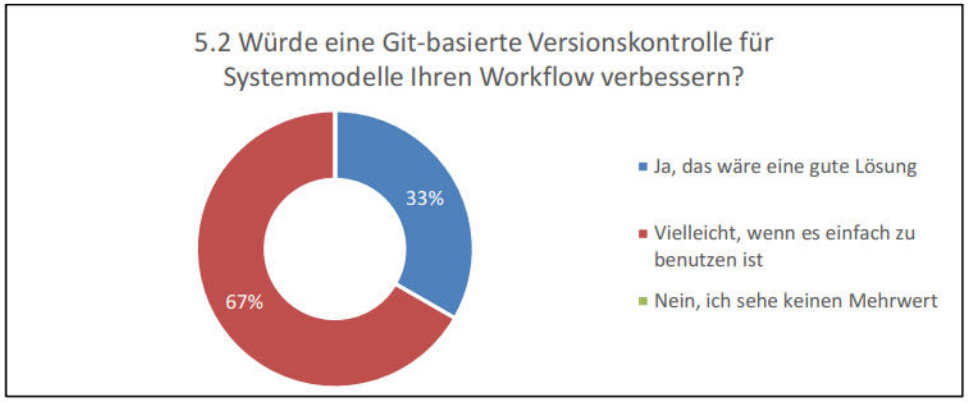


Bild 3-4 Bewertung Git-basierter Versionskontrolle für MBSE (Frage 5.2)

Die Einschätzungen zu agilen Methoden fallen gemischt aus: Einige erkennen deren Potenzial zur Flexibilisierung von Entwicklungsprozessen, während andere die Eignung im forschungsnahen Umfeld kritisch bewerten.

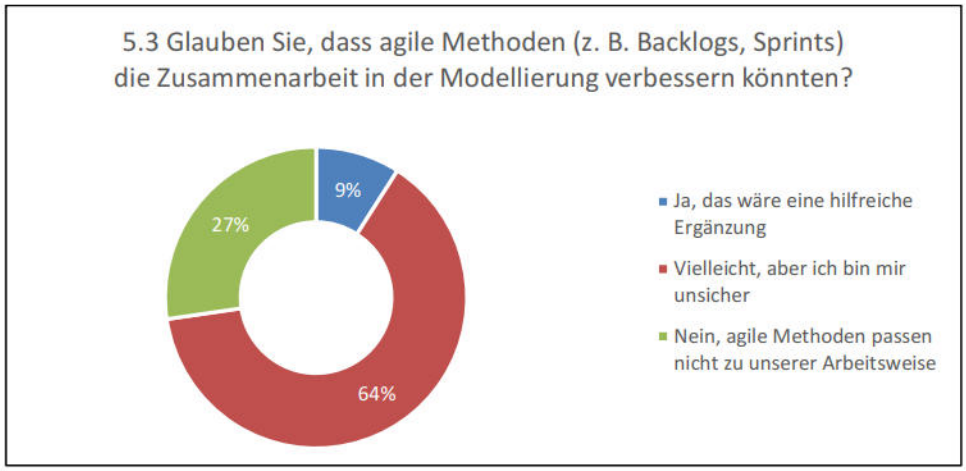


Bild 3-5 Anwendung agiler Prinzipien im MBSE-Kontext (Frage 5.3)

In der abschließenden offenen Frage wurde insbesondere die *Teamwork Cloud* von CSM als aktuell eingesetzte Lösung genannt – verbunden mit kritischen Anmerkungen zu deren Einschränkungen im Hinblick auf Versionsverwaltung und Zusammenarbeit.

Eine vollständige Übersicht aller Umfragefragen und ihrer Auswertung befindet sich im Anhang A3.

3.4 Rollenzuweisung anhand ARP4754B & ISO/IEC 15288

Weder ARP4754B noch ISO/IEC 15288 spezifizieren konkrete Rollen oder organisatorische Strukturen für die Durchführung der in den Normen definierten Prozesse. Vielmehr liegt es in der Verantwortung der Organisationen, geeignete Teams zu definieren, die in der Lage sind, die prozessbezogenen Anforderungen zu erfüllen und ein sicheres sowie normkonforme Flugsysteme zu entwickeln.

Vor diesem Hintergrund wird im Rahmen dieser Arbeit ein Vorschlag zur internen Rollenzuweisung innerhalb des betrachteten DLR-Teams unterbreitet. Die Grundlage dafür bildet die Teamumfrage, insbesondere die Auswertung von Frage 2.1 (siehe Bild 3-6), in der die vorhandenen Kompetenzen der Teammitglieder erhoben wurden.

Die Auswertung der Antworten auf Frage 2.1 zeigt, dass die Kompetenzen des Teams überwiegend im Bereich der Systementwicklung liegen, insbesondere in der Anforderungsanalyse, Systemarchitektur und Modellierung. Im Vergleich dazu sind Kenntnisse und Zuständigkeiten in der Implementierung und Validierung

weniger stark ausgeprägt. Dieses Kompetenzprofil deutet darauf hin, dass die Aktivitäten des Teams primär auf der linken Seite des ARP-V-Modells (vgl. Bild 2-3) verortet sind – also in den frühen Phasen der Systementwicklung, in denen die Systemanforderungen analysiert und das Design spezifiziert werden.

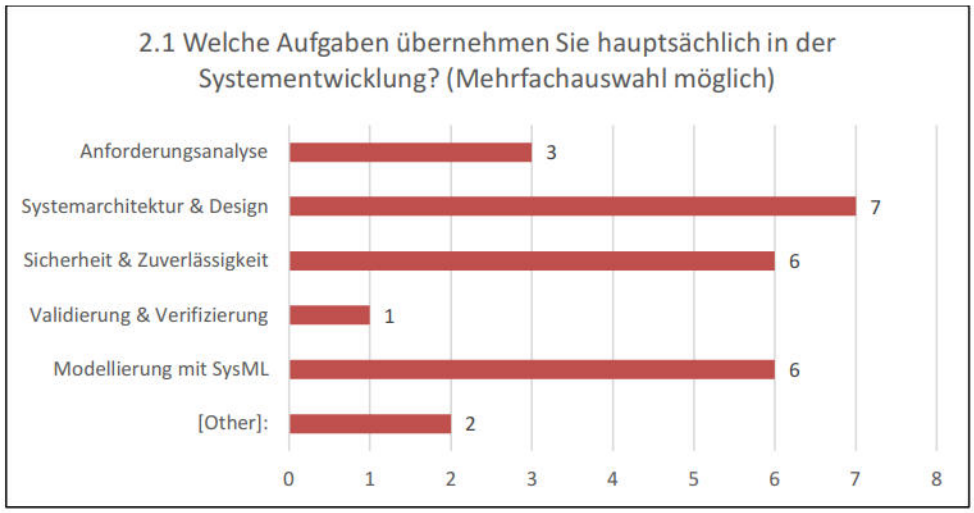


Bild 3-6 Übersicht von Teamkompetenzen (Frage 2.1)

Basierend auf den identifizierten Kompetenzfeldern und unter Berücksichtigung der charakteristischen Arbeitsergebnisse einzelner Prozesse konnten vier funktionale Teams abgeleitet werden:

- SE Management Team (nicht-technisch, Prozesskoordination)
- Anforderungsanalyse Team
- Systemarchitektur & Sicherheit Team
- Verifikation, Validierung (V&V) & Modellintegration Team

Zur strukturierten Zuweisung der relevanten Normprozesse zu den identifizierten Teams wird im nächsten Abschnitt eine RACI-Matrix herangezogen, um Verantwortlichkeiten nachvollziehbar abzubilden.

3.4.1 Methodik des Rollenzuweisungsprozesses

Für die systematische Zuordnung der in ARP4754B und ISO/IEC 15288 beschriebenen Prozesse zu den vier definierten Teams wird eine RACI-Matrix gemäß Gräßler et al., (2022) verwendet. Diese Methode erlaubt eine differenzierte Betrachtung der Rollenverteilung entlang folgender vier Verantwortlichkeitsstufen (Costello, 2012):

- R – *Responsible*: Die verantwortliche Person oder das Team führt die jeweilige Aufgabe durch.

- A – *Accountable*: Eine Person oder ein Team ist übergeordnet verantwortlich für die Zielerreichung und Qualität der Aufgabe. In kleinen Teams kann diese Rolle identisch mit der Rolle *Responsible* sein.
- C – *Consulted*: Personen oder Teams, die mit ihrer fachlichen Expertise beratend in den Prozess eingebunden werden.
- I – *Informed*: Beteiligte, die über den Fortschritt oder die Ergebnisse informiert werden müssen, aber nicht aktiv in die Durchführung eingebunden sind.

Die konkrete Zuordnung der Normprozesse zu den Teams erfolgt in Kapitel 3.4.2 und basiert auf den in der Umfrage ermittelten Teamfähigkeiten sowie den in den Normen geforderten Ergebnissen pro Prozess.

3.4.2 Ergebnisse der RACI-Matrix

Die in Tabelle 3-1 und Tabelle 3-2 dargestellten RACI-Matrizen bilden die erarbeiteten Rollenzuweisungen für die Normprozesse der ARP4754B bzw. der ISO/IEC 15288 ab. Ziel dieser systematischen Zuordnung ist es, die funktionale Verantwortungsverteilung innerhalb des DLR-Teams transparent darzustellen und aufzuzeigen, welche Rollen innerhalb des Teams welchen Prozessen der Standards zugewiesen werden können. Die Zuordnung unterstützt dabei, einen strukturierten Überblick über die Zuständigkeiten in der Systementwicklung zu gewinnen und liefert eine Orientierung für die praxisnahe Anwendung normativer Anforderungen im MBSE-Kontext.

Die RACI-Matrix zur ARP4754B (vgl. Tabelle 3-1) berücksichtigt sämtliche Prozesse der Norm, einschließlich der Planungs-, Entwicklungs- und Integralprozesse. Für jeden Hauptprozess wurde mindestens ein verantwortliches Team (*Responsible*) definiert, wodurch eine klare funktionale Rollenzuweisung innerhalb des betrachteten DLR-Teams ermöglicht wird. In den Planungsprozessen (1.1–1.3) übernimmt das SE-Management-Team die federführende Verantwortung, insbesondere hinsichtlich des Planungsumfangs und der Abstimmung mit externen Zertifizierungsstellen. Die übrigen Teams sind als *Consulted* eingestuft, da sie mit ihrer fachlichen Expertise zur Plausibilitätsprüfung und Absicherung der Planinhalte beitragen.

Der Prozess 3.1 (*Summary of Development Assurance Process Outputs*) wurde dem SE-Management-Team als *Responsible* zugewiesen, da es die Nachweisdokumente gemäß Normvorgaben zentral zusammenführt und verwaltet. Die technischen Teams liefern fachliche Inhalte zu (*Consulted*), tragen aber nicht die Gesamtverantwortung. Dieses zentralisierte Modell sieht die inhaltliche Zuarbeit durch die Fachbereiche vor, während das SE-Management Koordination und Endverantwortung übernimmt. In den operativen Entwicklungsprozessen ist das SE-Management

meist als *Informed* eingestuft, da sein Schwerpunkt auf der übergeordneten Steuerung liegt.

Für die Integralprozesse der ARP4754B wurden Zuständigkeiten in Abhängigkeit von den jeweiligen Inhalten und erwarteten Arbeitsergebnissen festgelegt. Dabei wurde berücksichtigt, dass diese Prozesse häufig eine enge fachliche Zusammenarbeit zwischen mehreren Teams erfordern. Um die Matrix übersichtlich zu halten, wurden nur explizit begründbare *Consulted*-Beziehungen eingetragen. Eine pauschale Beteiligung aller technischen Teams an sämtlichen Integralprozessen wurde bewusst vermieden, um die Zuordnung möglichst klar und zielgerichtet darzustellen.

Tabelle 3-1 RACI-Matrix zur Rollenzuweisung der ARP4754B-Prozesse im DLR-Projektkontext

Prozess #	Prozess (Engl.)	SE Management Team	Anforderungsanalyse Team	Systemarchitektur & Sicherheit Team	V&V & Modellintegration Team
1	**Development Planning**				
1.1	Development Assurance Planning Process	R	C	C	C
1.2	Development Assurance Plan	R	C	C	C
1.3	Certification Authority Coordination	R	C	C	C
2	**Aircraft/ System Development Process**				
2.1	Aircraft Function and Requirement Development	I	R	C	C
2.2	Development of Aircraft Architecture and Allocation of Aircraft Functions to Systems	I	C	R	C
2.3	Development of System Functions and Requirements	I	R	C	C
2.4	Development of System Architecture and Allocation of System Requirements to Items	I	C	R	C
2.5	Implementation	I	C	C	R
3	**Data & Documentation**				
3.1	Summary of Development Assurance Process Outputs	R	C	C	C
4	**Integral Processes**				
4.1	Safety Assessment	I	C	R	C
4.2	Development Assurance Level Assignment	A	C	R	I
4.3	Requirements Capture	I	R	C	I
4.4	Requirements Validation	I	C	C	R
4.5	Implementation Verification	I	I	C	R
4.6	Configuration Management	R	C	C	C
4.7	Process Assurance	R	I	I	I

Die zweite RACI-Matrix in Tabelle 3-2 basiert auf der internationalen Norm ISO/IEC 15288:2023 und bezieht sich ausschließlich auf die darin definierten *Technical Management Processes* (TMP) und *Technical Processes* (TP). Die Auswahl beschränkt sich bewusst auf diese beiden Prozessgruppen, da die umfassenden Lebenszyklusprozesse der Norm in ihrer Gesamtheit nicht vollständig durch die im Projektkontext definierten Teamrollen abgedeckt werden können. Insbesondere die Prozesse des Betriebs, der Wartung und der Entsorgung (4.12–4.14) liegen außerhalb des Wirkungskreises der betrachteten Forschungsgruppe und bleiben daher unberücksichtigt.

Tabelle 3-2 RACI-Matrix zur Rollenzuweisung der ISO/IEC 15288:2023 im DLR-Projektkontext

Prozess #	**Prozess (Engl.)**	**SE Management Team**	**Anforderungsanalyse Team**	**Systemarchitektur & Sicherheit Team**	**V&V & Modellintegration Team**
3	**Technical Management Processes**				
3.1	Project Planning Process	R	C	C	C
3.2	Project Assessment and Control Process	R	C	C	C
3.3	Decision Management Process	R	C	C	C
3.4	Risk Management Process	R	C	C	C
3.5	Configuration Management Process	R	C	C	C
3.6	Information Management Process	R	C	C	C
3.7	Measurement Process	R	C	C	C
3.8	Quality Assurance Process	R	C	C	C
4	**Technical Processes**				
4.1	Business or Mission Analysis Process	R	C	I	I
4.2	Stakeholder Needs and Requirements Definition Process	C	R	I	I
4.3	System Requirements Definition Process	I	R	C	C
4.4	System Architecture Definition Process	I	C	R	C
4.5	Design Definition Process	I	C	R	C
4.6	System Analysis Process	I	C	R	C
4.7	Implementation Process	I	C	C	R
4.8	Integration Process	I	C	C	R
4.9	Verification Process	I	C	C	R
4.10	Transition Process	I	C	C	R
4.11	Validation Process	I	C	C	R
4.12	Operation Process	-	-	-	-
4.13	Maintenance Process	-	-	-	-
4.14	Disposal Process	-	-	-	-

Die TMP (3.1–3.8) wurden vollständig dem SE-Management-Team als *Responsible* zugewiesen, da diese Prozesse im Wesentlichen Aufgaben wie Planung, Steuerung, Entscheidungsfindung, Qualitätssicherung sowie projektübergreifendes Konfigurations- und Informationsmanagement umfassen. Diese Tätigkeiten entsprechen dem Rollenverständnis des SE-Management-Teams als koordinierende Instanz innerhalb der Forschungsgruppe. Ergänzend wurden alle weiteren Teams als *Consulted* eingetragen, um ihre fachliche Einbindung in die Prozessgestaltung und -reflexion zu berücksichtigen. Diese beratende Beteiligung dient der Sicherstellung von Umsetzbarkeit, Transparenz und Plausibilität der Managemententscheidungen.

Für die TP (4.1–4.11) wurde jeweils ein fachlich zuständiges Team als *Responsible* festgelegt, basierend auf den in der Teamumfrage identifizierten Kompetenzen sowie den typischen Arbeitsergebnissen der jeweiligen Prozesse. In Übereinstimmung mit den Grundprinzipien des MBSE wurde zudem berücksichtigt, dass viele dieser Prozesse eine interdisziplinäre Zusammenarbeit erfordern. Dementsprechend wurden benachbarte Teams als *Consulted* eingetragen, wenn eine inhaltliche Rückkopplung oder Mitwirkung zu erwarten ist, etwa bei der Erstellung und Validierung technischer Anforderungen oder bei der architektonischen Modellintegration. Teams, die lediglich über Zwischenergebnisse oder Statusinformationen informiert werden müssen, erhielten die Rolle *Informed*. Eine übergreifende Beteiligung aller Teams an allen Prozessen wurde dabei vermieden, um die Matrix übersichtlich und aussagekräftig zu halten.

3.5 Herausforderungen der Zusammenarbeit

Die in Kapitel 2.5.1 identifizierten Herausforderungen der modellbasierten Kollaboration wurden auf Basis einer Literaturanalyse strukturiert und in acht übergeordnete Kategorien eingeordnet (vgl. Tabelle 2-4). Im Rahmen der Teamumfrage (Frage 4.1, vgl. Bild 3-3) konnten spezifische Hürden benannt werden, die innerhalb der betrachteten Forschungsgruppe im Umgang mit modellbasierten Arbeitsweisen auftreten. Diese wurden anschließend den bestehenden Kategorien zugeordnet, um ein konsistentes Bild zwischen Theorie und Praxis zu erhalten. Die identifizierten teaminternen Herausforderungen und deren systematische Zuordnung zu den literaturbasierten Kategorien sind in Tabelle 3-3 dargestellt.

Die in Kapitel 3.4 entwickelte Rollenzuweisung nach ARP4754B und ISO/IEC 15288 stellt einen ersten Ansatz zur Bewältigung der organisatorischen Herausforderung „fehlende klare Prozesse“ dar. Durch den Einsatz der RACI-Matrix konnte eine transparente Orchestrierung der Teamrollen erfolgen, wodurch Verantwortlichkeiten und Schnittstellen klarer definiert wurden.

Auch die im Rahmen der Teamumfrage genannten Herausforderungen im Bereich Änderungsmanagement und Informationsmanagement werden im weiteren Verlauf

adressiert. In Kapitel 4.4 wird ein Git-basierter Kollaborationsprozess spezifiziert, der es erlaubt, Modellversionen nachvollziehbar zu verwalten, Änderungen transparent zu dokumentieren und verteilte Zusammenarbeit effizient zu unterstützen. Auf diese Weise können zentrale Defizite der bisherigen Arbeitsweise gezielt behoben werden.

Tabelle 3-3 Zuordnung der teamintern identifizierten Herausforderungen zu den literaturbasierten Kategorien (vgl. Kapitel 2.5.1)

Teaminterne Herausforderung	**Zuordnung zur Kategorie (vgl. Kap. 2.5.1)**
Fehlende klare Prozesse	Orchestrierung
Inkonsistente Änderungen	Änderungsmanagement
Veraltete Modelle	Informationsmanagement
Fehlende geeignete Tools	Plattformtechnische Grenzen
Zeitlicher Mehraufwand im Forschungsbereich	Organisation & Kultur

Ein weiteres Hindernis stellt die eingeschränkte Werkzeugunterstützung dar. Um den Auswirkungen der plattformtechnischen Grenzen zu begegnen, wird ein prototypisches Anwendungsszenario entworfen, in dem ein SysML-v2-Modell, das in CSM erstellt wurde, zusätzlich im *Jupyter Notebook* visualisiert und bearbeitet werden kann. Diese Integration ermöglicht es, Modelle unabhängig vom proprietären Modellierungswerkzeug zu betrachten und zu analysieren.

Die zuletzt genannte Herausforderung – der zeitliche Mehraufwand im Forschungsbereich – verweist auf strukturelle Gegebenheiten innerhalb der Organisation. Diese sind durch projektbezogene Dynamiken, wechselnde Zuständigkeiten und begrenzte Modellierungskapazitäten geprägt. Solche Rahmenbedingungen lassen sich im Kontext dieser Arbeit nicht grundlegend verändern, können aber bei der Gestaltung von Prozessen und Werkzeugen insofern berücksichtigt werden, dass möglichst ressourcenschonende und intuitive Lösungen entwickelt werden.

Insgesamt zeigt sich, dass ein erheblicher Teil der identifizierten Herausforderungen durch gezielte methodische und technische Gestaltung adressiert werden kann. Die Erkenntnisse aus diesem Kapitel bilden somit eine wichtige Grundlage für den in Kapitel 4 folgenden *Design Cycle* der DSR-Methodologie.

4 Entwicklung eines Git-basierten Kollaborationsprozesses

Basierend auf den in Kapitel 3 identifizierten Anforderungen und Herausforderungen wird im vorliegenden Kapitel ein Git-basierter Kollaborationsprozess für die modellbasierte Systementwicklung mit SysML v2 entwickelt. Dieses Kapitel markiert den Übergang in den Gestaltungszyklus (*Design Cycle*) des DSR-Ansatzes und stellt somit den zentralen Konstruktionsanteil dieser Arbeit dar. Ziel ist es, einen nachvollziehbaren, reproduzierbaren und toolgestützten Arbeitsablauf zu entwerfen, der die gleichzeitige Bearbeitung von Systemmodellen durch verteilte Teammitglieder ermöglicht und dabei Aspekte wie Nachvollziehbarkeit, Konsistenz und Konfliktvermeidung adressiert.

Die Entwicklung des Prozesses erfolgt exemplarisch anhand eines UAV-Systems unter Verwendung des CSM mit SysML v2-Plugin sowie GitLab als Plattform zur verteilten Versionsverwaltung. Die Ausgestaltung orientiert sich an bewährten Vorgehensweisen aus der Softwareentwicklung, die gezielt auf die Anforderungen und Besonderheiten der modellbasierten Systementwicklung (MBSE) übertragen und angepasst werden.

Das folgende Kapitel gliedert sich in vier Abschnitte: Zunächst wird das methodische Vorgehen zur Prozessentwicklung erläutert (**Kapitel 4.1**). Anschließend werden die erforderlichen Konfigurationen für die Modellierung in CSM (**Kapitel 4.2**) und die Einrichtung der Versionsverwaltung mit GitLab (**Kapitel 4.3**) beschrieben. Den Abschluss bildet die Integration beider Komponenten in einen durchgängigen Kollaborationsprozess (**Kapitel 4.4**).

4.1 Methodisches Vorgehen zur Prozessentwicklung

Ziel dieses Kapitels ist die Entwicklung eines Git-basierten Prozesses zur verteilten modellbasierten Systementwicklung, der auf die Anforderungen des Projektkontexts zugeschnitten ist. Zur methodischen Absicherung wurde ein durchgängiges Beispielsystem verwendet, das den praktischen Einsatz und die Werkzeugintegration veranschaulicht.

Als Beispielsystem dient ein unbemanntes Luftfahrtsystem *„UAV Civil Drone“*, das ursprünglich in SysML v1 modelliert und vom DLR bereitgestellt wurde. Im Rahmen dieser Arbeit wurde dieses System manuell in SysML v2 überführt und entsprechend der aktuellen Sprachspezifikation strukturiert. Die Modellierung erfolgte in der Software CSM unter Verwendung des SysML v2-Plugins.

Zur zentralen Verwaltung des Modells wurde ein GitLab-*Repository* eingerichtet, das sowohl der Versionskontrolle als auch der kollaborativen Entwicklung dient.

Die Modellversionen werden als *.sysml*-Dateien exportiert und über Git synchronisiert.

Zur textuellen Bearbeitung und Anzeige der *.sysml*-Dateien außerhalb von CSM wird *Jupyter Notebook* (JN) mit dem offiziellen SysML-Kernel der OMG verwendet. Die Installation dieses Kernels erfolgte gemäß den im offiziellen GitHub-*Repository* dokumentierten Anleitungen (OMG Systems Modeling Community, n.d.).

Die Anzeige und Analyse des Modells im JN erfolgt über spezifische Kommandos des SysML-Kernels. Eine exemplarische Übersicht dieser Kernel-Kommandos ist im Anhang A4 dokumentiert. Damit kann das Modell unabhängig von CSM in einer offenen, textbasierten Umgebung analysiert, bearbeitet und visualisiert werden. Eine detaillierte Konfiguration von JN wird in dieser Arbeit nicht behandelt, da das Werkzeug ausschließlich in der Evaluationsphase (Kapitel 5) unterstützend zum Einsatz kommt.

Bild 4-1 veranschaulicht die eingesetzte Werkzeuglandschaft zur modellbasierten Systementwicklung mit SysML v2.

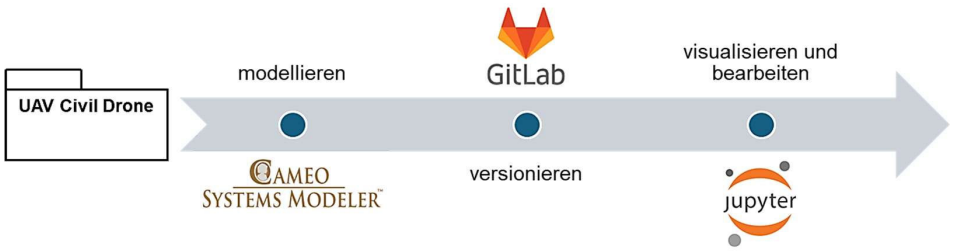


Bild 4-1 Werkzeuglandschaft im Git-basierten MBSE-Prozess (eigene Darstellung)

Die Entwicklung des Git-basierten Kollaborationsprozesses erfolgt entlang eines strukturierten methodischen Vorgehens, das sowohl die Zieldefinition als auch die Auswahl und Konfiguration geeigneter Werkzeuge umfasst. Die Konfigurationsrichtlinien betreffen ausschließlich die in der Definitions- und Implementierungsphase genutzten Hauptwerkzeuge: CSM und GitLab.

Das in Bild 4-2 dargestellte Aktionsdiagramm visualisiert die wesentlichen Schritte der Definitions- und Werkzeugkonfigurationsphase im Rahmen der Prozessentwicklung. Aufgeteilt in zwei parallele Handlungsstränge werden zunächst die Konfigurationsrichtlinien für das GitLab-*Repository* sowie für das Systemmodell in CSM festgelegt. Daraufhin erfolgt jeweils die konkrete Umsetzung: Das *Repository* wird erstellt, mit Zugriffsbeschränkungen versehen und für die Zusammenarbeit vorbereitet. Parallel dazu wird das Systemmodell in CSM neu aufgebaut, gemäß den Richtlinien strukturiert und anschließend als *.sysml*-Datei exportiert. Abschließend wird die exportierte Datei in das GitLab-*Repository* überführt und mit den beteiligten Teammitgliedern geteilt. Das Diagramm verdeutlicht somit den engen

Zusammenhang zwischen Modellierungs- und Versionsverwaltungswerkzeugen und legt die Grundlage für den in Abschnitt 4.4 beschriebenen Kollaborationsprozess.

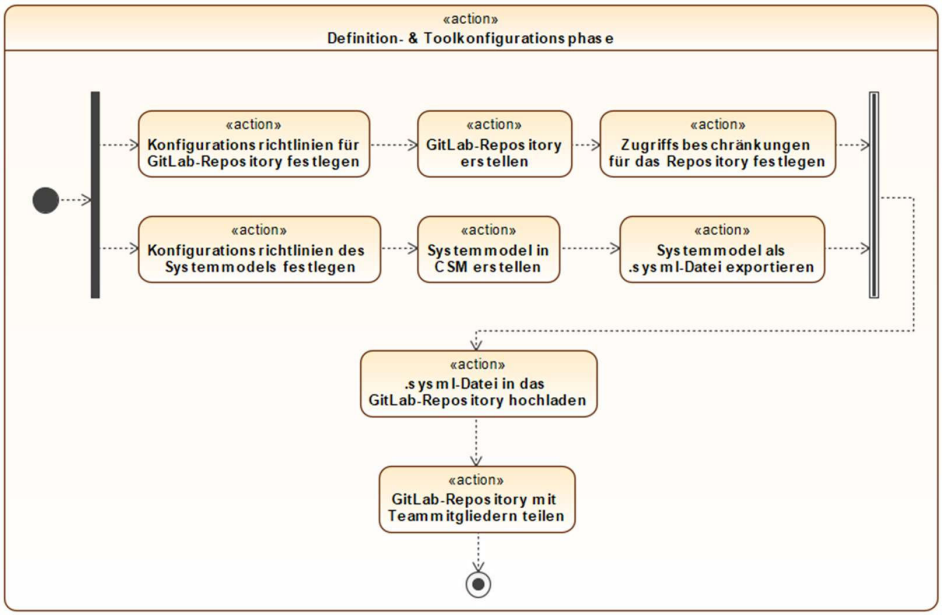


Bild 4-2 SysML v2 Aktionsdiagramm zur Darstellung der Definition- und Werkzeugkonfigurationsphase (eigene Darstellung in CSM)

Aufbauend auf dem in Bild 4-2 dargestellten Vorgehen konzentrieren sich die folgenden Abschnitte 4.2 und 4.3 auf die Konfigurationsrichtlinien für das Systemmodell in CSM sowie für das GitLab-*Repository*.

4.2 Konfigurationsrichtlinien für CSM mit SysML v2

Zur modellbasierten Entwicklung des Beispielsystems UAV wurde der CSM mit dem SysML v2-Plugin verwendet. Die Konfiguration des Systemmodells orientiert sich an den grundlegenden Sprachfähigkeiten von SysML v2, wie sie in Bild 2-7 dargestellt sind. Für die Umsetzung im Rahmen dieser Arbeit wurden die Fähigkeiten **Anforderungen**, **Verhalten**, **Struktur** sowie ***View & Viewpoint*** berücksichtigt. Die Fähigkeiten **Analyse** und **Verifikation** bleiben im aktuellen Modellierungsumfang unberücksichtigt, da sie für die Definitions- und Implementierungsphase nicht im Fokus stehen.

Die Umsetzung dieser Fähigkeiten erfolgt in CSM durch das Anlegen separater, sogenannter *namespaces*, die den jeweiligen Modellierungsaspekten zugeordnet sind. Bild 4-3 zeigt die initial konfigurierte Paketstruktur des UAV-Systems. Jedes Paket repräsentiert dabei eine sprachspezifische Fähigkeit gemäß SysML v2 und bildet eine klare Trennung der Modellinhalte. Diese Struktur unterstützt sowohl die

logische Ordnung innerhalb des Modells als auch die Wiederverwendbarkeit und Erweiterbarkeit des Systems.

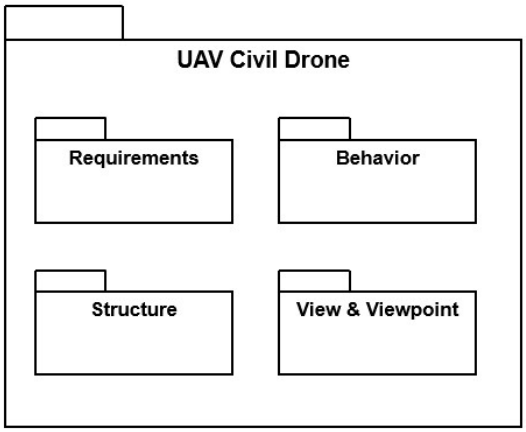


Bild 4-3 UAV-Paketstruktur gemäß SysML v2 Sprachfähigkeiten (eigene Darstellung)

Ein detaillierter Einblick in das strukturierte Systemmodell wird in Bild 4-4 gegeben. Im *namespace* **Anforderungen** sind Kundenanforderungen, Anforderungen auf *System-of-Systems*-Ebene (SoS) sowie Anforderungen für das *System-of-Interest* (SoI) enthalten. Der *namespace* **Verhalten** enthält modellierte *Use Cases* zur Beschreibung funktionaler Abläufe. Im *namespace* **Struktur** wurde eine hierarchische Zerlegung des Systems realisiert – unter anderem mit einer Part-Dekomposition des SoS und SoI.

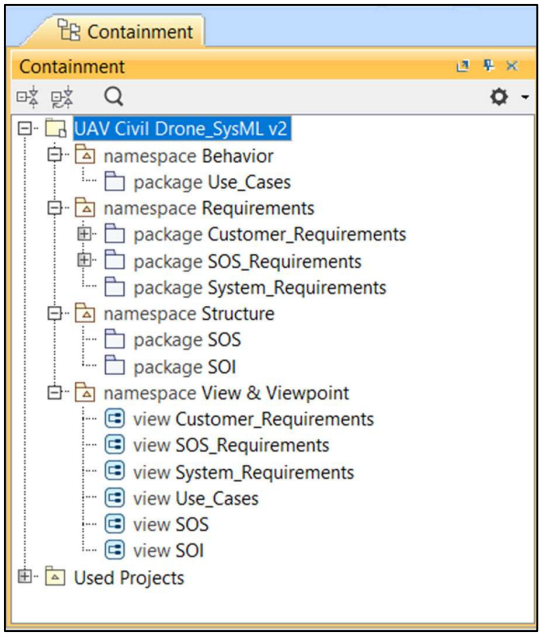


Bild 4-4 Modellstruktur in CSM (eigener Screenshot, CSM)

Besonderheiten ergeben sich beim Umgang mit der Fähigkeit ***View & Viewpoint***. In CSM ist diese Funktionalität direkt integriert und unterstützt die visuelle Aufbereitung von Modellinhalten entsprechend der Bedürfnisse unterschiedlicher Stakeholder. In der Modellierungsumgebung existiert lediglich nur ein Diagrammtyp, das sogenannte *View*-Diagramm, das unterschiedliche Elemente aus dem Modell in einem gemeinsamen Kontext darstellen kann.

Beim Export in die textuelle SysML v2-Notation – beispielsweise zur Weiterverarbeitung oder Analyse in JN – wird jedoch nicht zwischen *View*-Diagramm und *View*-Element unterschieden. Beide werden dort als reguläre *View*-Elemente dargestellt, was zu Darstellungsabweichungen führen kann. Diese Diskrepanz ist exemplarisch in Bild 4-5 erkennbar: Während CSM eine visuell konsolidierte Darstellung bietet, erscheinen im textuellen Notation zwei separate Elementdefinitionen für die *„Views“*.

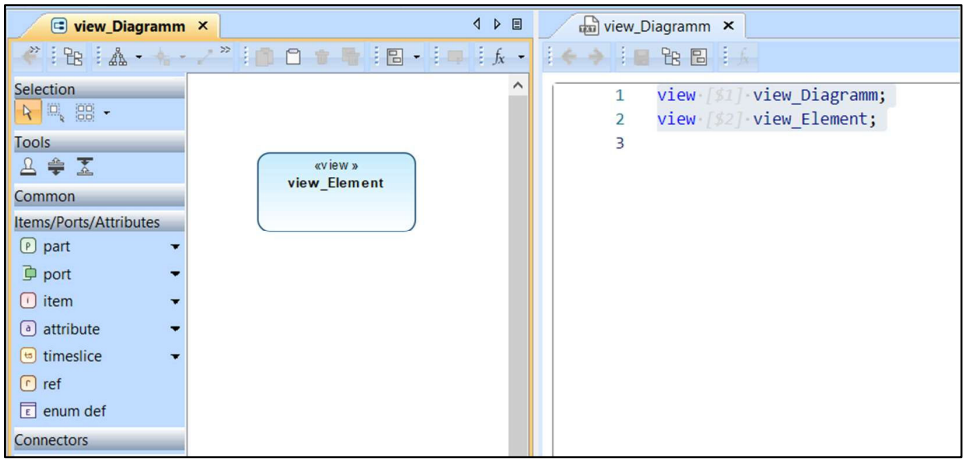


Bild 4-5 „View“-Diagramm und „View“-Element in SysML v2-textueller Notation (eigener Screenshot, CSM)

Zum Austausch des Modells in textueller Form stellt CSM eine Export- und Importfunktion für die SysML v2-Notation bereit. Der Export erfolgt über das Menü „Datei“ → *„Export to“* → *„Export SysML v2 textual notation“* (vgl. Bild 4-6). Um ein bestehendes Modell im *.sysml*-Format zu importieren, wird per Rechtsklick auf das Systemmodell im *Containment*-Baum die Option *„Import SysML v2 from textual notation“* ausgewählt (vgl. Bild 4-7).

Diese Funktionen ermöglichen den Export und Import von *.sysml*-Dateien, also des Modells in textueller Notation, was im Fokus dieses Git-basierten Ansatzes steht. Dabei enthält jeder *namespace* die textuelle Repräsentation der darin beschriebenen Pakete bzw. Modellelemente, welche auch einzeln exportiert werden können. Die Funktionsweise und Zuverlässigkeit dieser Import- und Exportmechanismen wird in Kapitel 5 im Rahmen der Testszenarien eingehender untersucht.

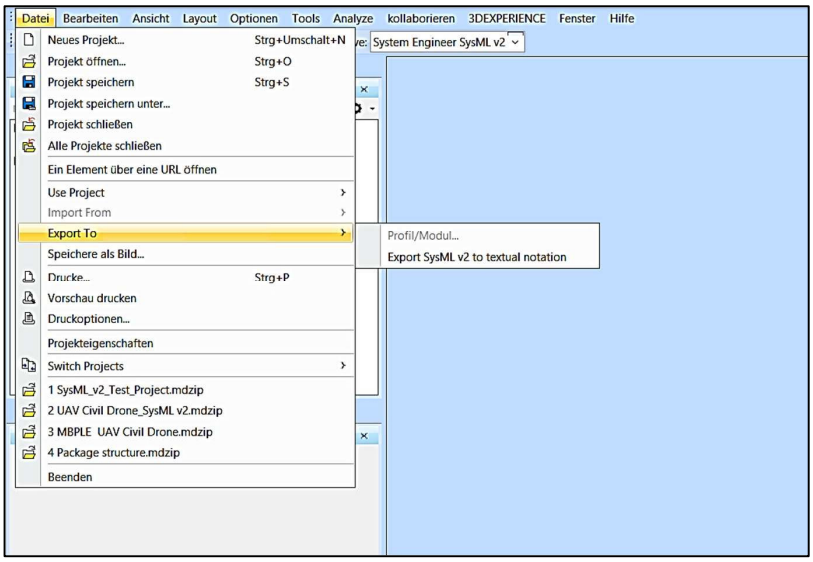


Bild 4-6 Export einer SysML v2-Datei in CSM (eigener Screenshot, CSM)

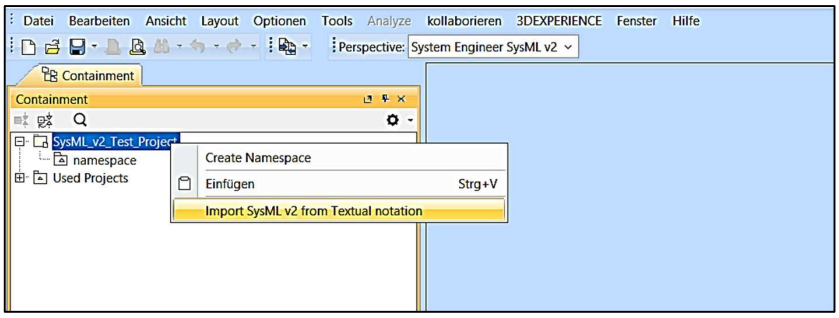


Bild 4-7 Import einer SysML v2-Datei in CSM (eigener Screenshot, CSM)

4.3 Konfigurationsrichtlinien für Git in GitLab

In diesem Abschnitt wird die Einrichtung eines Git-*Repository* in GitLab erläutert. In GitLab werden *Repositories* als „Projekte“ bezeichnet. Grundsätzlich gibt es zwei Möglichkeiten, ein *Repository* mit Git zu initialisieren:

1. Initialisierung lokal mit *´git init´* und anschließendes Hochladen in das Git-Lab-*Repository*, oder
2. Erstellung eines neuen Projekts direkt in GitLab und anschließendes Klonen in ein lokales *Repository*.

Für diese Arbeit wurde der zweite Weg gewählt, da er u.a. für Teamkollaboration und spätere Integration mit GitLab CI/CD und Zugriffsverwaltung besser geeignet ist.

Die Erstellung eines neuen Projekts erfolgt über die Schaltfläche „*Create blank project*“, wie in Bild 4-8 dargestellt. Für das Projekt wurde der Name *„UAV Civil Drone“* gewählt. Der „*Visibility Level“* wurde auf Private gesetzt, um unautorisierten Zugriff zu vermeiden. Zusätzlich wurde unter „*Project Configuration*“ eine *README*-Datei hinzugefügt. Diese Datei ist optional, wird hier jedoch eingefügt, um die erfolgreiche Verbindung beim späteren „Klonen“ des Projekts auf das lokale *Repository* sichtbar überprüfen zu können.

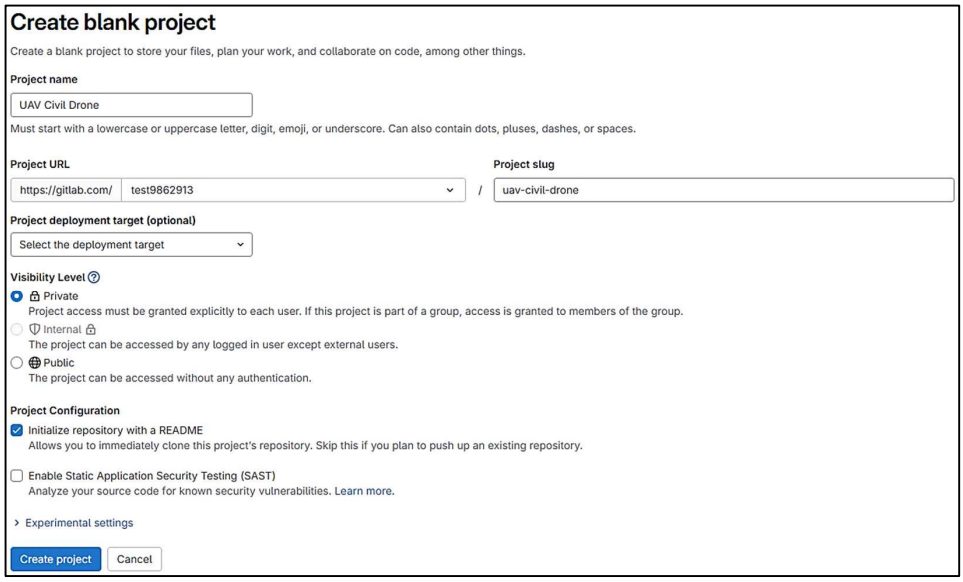


Bild 4-8 Erstellung eines neuen Projekts in GitLab über die Option „Create blank project“ (eigener Screenshot, GitLab)

Bevor das Projekt geklont wird, ist ein Überblick über die wichtigsten Git-Befehle hilfreich, die im Rahmen des später vorgestellten Git-basierten Kollaborationsprozesses (Kapitel 4.4) verwendet werden. Tabelle 4-1 enthält die zentralen Git-Kommandos, die regelmäßig benötigt werden. Eine ausführlichere Befehlsübersicht ist im Anhang A2 zu finden.

Die effektive Nutzung von Git als Versionskontrollsystem setzt ein grundlegendes Verständnis der wichtigsten Kommandos voraus – insbesondere in modellbasierten Entwicklungsprozessen, bei denen textuelle Inhalte wie SysML v2-Dateien versioniert werden. Die folgenden Git-Befehle dienen als Basis für den Git-basierten Kollaborationsprozess und kommen sowohl bei der lokalen Arbeit mit *Branches* als auch bei der Synchronisation mit dem entfernten *Repository* regelmäßig zum Einsatz. Da Git ursprünglich für Quellcode entwickelt wurde, ist ein disziplinierter und kontextbezogener Umgang mit den Befehlen erforderlich, um Modellversionen konsistent und nachvollziehbar zu verwalten. Dies gilt insbesondere beim parallelen Arbeiten im Team, wo Klarheit über den Ablauf von *Commit*-, *Pull*- und *Merge*-Vorgängen entscheidend ist.

Tabelle 4-1 Übersicht der wichtigsten Git-Befehle

Befehl	**Beschreibung**
´git init´	Initialisiert ein neues Git-*Repository* im aktuellen Verzeichnis
´git clone <URL>´	Klonen eines Remote-*Repositories* in ein lokales Verzeichnis
´git branch´	Listet alle lokalen *Branches* auf
´git checkout -b <branchname>´	Erstellen und Wechseln in einen neuen *Branch*
´git add .´	Stellt alle Änderungen im aktuellen Verzeichnis bereit
´git commit -m "<Nachricht>"´	Speichern der Änderungen mit einer *Commit*-Nachricht
´git push´	Überträgt lokale *Commits* zum entfernten *Repository*
´git push -u origin <branchname>´	Hochladen des *Branches* in das entfernte *Repository*
´git pull´	Abrufen und Zusammenführen von Änderungen aus dem entfernte Repository
´git merge <branchname>´	Zusammenführen eines *Branches* in den aktuellen *Branch*
´git status´	Zeigt den Status der Dateien (*staged*, *unstaged*, *untracked*)

Zum Klonen des GitLab-Projekts in ein lokales *Repository* wurde auf dem lokalen *Repository* zunächst ein leerer Ordner erstellt – ebenfalls mit dem Namen *„UAV Civil Drone“*. Um Git-Befehle in diesem Verzeichnis auszuführen, wurde im Windows-Datei-Explorer die Adresszeile genutzt. Dort kann durch Eingabe von „cmd“ (*Command Prompt*) direkt ein Eingabeaufforderungsfenster im entsprechenden Verzeichnispfad geöffnet werden. Dies ist in Bild 4-9 zu sehen.

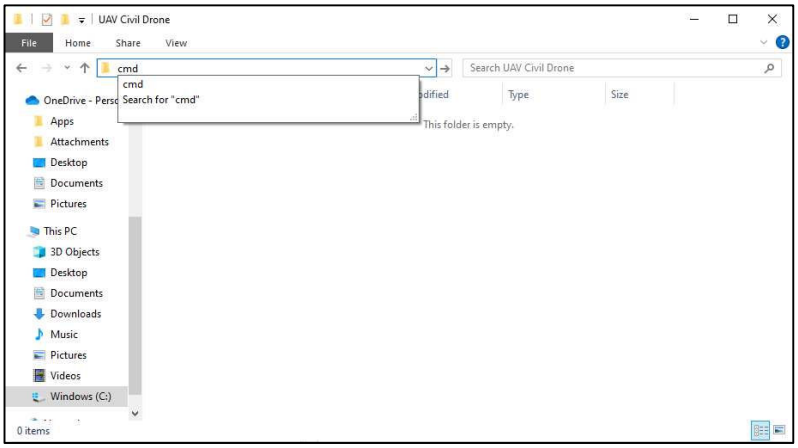


Bild 4-9 Öffnen der Eingabeaufforderung direkt aus dem lokalen Verzeichnispfad über die Adresszeile (eigener Screenshot, Windows Explorer)

Anschließend wurde das *Repository* mithilfe des Befehls *´git clone <Repository-URL>´* in das lokale *Repository* geklont. Bild 4-10 zeigt den *Cloning*-Vorgang im cmd bzw. Git CLI (*Command Line Interface*).

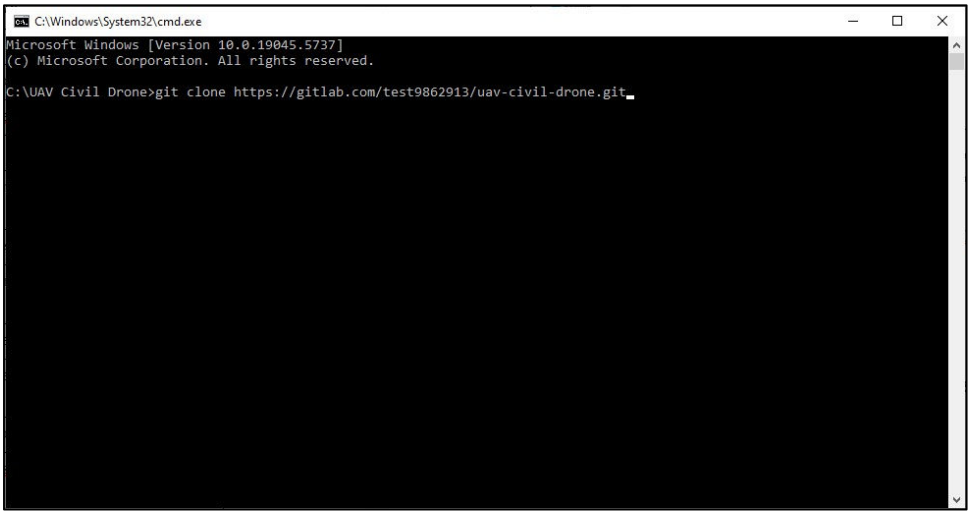


Bild 4-10 Klonen des GitLab-Repository über die Eingabeaufforderung mit ´git clone´ (eigener Screenshot, Git CLI)

Nach erfolgreichem Klonen ist die zuvor in GitLab angelegte *README*-Datei nun auch im lokalen *Repository* sichtbar, wie in Bild 4-11 dargestellt.

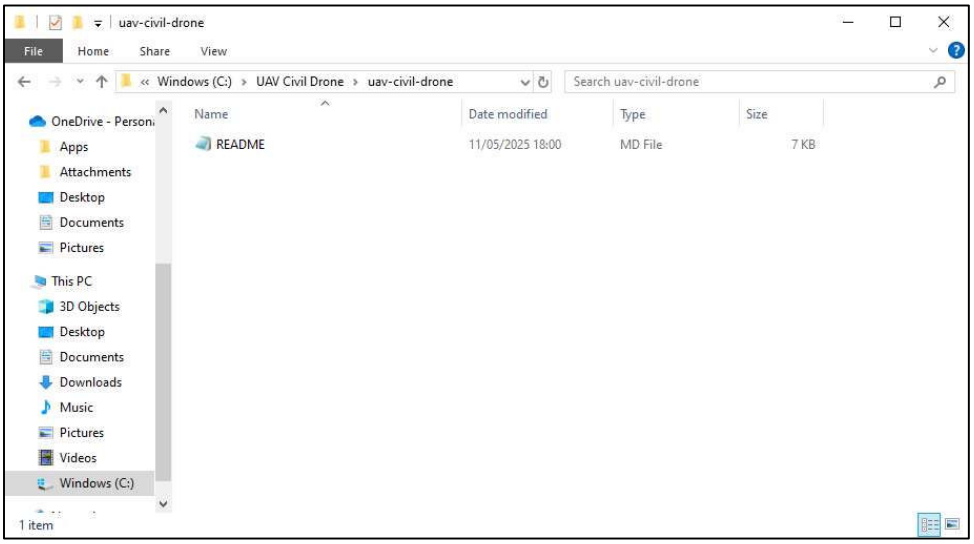


Bild 4-11 Anzeige der geklonten README-Datei im lokalen Repository (eigener Screenshot, Windows Explorer)

Nach dem erfolgreichen Verbinden des lokalen *Repository* mit dem entfernten GitLab-*Repository* besteht nun die Möglichkeit, Dateien lokal hinzuzufügen und anschließend in das entfernte *Repository* zu übertragen. Was bislang fehlt, ist eine

Definition eines Git-basierten Kollaborationsprozesses (Workflow), der als einheitliche Vorgehensweise von allen Beteiligten befolgt werden soll, um die Zusammenarbeit effizient und konsistent zu gestalten. Dieser Prozess wird im folgenden Kapitel 4.4 eingeführt.

4.4 Definition des Git-basierten Kollaborationsprozesses

Ein Git-Workflow definiert die Struktur und die Regeln für die Zusammenarbeit innerhalb eines Entwicklungsteams auf Basis von Git. Als verteiltes Versionskontrollsystem ermöglicht Git eine effiziente Nachverfolgbarkeit von Änderungen, parallele Entwicklungsstränge sowie eine verlässliche Verwaltung unterschiedlicher Softwareversionen (Cui, 2024). Dies schafft die Grundlage für kollaborative Software- und Systementwicklung in modernen, oft standortverteilten Projektteams.

Im Kontext dieser Arbeit bezieht sich der Git-Workflow nicht auf klassischen Quellcode, sondern auf modellbasierte Entwicklungsartefakte, die in CSM mit SysML v2 erstellt werden. Ziel ist es, Änderungen an Modellinhalten – etwa Paketen, Anforderungen oder Strukturelementen – versionierbar, nachvollziehbar und kollaborativ zu verwalten.

Zu den bekanntesten Workflows gehören Git Flow, GitHub Flow und GitLab Flow, die jeweils unterschiedliche Anforderungen und Projektgrößen adressieren. Neben der Organisation der *Branch*-Strukturen stehen auch moderne Praktiken der Softwarebereitstellung wie CI/CD sowie der kontrollierte Wechsel zwischen verschiedenen Systemzuständen in engem Zusammenhang mit diesen Workflows (Gowda, 2022).

Ein *Branch*-Modell ist dabei eine essenzielle Strategie, um die Zusammenarbeit im Team zu strukturieren. Es legt fest, wie neue Funktionen, Fehlerkorrekturen und Veröffentlichungsversionen organisiert werden, reduziert Integrationskonflikte, verbessert die Modellqualität und sorgt für eine stabile, auslieferbare Hauptversion des Modells. Ohne ein solches Modell kann Git zwar genutzt werden, verliert aber schnell an Übersichtlichkeit und Kontrolle – insbesondere bei mehreren beteiligten Personen oder komplexeren Projekten.

Git Flow eignet sich besonders für Projekte mit stabilen *Release*-Zyklen. Es nutzt zwei Hauptzweige (*main* und *develop*) und organisiert Entwicklung in *Feature*-, *Release*- und Fehlerbehebungszweige. Dieses Modell ist strukturiert, jedoch relativ komplex und für kontinuierliche Auslieferungen weniger geeignet.

GitHub Flow verfolgt hingegen einen sehr schlanken Ansatz. Für jede Änderung wird ein neuer *Branch* aus dem Hauptzweig (*main*) erstellt, darin gearbeitet und anschließend ein *Merge Request* (MR) geöffnet. Nach erfolgreicher Überprüfung wird der Zweig direkt in den Hauptzweig integriert. Diese Methode ist besonders für kontinuierliche Integration und kurze Iterationen geeignet.

GitLab Flow kombiniert Elemente aus Git Flow und GitHub Flow, um verschiedene Entwicklungs- und Bereitstellungsumgebungen zu unterstützen.

Die Wahl eines geeigneten Workflows hängt von mehreren Faktoren ab. Die meist relevanten sind:

- **Teamgröße:** Kleine Teams profitieren oft von einfacheren Modellen wie GitHub Flow.
- **Projektkomplexität:** Für umfangreiche Systeme mit mehreren parallelen Entwicklungen kann Git Flow sinnvoll sein.
- ***Release*-Zyklen:** Bei kontinuierlicher Bereitstellung empfiehlt sich GitHub Flow.
- **Automatisierungsgrad:** CI/CD-Verfahren harmonieren besonders gut mit schlanken Workflows.

Für die vorliegende Arbeit wird der GitHub Flow als Kollaborationsprozess gewählt. Ausschlaggebend dafür ist seine Einfachheit, die auf kurzen *„Feature"-Branches* und einer klaren Struktur basiert. Diese erlaubt eine schnelle Integration von Änderungen, reduziert Konfliktpotenziale und unterstützt einen kontinuierlichen Entwicklungsfluss. Obwohl GitHub Flow ursprünglich für GitHub entwickelt wurde, lässt er sich problemlos auch in GitLab umsetzen. Funktionen wie MR, *Pipeline*-Integration und Rechteverwaltung sind dort ebenfalls gegeben. Bild 4-12 zeigt schematisch den Ablauf des GitHub Flow.

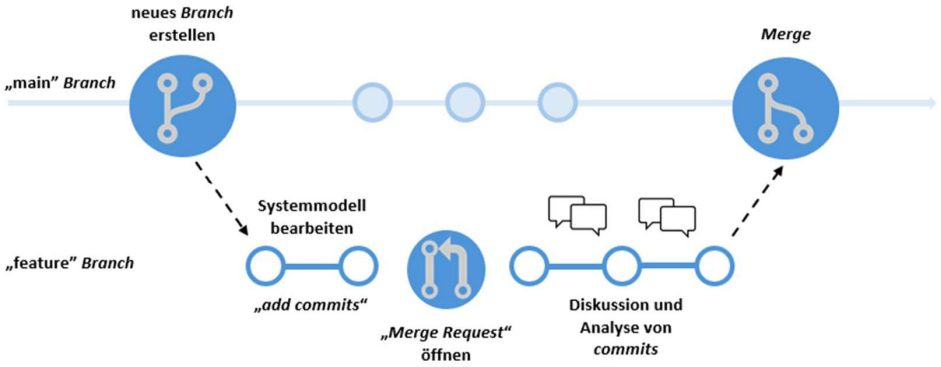


Bild 4-12 GitHub Flow – vereinfachter kollaborativer Entwicklungsprozess (eigene Darstellung)

Die Umsetzung des Workflows basiert auf dem Export von modellierten Inhalten im SysML v2-Textformat aus CSM. Die so erzeugten *.sysml*-Dateien können als Versionseinheiten über GitLab verwaltet und im Rahmen von *Branches* und MR ausgetauscht werden.

Nach der Einführung in das Konzept des GitHub Flow wird im Folgenden erläutert, wie dieser Workflow konkret im Rahmen dieser Arbeit umgesetzt wurde. Dabei wurden insbesondere die im Kapitel 3 dargestellten Rahmenbedingungen berücksichtigt – darunter Teamgröße, Projektkomplexität, *Release*-Zyklen sowie die identifizierten Herausforderungen. Vor diesem Hintergrund erwies sich GitHub Flow als geeigneter Ansatz für die modellbasierte Zusammenarbeit. Die Kollaborationsstrategie orientiert sich somit an den realen Gegebenheiten der Modellierungsumgebung mit CSM in Verbindung mit GitLab.

Im Sinne eines schlanken und zugleich qualitätsgesicherten Prozesses gelten für den Umgang mit *Branches* und MR folgende teamübergreifende Regeln:

1. ***Branch*-Erstellung:** Jede modellierende Person erstellt für eine geplante Änderung einen neuen *Feature-Branch* ausgehend vom Hauptentwicklungszweig (*main-Branch*). Der *Branch*-Name folgt dem Schema *feature/<kurze-Beschreibung>* – Beispiel: *feature/add-new-uc*, um eine klare Zuordnung und spätere Nachverfolgung zu ermöglichen.

2. ***Merge Requests* (MR):** Nach Abschluss und erfolgreicher Prüfung der Änderung in der lokalen Modellierungsumgebung wird ein MR erstellt. Dieser enthält eine kurze, präzise Beschreibung der Änderung sowie den Bezug zu den betroffenen Systemfunktionen oder Anforderungen. Der MR dient der Einleitung des Review-Prozesses.

3. **Review-Prozess:** Jeder MR wird von mindestens einer fachlich geeigneten Person aus einem anderen Team überprüft. Dabei wird insbesondere auf Modellkonsistenz, Einhaltung der Modellierungsrichtlinien und Konfliktfreiheit geachtet. Das SE-Management-Team nimmt nicht aktiv an der Modellierung teil, beteiligt sich jedoch am Review-Prozess zur Koordination, Validierung und finalen Entscheidung über die Integration in den Hauptzweig (*main*).

Durch die systematische Nutzung von *Branches*, *Commit*-Historien und MR kann die Entwicklungshistorie der Modellartefakte revisionssicher dokumentiert werden. Dies ermöglicht eine transparente Nachverfolgung aller Änderungen sowie die Rückverfolgbarkeit zu Anforderungen und Systemfunktionen.

Diese Regeln ermöglichen eine koordinierte Zusammenarbeit zwischen den technischen Teams, wobei gleichzeitig Transparenz und Modellqualität sichergestellt werden. Die *Branches* dienen der parallelen Entwicklung, der Review-Prozess verhindert unkontrollierte Änderungen, und die strukturierte Namensgebung unterstützt sowohl Nachvollziehbarkeit als auch Automatisierung.

Eine schrittweise Anleitung zur praktischen Durchführung dieses GitHub-Flow-orientierten Prozesses – einschließlich aller benötigten Git-Befehle – wird in Kapitel 5 unter Testszenario #5 detailliert beschrieben.

5 Prozessdurchführung und Bewertung

Im Sinne des *Design Cycle* im DSR-Ansatz steht dieses Kapitel im Zeichen der Umsetzung und Evaluation des in Kapitel 4 definierten Lösungsartefakts. Ziel ist es, den Git-basierten Kollaborationsprozess praktisch anzuwenden, seine Funktionsfähigkeit unter realitätsnahen Bedingungen zu überprüfen und daraus gewonnene Erkenntnisse systematisch zu analysieren. Die Evaluation erfolgt dabei entlang konkreter Anwendungsszenarien und dient sowohl der Validierung der Prozessdefinition als auch ihrer iterativen Verbesserung.

Dazu werden in **Kapitel 5.1** insgesamt acht Testszenarien definiert und durchgeführt, die zentrale Aspekte der Git-basierten Zusammenarbeit mit SysML v2 adressieren. Jedes Szenario wird dabei einzeln betrachtet, dokumentiert und hinsichtlich seines Ablaufs, der verwendeten Werkzeuge und der erzielten Ergebnisse beschrieben.

In **Kapitel 5.2** erfolgt anschließend eine strukturierte Bewertung des Git-basierten Kollaborationsprozesses. Auf Grundlage der Testergebnisse werden Stärken und Schwächen des gewählten Workflows identifiziert, zentrale Erkenntnisse herausgearbeitet und potenzielle Verbesserungspotenziale aufgezeigt. Die gewonnenen Erkenntnisse fließen in die Reflexion der Prozessgestaltung ein und unterstützen eine fundierte Beurteilung der Eignung des Ansatzes für den Einsatz im MBSE-Kontext.

5.1 Definition und Durchführung von Testszenarien

In diesem Kapitel wird die praktische Durchführung von acht definierten Test-szenarien dokumentiert. Die Szenarien sind so gewählt, dass sie den Git-basierten Kollaborationsprozess im Kontext modellbasierter Systementwicklung sowohl in Grundfunktionen als auch in fortgeschrittenen Anwendungsfällen wie der CI/CD-Integration abdecken. Jedes Testszenario wird strukturiert dokumentiert und hinsichtlich seiner Wirksamkeit, Reproduzierbarkeit und Relevanz für die Kollaboration im Team bewertet.

Die Testszenarien lassen sich in drei Gruppen gliedern: Die ersten vier Szenarien überprüfen grundlegende Funktionen von Git sowie die Kompatibilität der verwendeten Werkzeuge (CSM, JN und GitLab). Testszenario #5 demonstriert die Anwendung des in Kapitel 4.4 definierten GitHub-Flow-basierten Kollaborationsprozesses. Die letzten drei Szenarien untersuchen die automatisierte Verarbeitung von SysML-Modellen mithilfe von GitLab CI/CD – insbesondere Konfigurationsprüfungen, Syntaxanalysen und automatisierte Dokumentenerstellung. Diese Vielfalt ermöglicht eine ganzheitliche Bewertung des entwickelten Prozesses über den gesamten Lebenszyklus eines MBSE-Projekts hinweg und berücksichtigt dabei auch Aspekte der Automatisierung durch die CI/CD-Funktionalitäten von GitLab.

CI/CD beschreibt einen automatisierten Ansatz zur Qualitätssicherung und Bereitstellung von Software-artefakten. In GitLab wird CI/CD über sogenannte Pipelines umgesetzt, die in einer *.gitlab-ci.yml*-Datei definiert werden. In dieser Datei werden einzelne Jobs beschrieben, die bestimmten Stufen (z. B. build, test, deploy) zugeordnet sind. Jeder Job beschreibt einen spezifischen Schritt im Verarbeitungsprozess. Die automatisierte Ausführung dieser Jobs erfolgt bei definierten Triggern, z. B. nach jedem *Commit* oder bei MR. Auf diese Weise wird sichergestellt, dass Änderungen am Modell konsistent verarbeitet und validiert werden können, ohne manuelle Zwischenschritte.

Zur besseren Übersicht sind die Testszenarien in Tabelle 5-1 zusammengefasst. Die Tabelle gibt einen Überblick über die jeweiligen Zielsetzungen und ordnet jedes Szenario einer bestimmten Projektphase innerhalb des GitLab-Prozesses zu. Die Spalte „GitLab-Projektphase“ beschreibt dabei die logische Rolle, die das jeweilige Testszenario im Verlauf eines modellbasierten Projekts einnimmt. Sie orientiert sich an typischen Entwicklungsphasen – beginnend bei der Initialisierung des *Repositories*, über die Bearbeitung, Integration und Wartung der Inhalte bis hin zu spezifischen Aspekten wie Konfiguration, Validierung und Dokumentation. Die Bezeichnungen der Phasen dienen dazu, die Einordnung des Szenarios im Lebenszyklus des Projekts aus Sicht der GitLab-Nutzung zu verdeutlichen.

Tabelle 5-1 Übersicht der definierten Testszenarien und zugehöriger GitLab-Projektphasen

Testszenario	**GitLab Projektphase**
1. Export und Commit-Validierung	Initialisierung
2. Anzeige und Bearbeitung im JN	Bearbeitung
3. Multi-Tool-Kompatibilität (CSM und JN)	Integration
4. Versionierung und Rollback	Wartung
5. GitHub Flow Test (Kollaborationsworkflow)	Kollaboration
6. Automatisierte Konfigurationsprüfung	Konfiguration
7. Automatisierte Syntaxprüfung	Validierung
8. Automatisierte Dokumentenerstellung	Dokumentation

Für die Darstellung und Analyse jedes einzelnen Testszenarios wird in den folgenden Unterkapiteln ein einheitliches Gliederungsformat verwendet. Dies gewährleistet eine klare Nachvollziehbarkeit der Durchführung, Ergebnisse und Bewertungen. Jedes Testszenario wird in einer eigenen Untersektion dokumentiert (z. B. 5.1.1 Export und *Commit*-Validierung, usw.).

Der strukturelle Aufbau jeder Testszenario-Untersektion folgt folgendem Schema:

1. Ziel des Testszenarios: Es wird erläutert, welche Funktionalität oder Eigenschaft des Git-basierten MBSE-Prozesses überprüft werden soll.

2. Testumgebung: Beschreibung der verwendeten Werkzeuge, Konfigurationen und ggf. relevanter *Branches* oder Kernels.

3. Aktivitäten: Darstellung der konkreten Testschritte in nummerierter Form.

4. Durchführung: Beschreibung der Durchführung der zuvor gelisteten Aktivitäten in Fließtextform. Hierbei werden die technischen Abläufe erläutert und bei Bedarf auf unterstützende Bilder verwiesen (z. B. *Screenshots* oder Benutzeroberflächen).

5. Beobachtetes Ergebnis (*Screenshots*): Es werden ausgewählte *Screenshots* präsentiert, die die tatsächlichen Resultate dokumentieren. Alle relevanten *Screenshots* sind zusätzlich im Anhang A5 bis A12 dokumentiert.

6. Diskussion: Reflexion des Testergebnisses im Kontext der Git-basierten MBSE-Zielstellung. Hierbei werden positive Erkenntnisse, mögliche Abweichungen und Verbesserungspotenziale identifiziert.

Diese strukturierte Herangehensweise erlaubt es, die einzelnen Testszenarien vergleichbar darzustellen und ihren Beitrag zur Validierung des entwickelten Prozesses transparent zu machen.

5.1.1 Testszenario #1: Export und *Commit*-Validierung

Ziel des Testszenarios

Ziel dieses Testszenarios ist die Überprüfung, ob ein in CSM erstelltes SysML v2-Modell erfolgreich exportiert und im *.sysml*-Format über GitLab versioniert werden kann. Dabei steht die Sicherstellung der grundsätzlichen Kompatibilität zwischen Modellierungsumgebung und Versionsverwaltung im Vordergrund. Der Test validiert, ob ein sauber strukturierter Export sowie ein vollständiger *Commit*-Vorgang technisch möglich und für weitere Kollaborationsschritte reproduzierbar ist.

Testumgebung

- **Modellierungsumgebung:** CSM mit SysML v2-Plugin
- **Versionskontrolle:** Git CLI
- **Versionsverwaltung:** GitLab UI, *main-Branch*

Aktivitäten

1. Systemmodell in CSM erstellen
2. *.sysml*-Datei exportieren
3. *.sysml*-Datei im lokalen *Repository* speichern – hier *„UAV Civil Drone“*
4. Änderungen in GitLab aktualisieren: *´git pull´* → *´git add .´* → *´git commit -m “nachricht”´* → *´git push´*

Durchführung

Zunächst wurden die *Customer Requirements*, *SOS Requirements* sowie *System Requirements* in CSM modelliert (siehe Bild 5-1). Anschließend erfolgte der Export ausschließlich der Datei *„Requirements.sysml“*, wobei weitere Modellbestandteile wie *Use Cases* und Systemstruktur bewusst nicht exportiert wurden (siehe Bild 5-2). Die exportierte Datei wurde in das lokale Git-*Repository „UAV Civil Drone“* gespeichert. Anschließend wurden die in Schritt 4 beschriebenen Git-Befehle mit der *Commit*-Nachricht *„added requirements“* ausgeführt. Durch den erfolgreichen *Push*-Vorgang wurde die Datei im GitLab-*Interface* (UI) sichtbar gemacht (siehe Bild 5-3).

Beobachtetes Ergebnis (*Screenshots*)

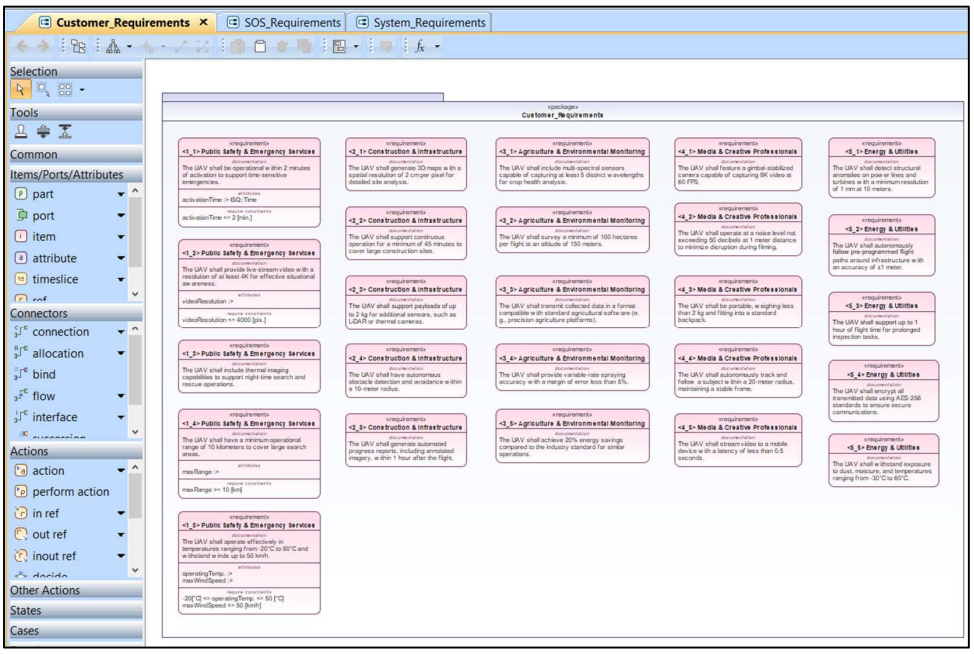


Bild 5-1 Modellierung der Anforderungen in CSM (eigener Screenshot, CSM)

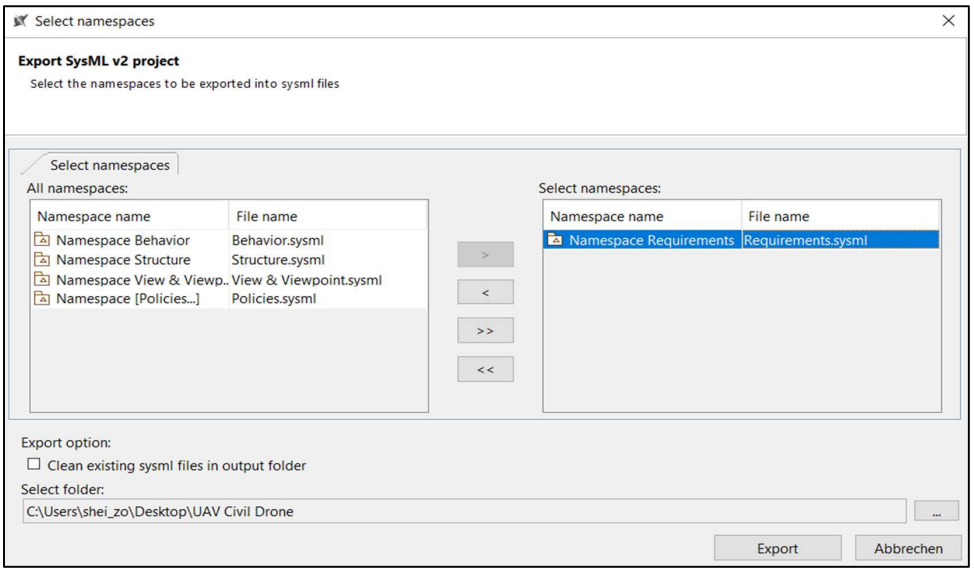


Bild 5-2 *Exportdialog und Dateispeicherung in das lokale Repository (eigener Screenshot, CSM)*

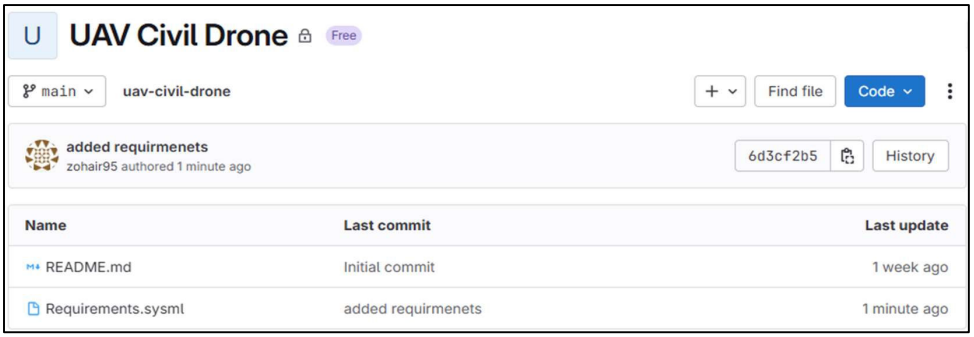


Bild 5-3 *Darstellung der „Requirements.sysml“-Datei im GitLab nach Push-Vorgang (eigener Screenshot, GitLab UI)*

Diskussion

Das Testszenario verlief erfolgreich und bestätigte die grundsätzliche technische Machbarkeit des Exports und der Versionierung eines SysML v2-Modells über Git. Die strukturierte Modellierung in CSM sowie der gezielte Export einzelner Modellteile funktionierten erwartungsgemäß. Ebenso konnte der *Commit*- und *Push*-Vorgang ohne Konflikte durchgeführt werden. Die Sichtbarkeit der Datei im GitLab UI demonstriert die Reproduzierbarkeit und Nachvollziehbarkeit des Vorgangs. Für zukünftige Kollaborationen bildet dieser Test die Grundlage, um weitere Modellbestandteile analog in den Versionsverwaltungsprozess zu integrieren.

5.1.2 Testszenario #2: Anzeige und Bearbeitung im JN

Ziel des Testszenarios

Ziel dieses Tests ist es, das *.sysml*-Modell außerhalb der Modellierungsumgebung zu laden, anzuzeigen und gezielt zu bearbeiten. Die Bearbeitung erfolgt direkt im JN durch textuelle Modifikation der Modellstruktur. Dadurch wird geprüft, inwieweit Git-basierte Workflows mit einer leichten, skriptbasierten Modifikation des Modells kombinierbar sind.

Testumgebung

- **Ausführungsumgebung:** JN mit SysML-Kernel
- **Versionskontrolle:** Git CLI
- **Versionsverwaltung:** GitLab UI, *main-Branch*

Aktivitäten

1. *.sysml*-Datei in JN öffnen – hier *„Requirements.sysml“*
2. In JN SysML-Kernel visualisieren und Änderungen vornehmen: *´%viz <Modellname>´*
3. *.sysml*-Datei im lokalen *Repository* speichern – hier *„UAV Civil Drone“*
4. Änderungen in GitLab aktualisieren: *´git add <Dateiname>´* → *´git commit -m “nachricht“´* → *´git push´*

Durchführung

Beim Öffnen der Datei *„Requirements.sysml“* in JN wurde diese zunächst als reiner Textinhalt angezeigt. Um eine Visualisierung und Bearbeitung zu ermöglichen, wurde der Inhalt in eine separate *Notebook*-Zelle innerhalb eines aktiven SysML-Kernels eingefügt (siehe Bild 5-4). Nach dem Ausführen der entsprechenden Zellen konnten die verfügbaren *Requirement*-Pakete geladen und mittels des Befehls *´%viz <Modellname>´* visualisiert werden (siehe Bild 5-5).

Im Anschluss wurde das Modell gezielt modifiziert. Konkret wurde die Anforderung *„Customer Requirement <5_5>“* erweitert, indem zusätzliche Attribute sowie *Constraints* direkt in der textuellen Notation ergänzt wurden. Die Änderungen wurden innerhalb des Kernels erneut visualisiert und überprüft (siehe Bild 5-6). Abschließend wurden sowohl die angepasste *„Requirements.sysml“*-Datei als auch das zugehörige JN *„UAV_JN.ipynb“* lokal gespeichert, in das *Repository* eingepflegt und per Git CLI auf GitLab mit der Nachricht *„updated Req <5_5>“* hochgeladen (siehe Bild 5-7).

Beobachtetes Ergebnis (*Screenshots*)

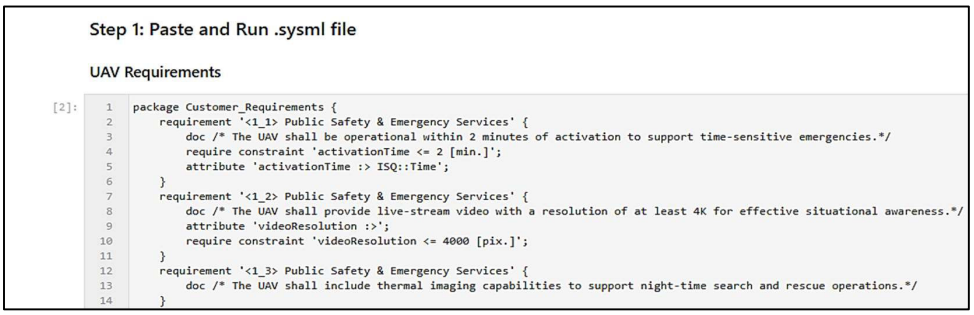


Bild 5-4 Textbasierte Anzeige der Datei „Requirements.sysml“ (eigener Screenshot, JN)

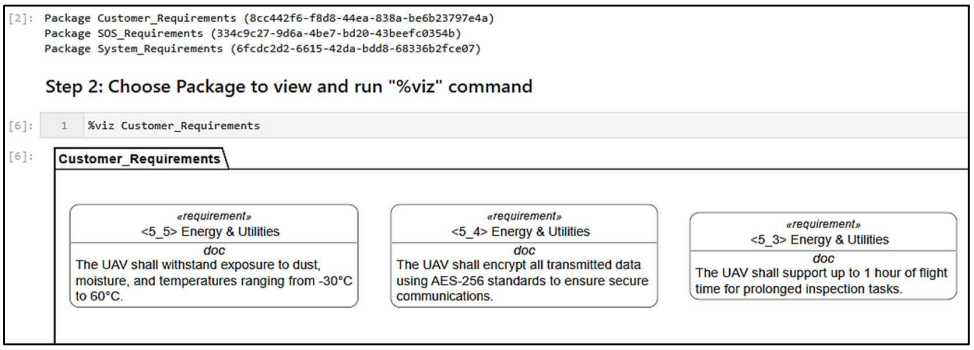


Bild 5-5 Visualisierung der modellierten Anforderungen im SysML-Kernel mittels „%viz“-Befehl (eigener Screenshot, JN)

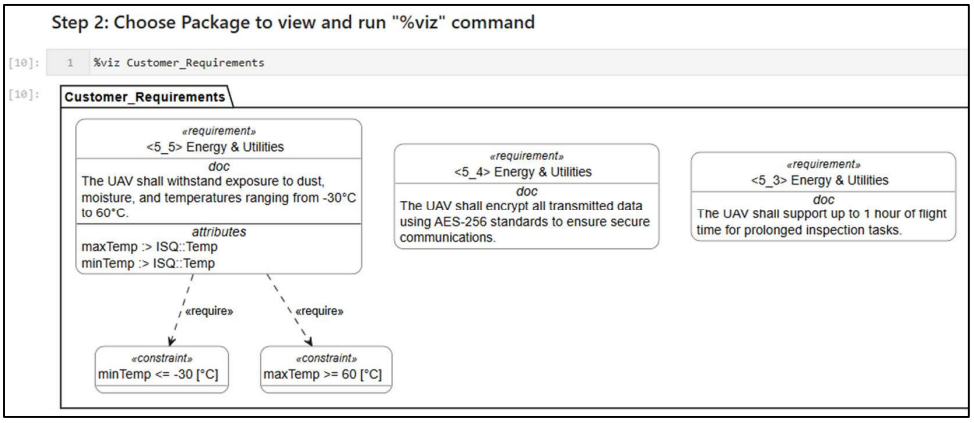


Bild 5-6 Modifikation der Anforderung „Customer Requirement <5_5>“ inkl. neuer Constraints und Attribute (eigener Screenshot, JN)

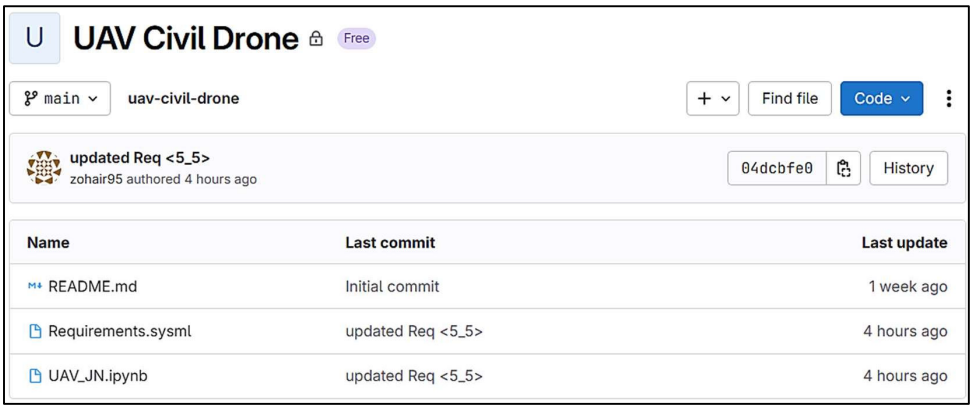


Bild 5-7 Erfolgreicher Push der geänderten Dateien (Requirements.sysml, UAV_JN.ipynb) auf GitLab (eigener Screenshot, GitLab UI)

Diskussion

Das Testszenario konnte erfolgreich durchgeführt werden. Die aus CSM exportierte *„Requirements.sysml“*-Datei wurde korrekt in JN geladen, visualisiert und anschließend textuell angepasst. Die vorgenommenen Änderungen konnten im Kernel validiert und über GitLab versioniert werden, wodurch die prinzipielle Kombinierbarkeit textbasierter Modellbearbeitung mit Git-basierten Workflows bestätigt wurde.

Allerdings zeigte sich eine Einschränkung hinsichtlich der Handhabung: Da JN *.sysml*-Dateien als reine Textdateien interpretieren, ist ein zusätzlicher Zwischenschritt erforderlich, bei dem die Inhalte manuell in eine *Notebook*-Zelle innerhalb eines laufenden SysML-Kernels eingefügt werden müssen. Diese Notwendigkeit der Zwischenspeicherung und Übertragung in ein *.ipynb*-Dateiformat erhöht den manuellen Aufwand. Positiv hervorzuheben ist jedoch, dass GitLab *.ipynb*-Dateien nativ unterstützt und deren Inhalte direkt im Webinterface angezeigt werden können, was die Nachverfolgbarkeit von Änderungen erleichtert.

5.1.3 Testszenario #3: Multi-Tool-Kompatibilität (CSM und JN)

Ziel des Testszenarios

Das Ziel dieses Szenarios ist die Validierung der Werkzeugkompatibilität zwischen CSM und JN. Es wird untersucht, ob textuelle Modelländerungen, die in JN vorgenommen und via GitLab versioniert wurden, anschließend erfolgreich in CSM importiert und angezeigt werden können. Dieser Test stellt sicher, dass ein Toolwechsel im Kollaborationsprozess verlustfrei möglich ist und keine semantischen Inkonsistenzen im Modell entstehen.

Testumgebung

- **Versionskontrolle:** Git CLI
- **Modellierungsumgebung:** CSM mit SysML v2-Plugin

Aktivitäten

1. Änderungen aus GitLab *Repository* lokal aktualisieren: *´git pull´*
2. *.sysml*-Datei in CSM importieren – hier *„Requirements.sysml“*
3. Änderungen in CSM visualisieren

Durchführung

Nach dem Abrufen der aktuellen *Repository*-Inhalte mittels *´git pull´* wurde die zuvor in JN modifizierte Datei *„Requirements.sysml“* in CSM importiert. Der Importvorgang führte zur automatischen Erstellung eines neuen *namespaces* mit der Bezeichnung *„Customer_Requirements“*, welcher dem ersten Paketnamen in der Datei entspricht (siehe Bild 5-8).

Infolgedessen waren im Modell zwei Anforderungs-*namespaces* vorhanden: der ursprüngliche und der neu importierte. Zur gezielten Analyse wurde ein separates SysML v2 *View*-Diagramm mit dem Namen *„Req_New“* erstellt, in dem sowohl die alte als auch die modifizierte Version der Anforderung *„Requirement <5_5>“* nebeneinander dargestellt wurden (siehe Bild 5-9). Dies ermöglichte einen direkten Vergleich der ursprünglichen und der textuell überarbeiteten Anforderungsversion.

Beobachtetes Ergebnis (*Screenshots*)

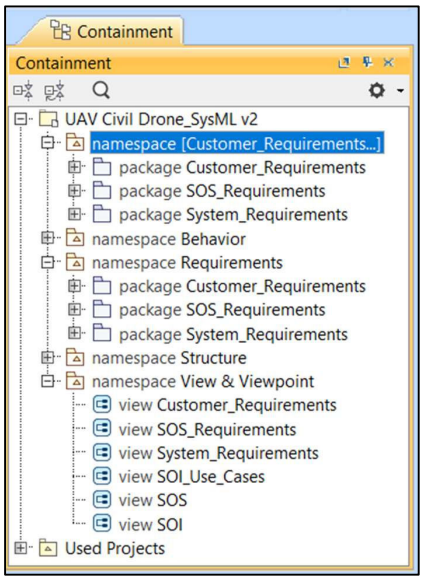


Bild 5-8 Neuer namespace „Customer_Requirements“ nach dem Import der „Requirements.sysml“-Datei (eigener Screenshot, CSM)

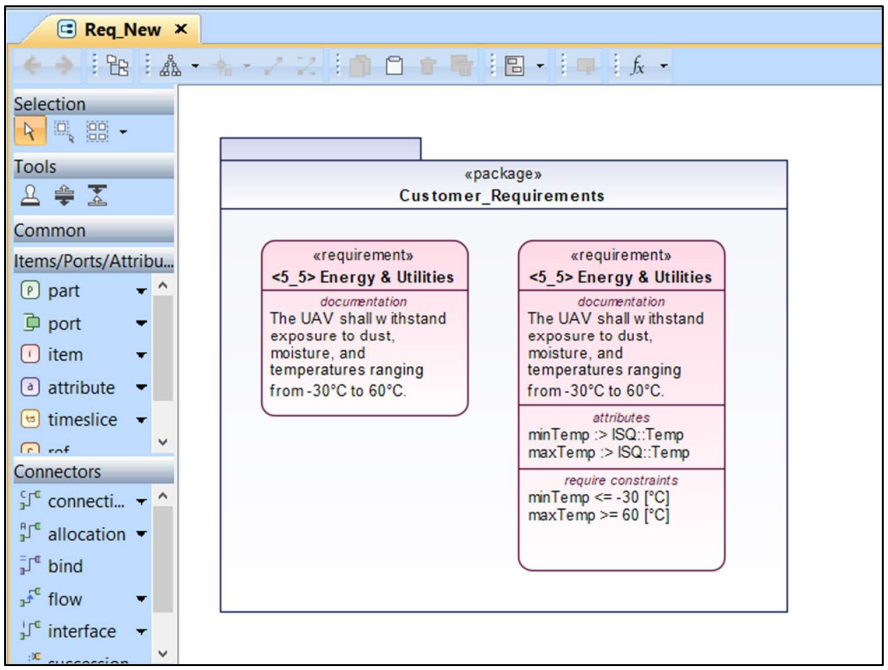


Bild 5-9 Vergleich der ursprünglichen (Links) und modifizierten (Rechts) Anforderung „Requirement <5_5>“ im View-Diagramm „Req_New“ (eigener Screenshot, CSM)

Diskussion

Das Szenario demonstriert grundsätzlich die Kompatibilität zwischen JN und CSM hinsichtlich des Imports von textuell geänderten *.sysml*-Dateien. Die Änderungen, die außerhalb von CSM vorgenommen wurden, konnten ohne technische Fehler importiert und im Modell angezeigt werden. Damit wurde die prinzipielle Toolkompatibilität bestätigt.

Allerdings zeigten sich im Detail Einschränkungen: Der Import einer geänderten Datei erzeugt in CSM automatisch einen neuen *namespace*, selbst wenn bereits ein gleichnamiger Inhalt im Modell vorhanden ist. Dies kann zu redundanten Strukturen führen. Um doppelte Inhalte zu vermeiden, sind zwei Alternativen denkbar: entweder das Löschen des ursprünglichen *namespaces* nach erfolgreichem Import oder das Anlegen eines neuen CSM-Projekts für den Importvorgang.

Ein weiterer Nachteil zeigt sich in der eingeschränkten Visualisierung der Modellelemente nach dem Import einer neuen *.sysml*-Datei: In CSM werden die bestehenden *View*-Diagramme nicht automatisch aktualisiert. Stattdessen muss ein neues *View*-Diagramm manuell erstellt und die relevanten Modellelemente erneut per *Drag-and-Drop* eingefügt werden, um eine visuelle Repräsentation zu erhalten. Dies erschwert die Nachvollziehbarkeit der Änderungen, insbesondere wenn zuvor keine Kommunikation oder Dokumentation erfolgte – wie im Fall der geänderten Anforderung *„Requirement <5_5>“*.

5.1.4 Testszenario #4: Versionierung und Rollback

Ziel des Testszenarios

Dieses Testszenario zielt darauf ab, die Rückverfolgbarkeit und Wiederherstellbarkeit von Modellzuständen innerhalb des Git-basierten Prozesses zu überprüfen. Es wird getestet, ob inkrementelle Änderungen korrekt versioniert werden und ob mit Hilfe von Git-Befehlen wie *´git log´* und *´git revert´* frühere Modellzustände zuverlässig wiederhergestellt werden können. Ziel ist es, die Robustheit des Versionskontrollmechanismus im Hinblick auf Fehlerbehandlung und iterative Entwicklung zu bewerten.

Testumgebung

- **Modellierungsumgebung:** CSM mit SysML v2-Plugin
- **Versionskontrolle:** Git CLI
- **Versionsverwaltung:** GitLab UI, *main-Branch*

Aktivitäten

1. Änderungen in mehreren *Commits* vornehmen
2. Ein *Commit* auswählen und entfernen: *´git log´* → *´git revert <commit>´* → *´git add .´* → *´git commit -m "nachricht"´* → *´git push´*
3. Modellkonsistenz überprüfen

Durchführung

Im Rahmen dieses Tests wurden schrittweise Änderungen am Modell vorgenommen, darunter das Hinzufügen von System *Use Cases* im *Behavior*-Paket. Diese Änderung wurde unter dem *Commit*-Nachricht *„added-behavior"* in das GitLab-*Repository* hochgeladen.

Anschließend wurden über die Git-Befehle *´git log´* und *´git revert <commit>´* gezielt frühere Modellzustände wiederhergestellt. Zunächst wurde mittels *´git log´* der *Commit*-Verlauf eingesehen (siehe Bild 5-10). Daraufhin wurde der *Commit „added-behavior"* (*Commit 89adfb82*) identifiziert und über *´git revert 89adfb82´* zurückgesetzt. Die Änderung wurde mit dem *Commit*-Kommentar *„deleted last commit"* erneut versioniert und in das zentrale *Repository* übertragen. In der GitLab-Oberfläche war zunächst die Datei *„Behavior.sysml"* sichtbar (siehe Bild 5-11), nach dem *Revert*-Vorgang jedoch nicht mehr vorhanden (siehe Bild 5-12), was auf eine erfolgreiche Wiederherstellung des vorherigen Modellzustands hinweist.

Beobachtetes Ergebnis (*Screenshots*)

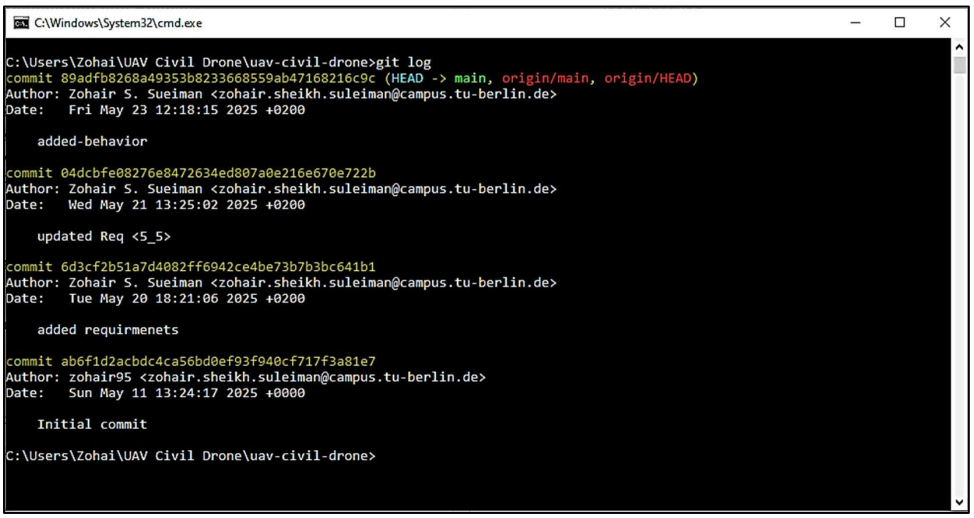


Bild 5-10 Übersicht des ´git log´-Befehls (eigener Screenshot, Git CLI)

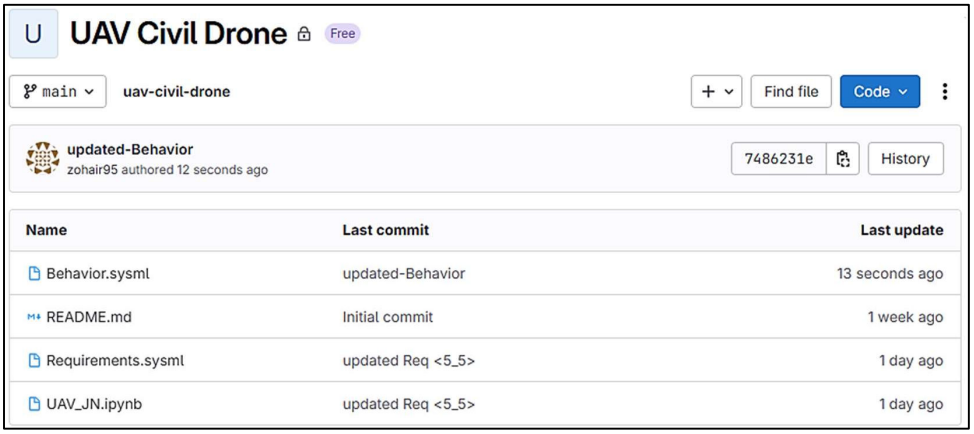


Bild 5-11 GitLab UI mit „Behavior.sysml“ (eigener Screenshot, GitLab UI)

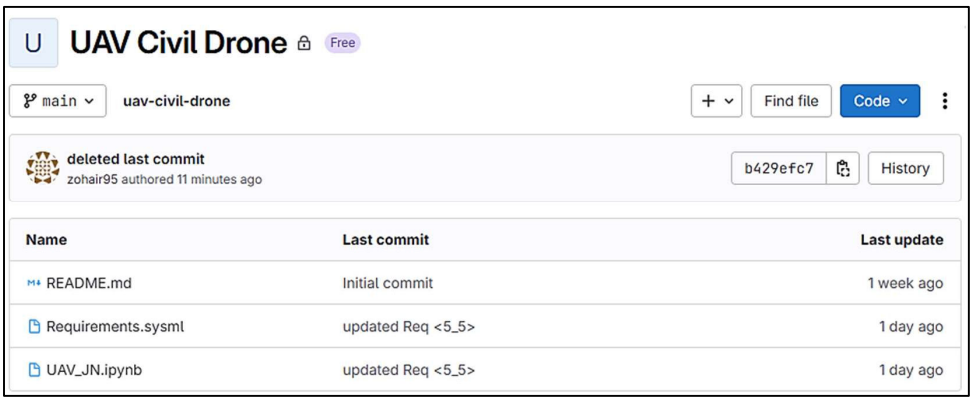


Bild 5-12 GitLab UI ohne „Behavior.sysml“ (eigener Screenshot, GitLab UI)

Diskussion

Das Testszenario bestätigt die korrekte Funktionsweise der Git-basierten Versionierung und die Möglichkeit zur Wiederherstellung früherer Modellzustände mittels *´git revert´*. Die erfolgreiche Entfernung der Datei *„Behavior.sysml“* nach dem Zurücksetzen des *Commits* verdeutlicht, dass Änderungen nicht nur versioniert, sondern auch gezielt rückgängig gemacht werden können.

Im Kontext kollaborativer Arbeitsumgebungen ist jedoch Vorsicht geboten: Das direkte Rückgängigmachen von *Commits* im *main-Branch* kann zu Inkonsistenzen führen, wenn andere Teammitglieder parallel arbeiten. Daher ist es empfehlenswert, Änderungen über einen MR rückgängig zu machen. Dies ermöglicht eine Prüfung der Maßnahme durch andere Beteiligte und bietet eine zusätzliche Qualitätssicherung, bevor Änderungen wirksam werden.

Zusammenfassend zeigt das Testszenario, dass Git ein robustes Werkzeug zur Verwaltung von Modellzuständen darstellt, jedoch organisatorische Maßnahmen wie MRs notwendig sind, um Konflikte in Teamumgebungen zu vermeiden.

5.1.5 Testszenario #5: GitHub Flow Test (Kollaborationsworkflow)

Ziel des Testszenarios

Ziel dieses Tests ist es, den in Kapitel 4.4 beschriebenen GitHub-Flow-Prozess exemplarisch anzuwenden. Der Fokus liegt dabei auf der Erstellung eines *Feature-Branch*, dem Durchlaufen des Review-Prozesses und dem kontrollierten *Merge* in den *main“-Branch*. Durch diesen Test wird überprüft, ob die definierten Kollaborationsregeln im Teamkontext praktisch umsetzbar sind und ob typische Teamaufgaben wie Änderungsdokumentation, Review und *Merge*-Prozesse reibungslos funktionieren.

Testumgebung

- **Versionskontrolle:** Git CLI
- **Versionsverwaltung:** GitLab UI, *feature-* und *main-Branch*

Aktivitäten

1. *feature-Branch* erstellen: *´git checkout -b feature´*
2. Änderungen vornehmen: *´git add .´* → *´git commit -m “nachricht“´* → *´git push -u origin feature´*
3. MR in GitLab UI eröffnen
4. Review und *Merge* – In GitLab UI oder in Git CLI: *´git checkout main´* → *´git pull origin main´* → *´git merge feature´* → *´git push origin main´*

5. (Optional) *feature-Branch* lokal löschen: *´git branch -d feature´*

Durchführung

Im Rahmen dieses Tests wurde ein typischer GitHub-Flow-Kollaborationsprozess durchgeführt. Zunächst wurde lokal ein neuer *feature-Branch* mit dem Namen *„feature/add-new-uc“* erstellt und darin Änderungen an der Datei *„Behavior.sysml“* vorgenommen. Anschließend wurden nach Schritt 2 diese Änderungen in das entfernte *Repository* übertragen.

Im nächsten Schritt wurde über die GitLab-Oberfläche ein MR eröffnet, um die Änderungen vor dem *Merge* in den *main-Branch* zu überprüfen (siehe Bild 5-13 und 5-14). Der Review- und Freigabeprozess erfolgte ebenfalls vollständig in der GitLab UI. Nach erfolgreicher Überprüfung wurde der *feature-Branch* in den *main-Branch* zusammengeführt (siehe Bild 5-15). Abschließend wurde der lokale *Feature-Branch* mit *´git branch -d feature/add-new-uc´* gelöscht, um die lokale Arbeitsumgebung zu bereinigen, da er trotz *Pull*-Vorgang weiterhin lokal sichtbar war.

Beobachtetes Ergebnis (*Screenshots*)

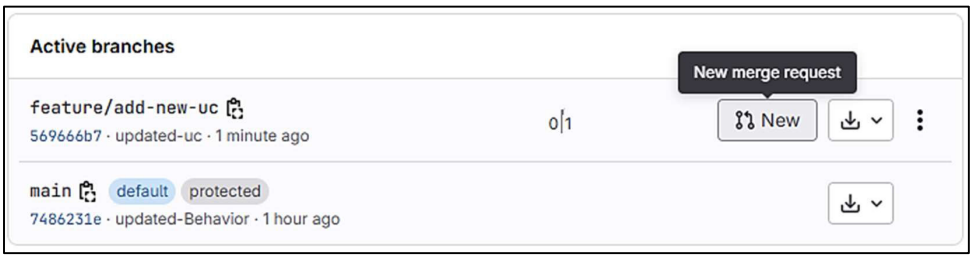


Bild 5-13 Übersicht der Branches „main“ und „feature/add-new-uc“ (eigener Screenshot, GitLab UI)

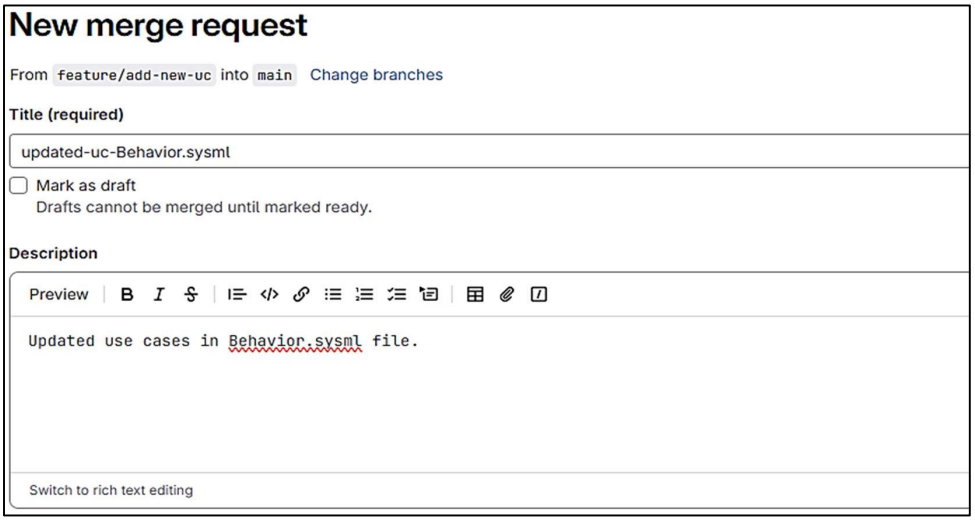


Bild 5-14 Merge Request-Erstellung (eigener Screenshot, GitLab UI)

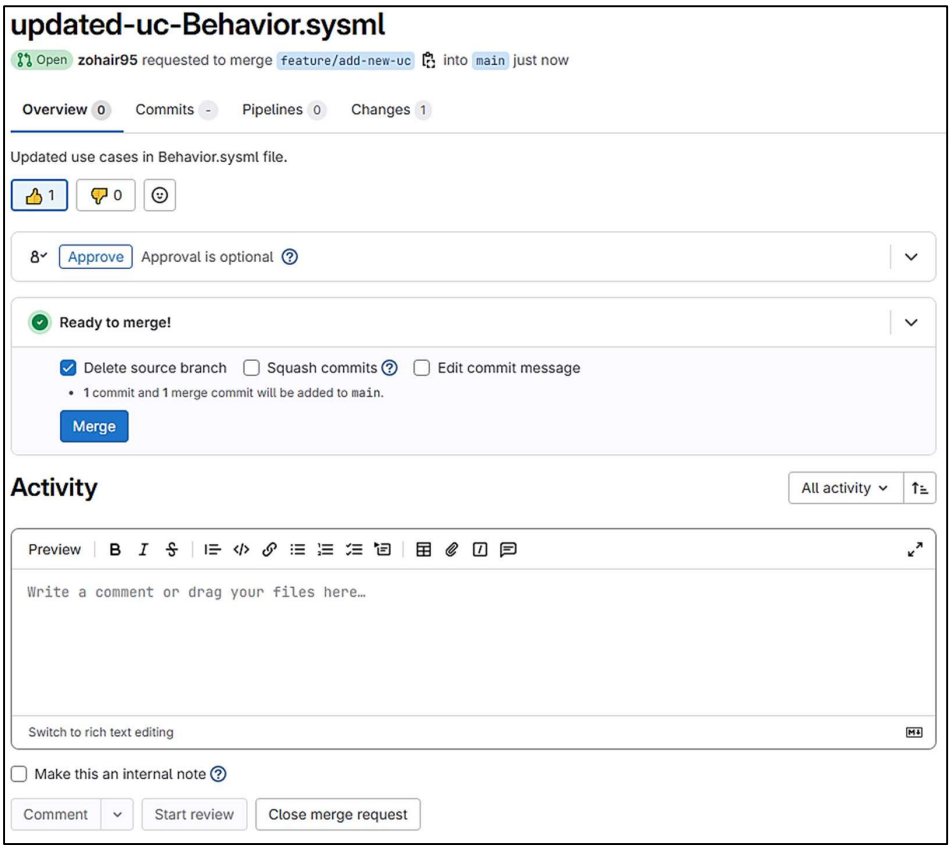


Bild 5-15 Merge-Vorgang des Feature-Branch nach abgeschlossenem Review, Abschluss des Vorgangs über „merge“ (eigener Screenshot, GitLab UI)

Diskussion

Dieses Testszenario bestätigt die erfolgreiche Umsetzung eines GitHub-Flow-Prozesses im Rahmen einer GitLab-gestützten Modellierungsumgebung. Der gesamte Ablauf – von der Erstellung eines *Feature-Branch* über die Änderungsübertragung bis hin zum *Merge* nach abgeschlossenem Review – verlief reibungslos und erfüllte alle definierten Anforderungen an kollaborative Entwicklungsprozesse.

Besonders hervorzuheben ist die Benutzerfreundlichkeit der GitLab UI, die den Review- und *Merge*-Prozess visuell unterstützt und durch intuitive Funktionen wie Reviewer-Zuweisung, Kommentierung und Änderungsverfolgung ergänzt. Die im Git CLI verfügbaren Kommandos für den *Merge*-Prozess sind bei Verwendung der GitLab UI nicht zwingend notwendig, bieten jedoch zusätzliche Flexibilität für fortgeschrittene Nutzer:innen.

Ein wichtiger Hinweis aus diesem Test ist die Notwendigkeit, nach dem *Merge* abgeschlossener *Feature-Branches* lokal zu löschen, um die Arbeitsumgebung übersichtlich zu halten. Diese Praxis unterstützt die langfristige Wartbarkeit und vermeidet Konflikte durch veraltete lokale *Branches*.

5.1.6 Testszenario #6: Automatisierte Konfigurationsprüfung

Ziel des Testszenarios

Das Ziel dieses Testszenarios besteht darin, eine automatisierte Prüfung und Korrektur der Konfiguration von *.sysml*-Dateien zu etablieren, um syntaktische Inkonsistenzen zu beheben, die beim Export aus dem CSM auftreten. Insbesondere betrifft dies die fehlerhafte Behandlung von benutzerdefinierten Bezeichnern innerhalb der textuellen SysML v2-Syntax, bei denen fälschlicherweise keine einfachen Anführungszeichen gesetzt werden, wenn es sich um ein einzelnes Wort handelt (Beispiel: einWort anstelle von 'einWort').

Diese Abweichung tritt systematisch dann auf, wenn das benutzerdefinierte Eingabefeld nur aus einem einzigen, zusammenhängenden Wort besteht – während bei mehrteiligen Werten (mit Leerzeichen) automatisch einfache Anführungszeichen generiert werden (z. B. 'ein Wort').

Um die Lesbarkeit und die maschinelle Verarbeitbarkeit zu verbessern, soll eine Korrektur dieser Fälle direkt im GitLab CI/CD-Pipelineprozess erfolgen. Ziel ist es, fehlerhafte Stellen automatisiert zu erkennen, zu korrigieren und die aktualisierte Datei bei Bedarf zurückzuschreiben.

Testumgebung

- **Skriptausführung:** JN mit Python 3-Kernel
- **Modellierungsumgebung:** CSM mit SysML v2-Plugin
- **Versionskontrolle:** Git CLI
- **Automatisierung:** GitLab CI/CD mit *.gitlab-ci.yml*

Aktivitäten

1. Erstellung einer *.sysml*-Datei mit einem absichtlich eingebauten Formatierungsfehler
2. Implementierung eines Python-Skripts (*fix_config.ipynb*), das systematisch alle benutzerdefinierten Eingaben analysiert und bei Bedarf korrekt formatiert.
3. Integration des Skripts in die GitLab-CI-Pipeline durch Erweiterung der Datei *.gitlab-ci.yml* mit einem neuen Job *check_config*
4. Ausführung des Skripts sowohl lokal als auch über GitLab CI zur Überprüfung der Funktionalität und automatischen Korrektur
5. *Push* der Änderungen ins zentrale GitLab-*Repository* über die Git-Befehle: *´git add .´ → ´git commit -m "nachricht"´ → ´git push´*

Durchführung

Für die Durchführung des Testszenarios wurde zunächst eine bewusst fehlerhafte *.sysml*-Datei mit dem Namen *Structure.sysml* erstellt. Diese Datei enthält zwei Part-Dekomposition-Pakete für die Subsysteme SOS und SOI. Nach dem Export aus CSM zeigte sich, dass Benutzereingaben ohne Leerzeichen nicht in einfache Anführungszeichen (' ') gesetzt wurden (siehe Bild 5-17, links). Betroffen waren u. a. Begriffe wie SOI, SOS, *Communication* und *Airframe*.

Im Anschluss wurde das Python-Notebook *fix_config.ipynb* ausgeführt (siehe Anhang A10). Dieses durchsucht alle *.sysml*-Dateien im Verzeichnis *System_Models* und überprüft deren Inhalt zeilenweise auf eine korrekte Konfiguration. Das enthaltene Skript identifiziert mithilfe regulärer Ausdrücke fehlerhaft gesetzte oder fehlende Anführungszeichen im Bereich der Benutzereingaben und ersetzt diese durch eine korrekt formatierte Version. Nach erfolgreicher Ausführung wird die korrigierte *.sysml*-Datei automatisch mit der fehlerhaften Ursprungsversion überschrieben (Bild 5-17, rechts).

Nach dem erfolgreichen Test des Notebooks im lokalen JN-Umfeld wurde die GitLab-CI-Konfigurationsdatei *.gitlab-ci.yml* um einen neuen Job erweitert, der *fix_config.ipynb* automatisch ausführt, sobald *.sysml*-Dateien im *Repository* geändert werden (Bild 5-16).

Im Zuge der Anpassung wurde außerdem die Verzeichnisstruktur im GitLab-*Repository* optimiert: Alle *.sysml*-Dateien wurden in einen zentralen Ordner mit dem Namen *System_Models* verschoben, um den Zugriff innerhalb der CI-Pipeline zu vereinheitlichen und zukünftige Automatisierungsschritte zu erleichtern (Bild 5-18).

Beobachtetes Ergebnis (*Screenshots*)

```yaml
check_config:
  stage: check-config
  image: python:3.10                    # Leichtgewichtiges Python-Image
  before_script:
    - pip install jupyter               # Jupyter wird für die Konvertierung benötigt
  script:
    - jupyter nbconvert --to script fix_config.ipynb     # Notebook in .py-Datei konvertieren
    - python fix_config.py                               # fix_config Skript ausführen
  only:
    changes:
      - System_Models/*.sysml           # Trigger nur bei Änderungen an SysML-Dateien
```

Bild 5-16 Ausschnitt aus .gitlab-ci.yml mit Definition des neuen Jobs zur automatisierten Konfigurationsprüfung (eigener Screenshot, .gitlab-ci.yml)

```
package SOS {
    part SOS {
        part 'Communication System';
        part SOI;
        part 'Payload Sensor System';
        part 'Ground Control Sytem';
        part 'Air Traffic Management';
    }
}
package SOI {
    part SOI {
        part Airframe {
            part 'Wing type';
        }
        part 'Payload System' {
            part 'Environmental Sensors';
            part Camera;
            part 'Delivery Meachnism';
        }
        part 'Propulsion System';
        part 'Safety System' {
            part 'Return to Home';
            part 'Emergency Landing';
        }
        part 'Power System';
        part Communication {
            part 'RF Communication';
        }
        part 'Navigation System' {
            part GPS;
        }
        part 'Flight System';
    }
    part 'Fixed Wing' :> SOI::Airframe::'Wing type';
    part 'Roraty Wing' :> SOI::Airframe::'Wing type';
    part Hybrid :> SOI::Airframe::'Wing type';
}
```

```
package 'SOS' {
    part 'SOS' {
        part 'Communication System';
        part 'SOI';
        part 'Payload Sensor System';
        part 'Ground Control Sytem';
        part 'Air Traffic Management';
    }
}
package 'SOI' {
    part 'SOI' {
        part 'Airframe' {
            part 'Wing type';
        }
        part 'Payload System' {
            part 'Environmental Sensors';
            part 'Camera';
            part 'Delivery Meachnism';
        }
        part 'Propulsion System';
        part 'Safety System' {
            part 'Return to Home';
            part 'Emergency Landing';
        }
        part 'Power System';
        part 'Communication' {
            part 'RF Communication';
        }
        part 'Navigation System' {
            part 'GPS';
        }
        part 'Flight System';
    }
    part 'Fixed Wing' :> 'SOI'::'Airframe'::'Wing type';
    part 'Roraty Wing' :> 'SOI'::'Airframe'::'Wing type';
    part 'Hybrid' :> 'SOI'::'Airframe'::'Wing type';
}
```

Bild 5-17 Links: Ursprüngliche Exportdatei mit fehlerhafter Formatierung; Rechts: Datei nach automatischer Korrektur durch fix_config.ipynb (eigener Screenshot, Structure.sysml)

Name	Last commit
.ipynb_checkpoints	Added Req. and updated System_Models
SysML_Reports	Added Req. and updated System_Models
System_Models	Added Req. and updated System_Models
.gitlab-ci.yml	added comments to .gitlab-ci.yml file
README.md	Initial commit
fix_config.ipynb	Added Req. and updated System_Models

Bild 5-18 Organisationsstruktur der Repository für eine saubere und konsistente Ablagestruktur (eigener Screenshot, GitLab UI)

Diskussion

Das Skript *fix_config.ipynb* erfüllte die erwartete Funktionalität und konnte die zuvor identifizierten Formatierungsprobleme in den *.sysml*-Dateien erfolgreich beheben. Insbesondere wurden Benutzereingaben ohne Leerzeichen, die beim Export aus CSM nicht korrekt mit einfachen Anführungszeichen versehen waren, automatisiert korrigiert.

Die Idee zur Aufnahme dieses Testszenarios entstand im Verlauf der Entwicklung des Syntaxprüfskripts, das im folgenden Testszenario #7 beschrieben wird. Ursprünglich war lediglich eine Syntaxprüfung geplant. Während der ersten Testläufe zeigte sich jedoch, dass bestimmte Formatierungsfehler – insbesondere fehlende Anführungszeichen bei bestimmten Texteingaben – zu Validierungsfehlern führten. Daher wurde Testszenario #6 als notwendiger Zwischenschritt eingeführt, um die zugrunde liegenden Probleme zunächst zu beheben. Wäre die von CSM erzeugte Syntax an dieser Stelle korrekt, hätte dieser zusätzliche Schritt nicht implementiert werden müssen.

Eine bekannte Einschränkung des aktuellen Skripts besteht darin, dass es nur einmal ausgeführt werden sollte. Bei einer zweiten Ausführung werden bestimmte Konnektoren der Form 'xx' :: 'yy' :: 'zz' fehlerhaft umformatiert, was zu Syntaxproblemen führen kann. Bei einem dritten Durchlauf wird dieser Fehler jedoch wieder korrigiert. Trotz mehrerer Anpassungsversuche konnte dieses Verhalten bislang nicht behoben werden, ohne gleichzeitig funktionierende Teile des Codes zu beeinträchtigen. Daher ist es essenziell, die Ausführung des Notebooks auf einen einmaligen Durchlauf pro Änderungszyklus zu beschränken.

Zur Validierung der Korrekturen wurden die modifizierten *.sysml*-Dateien erneut in den CSM importiert. Der Import verlief fehlerfrei, sodass bestätigt werden konnte, dass die vorgenommenen Anpassungen zu einer konsistenten und verarbeitbaren SysML-Syntax führen.

5.1.7 Testszenario #7: Automatisierte Syntaxprüfung

Ziel des Testszenarios

Ziel dieses Testszenarios ist die automatisierte Prüfung der syntaktischen Korrektheit von *.sysml*-Dateien im Verzeichnis *System_Models*. Der Fokus liegt auf der Überprüfung standardisierter SysML-Begriffe wie *part*, *requirement*, *action* etc., die im ersten Teil der SysML-Textnotation vorkommen. Die Syntax in SysML ist in zwei Bereiche gegliedert: die standardisierte Notation (z. B. Schlüsselwörter) und die nutzergenerierten Inhalte, welche typischerweise in einfachen Anführungszeichen gesetzt werden.

Das hier entwickelte Skript überprüft ausschließlich die standardisierten Begriffe auf korrekte Schreibweise und Groß-/Kleinschreibung. Bei Abweichungen werden diese erkannt und mit Angabe der entsprechenden Zeilennummer zur Korrektur gemeldet.

Testumgebung

- **Skriptausführung:** JN mit Python 3-Kernel
- **Modellierungsumgebung:** CSM mit SysML v2-Plugin

- **Versionskontrolle:** Git CLI
- **Automatisierung:** GitLab CI/CD mit *.gitlab-ci.yml*

Aktivitäten

1. Manuelles Einfügen syntaktischer Fehler in eine *.sysml*-Datei
2. Entwicklung eines Python-Skripts (*syntax_check.ipynb*) zur automatisierten Syntaxprüfung
3. Definition gültiger Begriffe basierend auf OMG SysML v2.0 *Language Specification* im Skript
4. Anpassung der *.gitlab-ci.yml*, um das Notebook im CI-Prozess automatisch auszuführen
5. Durchführung des Tests lokal und in GitLab CI
6. Überprüfung und Dokumentation der Ausgaben

Durchführung

Für die Durchführung des Tests wurde das Notebook *syntax_check.ipynb* entwickelt (vgl. Anhang A11). Dieses analysiert *.sysml*-Dateien im Verzeichnis *System_Models* hinsichtlich der korrekten Verwendung standardisierter Begriffe aus der SysML v2.0-Spezifikation (OMG Systems Modeling Language, 2024). Die gleiche Liste an Schlüsselwörtern kam bereits im Skript *fix_config.ipynb* zum Einsatz.

Das Notebook durchläuft jede *.sysml*-Datei zeilenweise und prüft:

- ob gültige Schlüsselwörter korrekt geschrieben und kleingeschrieben sind,
- ob öffnende und schließende geschweifte Klammern {} korrekt verwendet werden,
- ob Zeilen mit Semikolon ; abgeschlossen sind.

Zur Validierung wurde die Datei *Structure.sysml* gezielt mit zwei Fehlern versehen: In Zeile 4 wurde *part* versehentlich als *patt*, in Zeile 13 als *past* geschrieben (vgl. Bild 5-20). Nach dem Ausführen des Skripts wurden beide Fehler wie erwartet erkannt und mit korrekter Fehlermeldung ausgegeben (vgl. Bild 5-21).

Die Ausführung innerhalb der GitLab CI wurde über die *.gitlab-ci.yml* konfiguriert, wobei die automatische Analyse bei jeder Änderung an Dateien im *System_Models*-Verzeichnis erfolgt (vgl. Bild 5-19).

Beobachtetes Ergebnis (*Screenshots*)

```yaml
syntax_check:
  stage: syntax-check
  image: python:3.10
  before_script:
    - pip install jupyter
  script:
    - jupyter nbconvert --to script syntax_check.ipynb   # Notebook in ausführbares Skript umwandeln
    - python syntax_check.py                             # Syntaxprüfung starten
  only:
    changes:
      - System_Models/*.sysml
```

Bild 5-19 Ausschnitt aus .gitlab-ci.yml mit Definition des neuen Jobs zur automatisierten Syntaxprüfung (eigener Screenshot, .gitlab-ci.yml)

```
1  package 'SoS_System_Context' {
2      part 'SoS' {
3          part 'Communication System';
4          patt 'UAV Civil Drone';
5          part 'Payload Sensor System';
6          part 'Ground Control Sytem';
7          part 'Air Traffic Management';
8      }
9  }
10 package 'SoI_Subsystems' {
11     part 'UAV Civil Drone' {
12         part 'Airframe : UA-AF';
13         past 'Payload System : EO';
14         part 'Propulsion System : UA-PS';
15         part 'Safety System : SS';
16         part 'Power System : EP';
17         part 'Communication : C2';
18         part 'Navigation System : NAV';
19         part 'Flight System : UA-FCS';
20     }
21 }
22
```

Bild 5-20 Manuell eingefügte Syntaxfehler in Structure.sysml (eigener Screenshot, Structure.sysml)

```
Scanning 'Behavior.sysml'...
No issues found.

Scanning 'Customer_Requirements.sysml'...
No issues found.

Scanning 'SoS_Requirements.sysml'...
No issues found.

Scanning 'Structure.sysml'...
 Line 4: unknown term 'Unknown term: 'patt''
   → patt 'UAV Civil Drone';
 Line 13: unknown term 'Unknown term: 'past''
   → past 'Payload System : EO';
```

Bild 5-21 Konsolenausgabe des Syntaxprüfskripts (eigener Screenshot, Python 3-Kernel)

Diskussion

Der Test verlief erfolgreich. Das Skript konnte die fehlerhafte Schreibweise von standardisierten SysML-Schlüsselwörtern zuverlässig erkennen und mit klaren Fehlermeldungen kennzeichnen. Auch syntaktische Fehler wie fehlende Semikolons oder nicht geschlossene Klammern wurden korrekt detektiert.

Im Unterschied zum vorherigen Test (5.1.6) greift dieses Skript nicht in die Datei ein, sondern meldet ausschließlich die gefundenen Abweichungen. Eine automatische Korrektur findet nicht statt, was aus Nachvollziehbarkeits- und Validierungsgründen im Rahmen eines Review-Prozesses von Vorteil ist.

Die Syntaxprüfung ist insbesondere dann hilfreich, wenn *.sysml*-Modelle manuell im Texteditor oder durch Skripte angepasst werden. Durch die Integration in GitLab CI kann eine kontinuierliche Überprüfung gewährleistet werden, ohne die Arbeitsschritte der Modellierer zu beeinträchtigen.

5.1.8 Testszenario #8: Automatisierte Dokumentenerstellung

Ziel des Testszenarios

Ziel dieses Tests ist die automatisierte Erstellung eines aktuellen SysML-Reports, sobald Änderungen an *.sysml*-Dateien im Verzeichnis *System_Models* festgestellt werden. Der Report soll stets die aktuellste Version der modellierten Pakete enthalten und automatisch im GitLab *Repository* bereitgestellt werden. Damit steht allen Beteiligten jederzeit ein konsistenter Überblick über die Modellstruktur zur Verfügung, ohne dass der Report manuell gepflegt werden muss.

Testumgebung

- **Skriptausführung:** JN mit Python 3-Kernel
- **Ausführungsumgebung:** JN mit SysML-Kernel
- **Versionskontrolle:** Git CLI
- **Automatisierung:** GitLab CI/CD mit *.gitlab-ci.yml*

Aktivitäten

1. Erstellung des Notebooks *generate_render_notebook.ipynb* zur Analyse und Aufbereitung der *.sysml*-Dateien
2. Integration des Notebooks in den GitLab-CI-Prozess über *.gitlab-ci.yml*
3. Konfiguration von GitLab CI zur Speicherung des Berichts als Artefakt bei Änderungen an den *.sysml*-Dateien

4. Lokale Testläufe des Notebooks mit aktivem SysML-Kernel zur Verifikation der Ergebnisdarstellung
5. Durchführung automatisierter Testläufe bei Dateiänderungen im *Repository*

Durchführung

Die initiale Idee bestand darin, automatisch einen visuell aufbereiteten SysML-Report im HTML- oder PDF-Format zu erzeugen. Da GitLab CI jedoch keine Unterstützung für die Ausführung von Notebook-Zellen mit dem SysML-Kernel bietet, musste ein alternativer Ansatz gewählt werden. Stattdessen wird ein JN erzeugt, das die Struktur des Reports vorbereitet und später manuell im SysML-Kernel ausgeführt werden kann.

Das Skript in *generate_render_notebook.ipynb* analysiert alle *.sysml*-Dateien im Verzeichnis *System_Models* und extrahiert daraus die enthaltenen Pakete. Für jedes erkannte Paket wird automatisch eine Notebook-Zelle generiert, die den *%viz*-Befehl aufruft, um die grafische Darstellung im SysML-Kernel zu ermöglichen. Dadurch lässt sich der Bericht manuell im Notebook ausführen und vollständig visualisieren (vgl. Bild 5-23 und Bild 5-24).

Die erzeugten Reports werden automatisch im GitLab CI-Prozess gespeichert – sowohl lokal im Ordner *SysML_Reports* als auch im GitLab-Projekt als Artefakt (vgl. Bild 5-22). Zur besseren Nachverfolgbarkeit wird dabei eine semantische Versionierung verwendet:

- Wird eine *.sysml*-Datei inhaltlich verändert, wird die *Patch*-Version erhöht (z. B. v1.1.0 → v1.1.1).
- Werden *.sysml*-Dateien hinzugefügt oder gelöscht, wird die *Minor*-Version erhöht (z. B. v1.1.1 → v1.2.0).

Eine *Major*-Version ist bislang nicht definiert, da der Report noch experimentell eingesetzt wird – das Skript kann jedoch jederzeit an veränderte Anforderungen angepasst werden.

Beobachtetes Ergebnis (*Screenshots*)

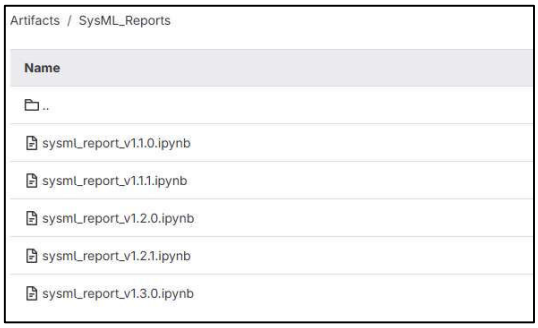


Bild 5-22 SysML-Reports als GitLab-Artefakte (eigener Screenshot, GitLab)

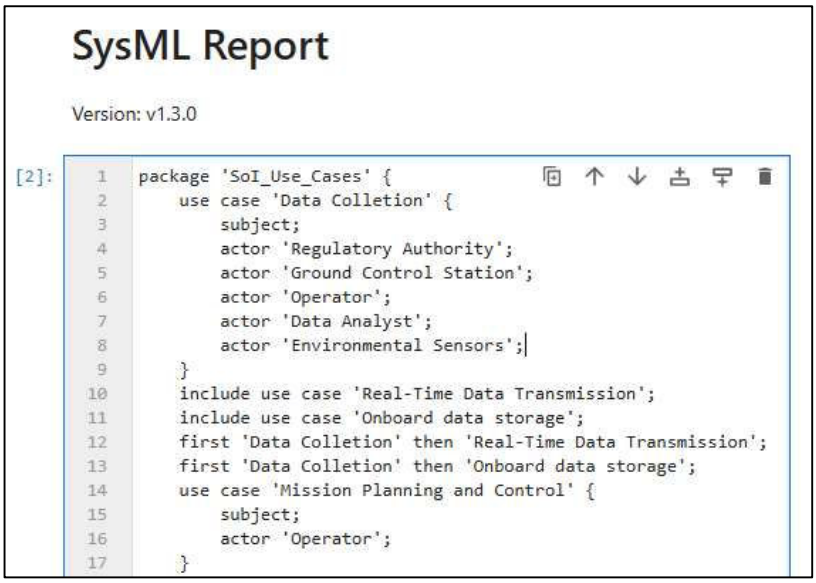


Bild 5-23 Generierter SysML-Report in JN, Teil 1 (eigener Screenshot, JN mit SysML-Kernel)

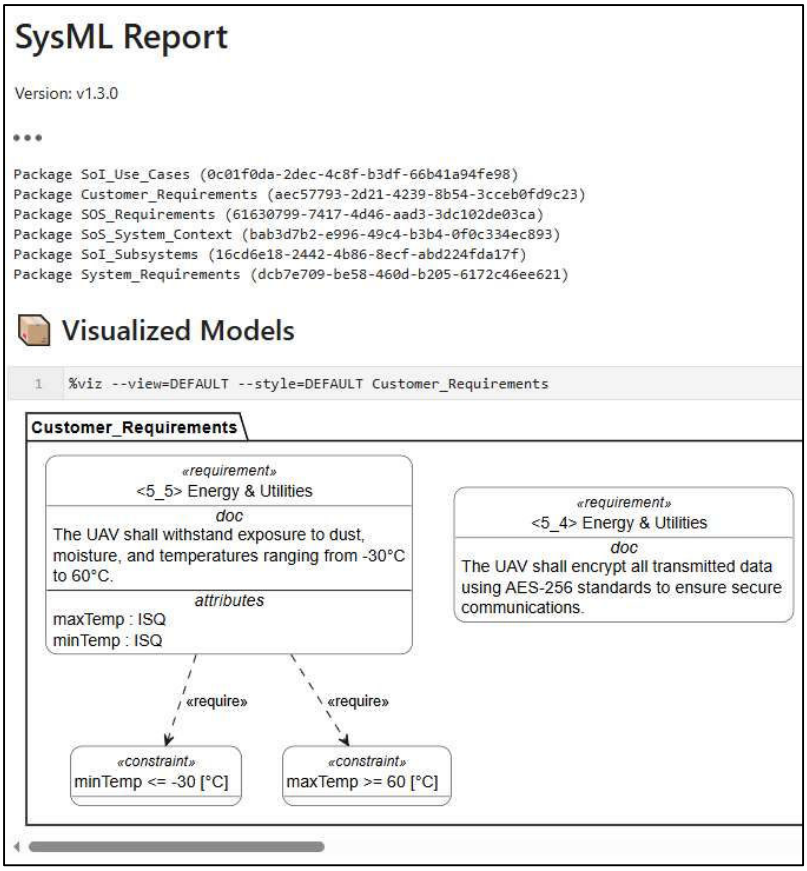


Bild 5-24 Generierter SysML-Report in JN, Teil 2 (eigener Screenshot, JN mit SysML-Kernel)

Diskussion

Die automatisierte Erstellung von SysML-Reports über GitLab CI konnte erfolgreich umgesetzt werden – wenngleich mit der Einschränkung, dass die eigentliche Visualisierung der Modelle manuell im SysML-Kernel erfolgen muss. Eine vollständige Automatisierung der grafischen Darstellung war nicht möglich, da GitLab keine native Unterstützung für den SysML-Kernel bietet.

Dennoch stellt das Testskript einen wertvollen Beitrag zur Dokumentation dar: Es bereitet die notwendigen Visualisierungsbefehle vor und ermöglicht es, die aktuellsten Systemmodelle jederzeit strukturiert einzusehen. Besonders hilfreich ist der Ansatz im kollaborativen Umfeld, da alle Teammitglieder auf einen stets aktuellen Report zugreifen können.

Die Kombination aus lokalem Testlauf mit manuell ausführbarem Report und automatischer Speicherung als Artefakt bietet eine praktikable Lösung, um Modelländerungen nachvollziehbar zu dokumentieren. Optional kann der Report vor dem *Push*-Vorgang lokal ausgeführt und überprüft werden. So wird nicht nur ein gültiges Artefakt erzeugt, sondern auch eine nachvollziehbare Integration der Modelländerungen im GitLab-*Repository* im Verzeichnis *SysML_Reports* gewährleistet.

5.2 Bewertung des Git-basierten Kollaborationsprozesses

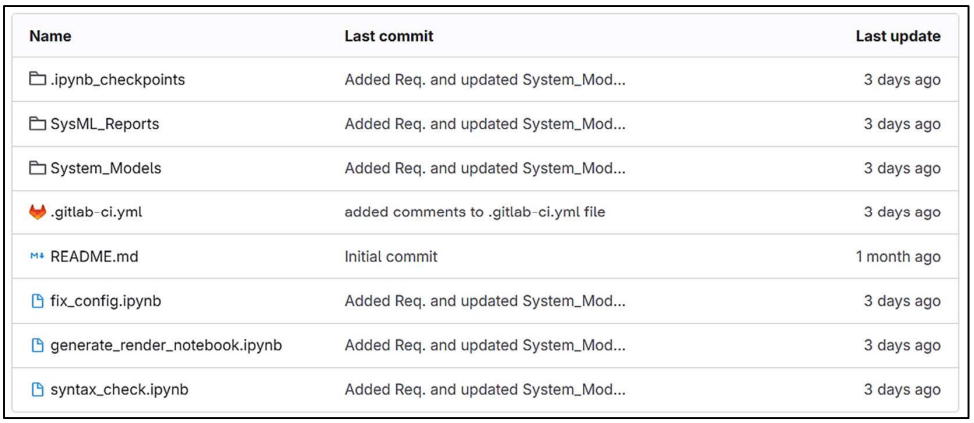
Im Anschluss an die Durchführung der acht definierten Testszenarien konnte der entwickelte Git-basierte Kollaborationsprozess im Hinblick auf Struktur, Verantwortlichkeiten, Integration sowie Vor- und Nachteile systematisch bewertet werden. Im Folgenden werden zentrale Erkenntnisse und offene Fragen diskutiert, die sich aus der praktischen Erprobung ergeben haben.

Die Bewertung erfolgt dabei auch in Bezug zu den in Kapitel 3.5 identifizierten teaminternen Herausforderungen, welche in fünf zentrale Kategorien eingeordnet wurden – darunter fehlende klare Prozesse, Änderungsmanagement, Informationsmanagement sowie plattformtechnische Grenzen. Ziel des Git-basierten Kollaborationsprozesses war es unter anderem, diese Defizite durch methodische und technische Maßnahmen gezielt zu adressieren. Im Folgenden wird daher auch reflektiert, inwieweit der entwickelte Ansatz zur Überwindung dieser Herausforderungen beitragen konnte.

Finale Repository-Struktur und Funktionsweise der Pipeline

Die endgültige Struktur des GitLab-*Repositories* wurde im Laufe der Tests mehrfach angepasst und spiegelt nun eine klare Trennung der Systemmodelle (Verzeichnis *System_Models*), generierter Reports (*SysML_Reports*) sowie unterstützender CI-Konfigurationen wider (siehe Bild 5-25). Dadurch ist gewährleistet, dass modellierte Inhalte, Konfigurationslogik und Auswertungen strukturell voneinander

getrennt, aber funktional aufeinander abgestimmt sind. Die entwickelte *.gitlab-ci.yml*-Konfiguration erkennt automatisch Änderungen in *.sysml*-Dateien und führt je nach Änderungstyp unterschiedliche Pipelines aus – etwa Konfigurations- oder Syntaxprüfungen oder die Erzeugung von Reports.



Name	Last commit	Last update
.ipynb_checkpoints	Added Req. and updated System_Mod...	3 days ago
SysML_Reports	Added Req. and updated System_Mod...	3 days ago
System_Models	Added Req. and updated System_Mod...	3 days ago
.gitlab-ci.yml	added comments to .gitlab-ci.yml file	3 days ago
README.md	Initial commit	1 month ago
fix_config.ipynb	Added Req. and updated System_Mod...	3 days ago
generate_render_notebook.ipynb	Added Req. and updated System_Mod...	3 days ago
syntax_check.ipynb	Added Req. and updated System_Mod...	3 days ago

Bild 5-25 Übersicht über die finale Repository-Struktur (eigener Screenshot, GitLab UI)

Die Gestaltung der Pipeline orientierte sich dabei nicht an einer Best Practice-Vorgabe, sondern diente der Erprobung möglicher CI/CD-Funktionen im MBSE-Kontext. Trotzdem wurde bewusst darauf geachtet, Konfigurationen zu wählen, die übertragbar und erweiterbar sind. Einschränkungen zeigten sich insbesondere bei den Notebooks *fix_config.ipynb* und *generate_render_notebook.ipynb*. Während *fix_config.ipynb* zwar automatisch Syntax korrigiert, kann es in komplexeren Modellen zu Überschreibungsproblemen kommen. Das Notebook zur Reportgenerierung wiederum hängt stark vom SysML-Kernel ab und lässt sich nicht vollständig automatisieren, da GitLab CI keine SysML-Kernel-Ausführung unterstützt.

Herausforderung zu vieler Pipeline-Stufen

Die nach Abschluss der Testszenarien entstandene CI-Pipeline umfasst drei aufeinanderfolgende Stufen, die jeweils unterschiedliche Aufgaben übernehmen – von der Konfigurationsprüfung bis zur automatisierten Berichtserzeugung. Die Struktur der finalen CI-Pipeline wurde zur besseren Nachvollziehbarkeit in einem SysML v2 *Action*-Diagramm modelliert und ist in Bild 5-26 dargestellt.

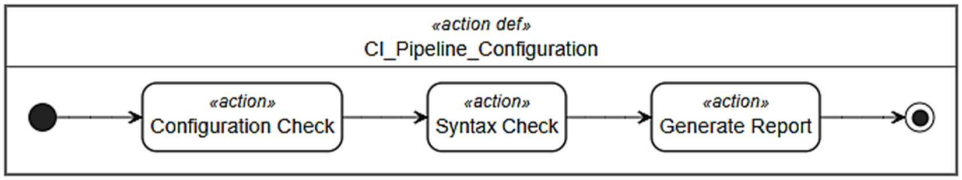


Bild 5-26 Action-Diagramm mit den drei CI-Stufen im finalen Repository (eigene Darstellung, JN mit SysML-Kernel)

Diese Aufteilung ermöglicht eine saubere Trennung der Verarbeitungsschritte und eine höhere Nachvollziehbarkeit des Ablaufs. Gleichzeitig sollte jedoch bedacht werden, dass mit jeder zusätzlichen Stufe der Pflegeaufwand steigt und die Pipeline komplexer wird. Dies kann insbesondere bei kleinen Teams oder bei häufigen, inkrementellen Änderungen zu Mehraufwand führen und erfordert erhöhte Aufmerksamkeit bei der Pflege und Nutzung der CI-Infrastruktur.

Darüber hinaus ist es empfehlenswert, bereits vor der Implementierung einer CI/CD-Pipeline bewusst zu entscheiden, welche Schritte tatsächlich als eigene Stufen ausgeführt werden sollen. Nicht jeder Verarbeitungsschritt muss zwangsläufig durch die GitLab-Pipeline automatisiert werden. Manche Aufgaben – etwa erste Modellprüfungen oder Formatierungschecks – lassen sich lokal vor dem *Commit* durchführen. Eine wohlüberlegte Balance zwischen lokalen Prüfungen und serverseitigen Automatisierungen ist entscheidend, um die Pipeline handhabbar zu halten und trotzdem eine verlässliche Qualitätssicherung zu gewährleisten

Ein praktikabler Lösungsansatz zur Reduktion dieses Aufwands besteht in der Einführung strukturierter Checklisten innerhalb der MR. Diese könnten dazu genutzt werden, typische Fehlerquellen vorab zu prüfen und sicherzustellen, dass alle relevanten Anforderungen vor dem *Merge* erfüllt sind. In Verbindung mit der in Kapitel 3.4.2 vorgestellten RACI-Matrix kann dadurch zudem eine klare Abgrenzung der Prüfverantwortlichkeiten zwischen SE-Management-Team und den technischen Teams unterstützt werden.

Verantwortlichkeit für Branches

Ein wichtiger Aspekt betrifft die organisatorische Verantwortung für das Anlegen und Verwalten von *Branches*. In der Praxis stellt sich die Frage, ob dieser Schritt durch das SE-Management-Team zentral koordiniert oder durch die jeweiligen Fachteams selbstständig übernommen werden sollte. Beide Optionen haben Vor- und Nachteile: Eine zentrale Verwaltung kann Konsistenz und Nachverfolgbarkeit sichern, erhöht jedoch den Koordinationsaufwand. Dezentrale *Branch*-Erstellung durch die technischen Teams ist flexibler, birgt aber die Gefahr von Unübersichtlichkeit, da viele Akteure gleichzeitig *Branches* anlegen können. Um diesem Risiko zu begegnen, ist es sinnvoll, vorab verbindliche Namenskonventionen, Rollenverteilungen und Qualitätskriterien für *Branches* festzulegen und teamübergreifend zu kommunizieren.

In kleineren Projekten oder bei enger Abstimmung im Team ist die dezentrale Variante häufig praktikabel. In größeren Teams oder bei gleichzeitiger Arbeit an mehreren Systemmodellen empfiehlt sich die zentrale Übernahme der *Branch*-Erstellung durch das SE-Management-Team, da so eine einheitliche Struktur gewahrt und die Kollaboration gezielter gesteuert werden kann.

Bild 5-27 zeigt ein solches *Swimlane*-Diagramm zur *Repository*-Verwaltung, das den dezentralen Ablauf exemplarisch abbildet: Das SE-Management-Team übernimmt die Pflege und Strukturierung des GitLab-*Repositories*. Die technischen Teams arbeiten eigenständig an Systemmodellen, indem sie nach Bedarf *Feature-Branches* anlegen, Änderungen lokal in der *.sysml*-Datei im Verzeichnis *System_Models* vornehmen und ihre Arbeit anschließend in das zentrale *Repository* hochladen. Sobald alle relevanten Änderungen implementiert sind, initiiert das SE-Management-Team einen MR, um den Integrationsprozess zu starten. Im Review-Prozess wird geprüft, ob die Modelländerungen den vereinbarten Qualitätskriterien entsprechen. Bei positiver Bewertung werden die Änderungen in den *main-Branch* übernommen. Bei Ablehnung werden die entsprechenden Fachteams benachrichtigt und zur Überarbeitung aufgefordert.

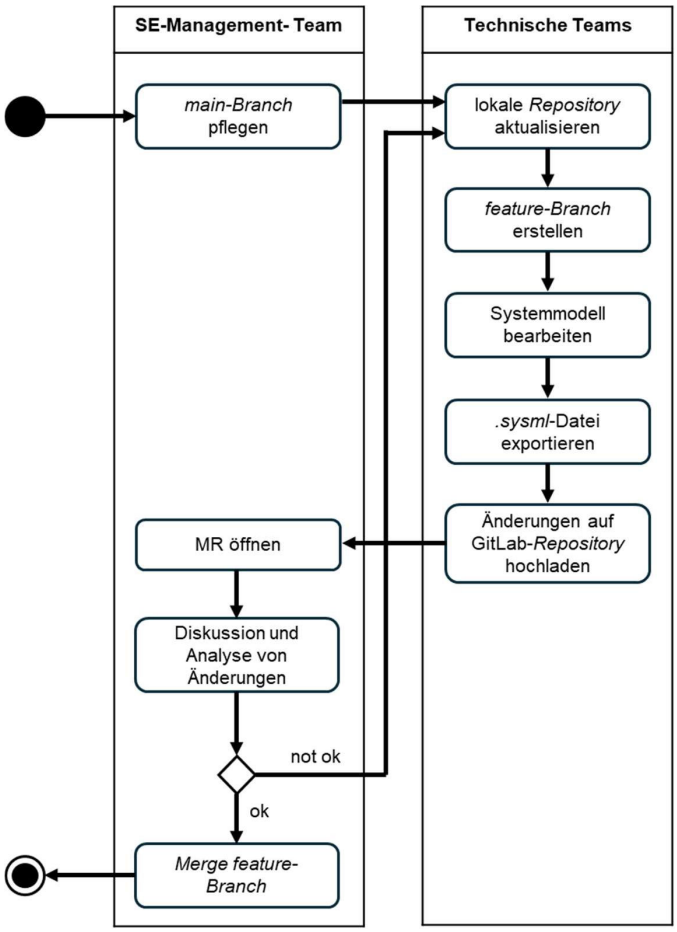


Bild 5-27 Swimlane-Diagramm zur Repository-Verwaltung (eigene Darstellung)

Rolle zur Repository-Verwaltung

Im Verlauf der Testszenarien wurde deutlich, dass die Einrichtung, Pflege und Weiterentwicklung des *Repositories* sowie der zugehörigen CI/CD-Pipeline spezialisiertes Wissen erfordert, das nicht bei allen Teammitgliedern vorausgesetzt werden kann. Besonders in den initialen Phasen – etwa beim Aufsetzen der CI-Konfiguration, der Anpassung der *.gitlab-ci.yml*-Datei sowie der Einrichtung einer sinnvollen Projektstruktur – entsteht ein erheblicher Mehraufwand. Darüber hinaus ist eine kontinuierliche Überwachung notwendig, um sicherzustellen, dass alle automatisierten Prozesse zuverlässig funktionieren.

Vor diesem Hintergrund erscheint es sinnvoll, eine dedizierte Rolle für die technische Betreuung von Git und CI/CD-Prozessen vorzusehen – beispielsweise in der Funktion eines „Konfigurations-Managers“. Diese Person wäre nicht nur für die Wartung der CI/CD-Konfigurationen und die Weiterentwicklung der Git-Struktur verantwortlich, sondern könnte auch als zentrale Ansprechperson bei Fragen zur Git-Nutzung dienen. Voraussetzung dafür ist ein solides Verständnis der Git-Konzepte und -Workflows. Optional könnte diese Rolle auch Schulungen oder *Onboardings* für neue Teammitglieder durchführen, um einen einheitlichen Wissensstand im Umgang mit Git sicherzustellen.

Entscheidend ist dabei die Klärung folgender Fragen: Welche Aufgaben fallen regelmäßig an, welche sind eher einmalig? Wie hoch ist der laufende Aufwand zur Betreuung der Infrastruktur? Und wie lässt sich diese Rolle sinnvoll in die bestehende Teamstruktur integrieren, ohne unnötige Hierarchieebenen zu schaffen?

Integration in Toollandschaft und Rückverfolgbarkeit

Ein zentraler Vorteil des Git-basierten Ansatzes in Kombination mit der textuellen Notation von SysML v2 besteht in der Möglichkeit, Änderungen an einzelnen Systemelementen gezielt nachzuverfolgen. Bild 5-28 zeigt exemplarisch, wie Modifikationen an *package*- und *part*-Elementen im *.sysml*-Dateiformat über die GitLab-Oberfläche differenziert dargestellt werden. Jede Änderung im Text – selbst auf Zeilenebene – lässt sich so revisionssicher dokumentieren und dem jeweiligen Bearbeitungszeitpunkt sowie dem verantwortlichen Teammitglied zuordnen.

Auch die Integration des JN erweist sich im Rahmen der gewählten Toolkette als sinnvoll. Die Möglichkeit, *Markdown*-Zellen zur strukturierten Dokumentation zu nutzen und gleichzeitig durch den SysML-Kernel systemtechnische Inhalte darzustellen, schafft eine leistungsfähige Umgebung für die Modellvisualisierung und Berichtserstellung. Dies fördert nicht nur die Lesbarkeit, sondern ermöglicht auch eine bessere Trennung zwischen Beschreibung, Code und generierten Ergebnissen.

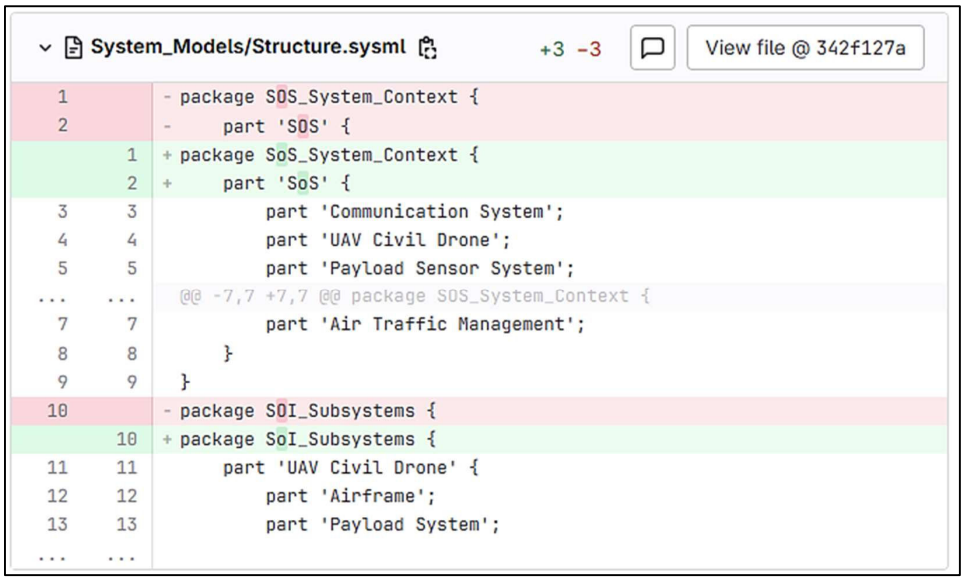


Bild 5-28 Änderungsübersicht in GitLab für .sysml-Dateien (eigener Screenshot, GitLab UI)

Die Einführung und Etablierung eines Git-basierten Workflows in der modellbasierten Systementwicklung bringt zunächst gewisse Herausforderungen mit sich – insbesondere im Hinblick auf die Einrichtung der Toolkette, die CI/CD-Konfiguration und die Schulung der Beteiligten im Umgang mit Git und der textuellen Notation. Langfristig jedoch zeigt sich, dass diese Investitionen zu einem robusten, nachvollziehbaren und effizient wartbaren Entwicklungsprozess führen. Sobald die Abläufe etabliert und die Rollen geklärt sind, sinkt der operative Aufwand deutlich, und die Vorteile eines versionierbaren Systemmodells treten deutlich hervor.

6 Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Untersuchung der Integration von Git mit SysML v2-Modellen zur Versionskontrolle und Kollaboration im modellbasierten Systems Engineering (MBSE). Im Fokus standen insbesondere die Potenziale und Herausforderungen einer Git-basierten Verwaltung von Systemmodellen im Kontext sicherheitskritischer Systeme. Die Arbeit orientierte sich am *Design Science Research* (DSR)-Ansatz und kombinierte methodische Analyse mit praktischer Gestaltung und Evaluation.

Zur Beantwortung der Forschungsfragen wurde ein exemplarisches UAV-Modell mit dem *Cameo Systems Modeler* (CSM) und dem SysML v2-Plugin entwickelt. Dieses wurde über GitLab in Kombination mit *Jupyter Notebook* (inkl. SysML-Kernel) versioniert, automatisiert geprüft und kollaborativ weiterentwickelt. Die technische Umsetzung erfolgte iterativ anhand von acht Testszenarien, die eine praxisnahe Bewertung des gewählten Workflows ermöglichten.

Die im Rahmen dieser Arbeit aufgeworfenen Forschungsfragen lassen sich auf Grundlage der durchgeführten Analysen, Implementierungen und Testszenarien wie folgt beantworten:

1. Wie lässt sich die bestehende Kollaborationsstruktur im MBSE unter Berücksichtigung relevanter Luftfahrtstandards und Teamstrukturen analysieren?

Die Analyse der Forschungsumgebung zeigte, dass Kollaboration im MBSE stark durch organisatorische und regulatorische Rahmenbedingungen geprägt ist. Insbesondere in sicherheitskritischen Domänen wie der Luftfahrt müssen Standards wie ARP4754B, ARP4761A und ISO/IEC 15288 berücksichtigt werden, die klare Verantwortlichkeits- und Dokumentationsstrukturen verlangen. Die durchgeführten Umfragen und RACI-Analysen zeigten zudem, dass eine effektive Kollaboration eine eindeutige Rollenverteilung, eine abgestimmte Methodik sowie eine geeignete technische Infrastruktur voraussetzt.

Die in Kapitel 3.5 identifizierten teaminternen Herausforderungen bestätigten die Relevanz dieser Analyse. Sie wurden systematisch den in der Literatur beschriebenen Problemfeldern zugeordnet und bildeten eine zentrale Grundlage für die Gestaltung des Git-basierten Kollaborationsprozesses. Die Arbeit zeigt exemplarisch, wie sich solche praktischen Herausforderungen durch gezielte methodische und technische Maßnahmen adressieren lassen – etwa durch die Definition klarer Rollen (vgl. RACI-Matrix), eine strukturierte *Repository*-Organisation oder den Einsatz CI-gestützter Prüfroutinen.

2. Welche Werkzeug-Konfigurationsrichtlinien sind erforderlich, um eine effiziente Nutzung von Git mit SysML v2 zu gewährleisten?

Die Integration von Git mit SysML v2 erfordert spezifische Konfigurationsrichtlinien, um eine effiziente und skalierbare Zusammenarbeit zu ermöglichen. Dazu zählen:

- Eine modulare Strukturierung des *Repositories* zur besseren Trennung von Verantwortlichkeiten,
- eine konsistente *Branch*-Strategie zur Steuerung von Entwicklungsaktivitäten,
- die Nutzung standardisierter *Merge-Request*-Vorlagen mit technischen und inhaltlichen Prüfkriterien,
- sowie die Definition von Konventionen zur textuellen Modellierung.

Die textuelle Notation von SysML v2 ermöglicht eine quellcodeähnliche Handhabung von Systemmodellen, wodurch sich Modelländerungen strukturiert verwalten, nachverfolgen und automatisiert verarbeiten lassen. Dieser Ansatz erlaubt es, Prinzipien aus dem *DevOps*-Umfeld in das MBSE zu integrieren – etwa durch CI/CD, automatisierte Prüfprozesse und die verlässliche Generierung einheitlicher Modellstände.

3. Wie kann ein Git-basierter Arbeitsablauf für SysML v2 gestaltet und implementiert werden?

Der entworfene Kollaborationsprozess orientiert sich am GitHub Flow und wurde um MBSE-spezifische Erweiterungen ergänzt. Die finale *Repository*-Struktur, die in Kapitel 5 dokumentierte CI/CD-Pipeline sowie das *Swimlane*-Diagramm zur Verantwortlichkeitsverteilung ermöglichen eine nachvollziehbare, skalierbare Arbeitsweise. Besondere Aufmerksamkeit erhielt die Rolle eines potenziellen „Konfiguration Managers“, der für Wartung, technische Unterstützung und Schulung zuständig sein könnte.

4. Inwiefern ermöglicht dieser Arbeitsablauf eine verbesserte Nachverfolgbarkeit und Effizienz in der modellbasierten Entwicklung sicherheitskritischer Systeme?

Die Git-basierte Verwaltung von SysML v2-Modellen erlaubt eine detaillierte und transparente Nachverfolgung aller Änderungen auf Elementebene. Änderungen an *Packages*, *Parts* oder anderen Modellkomponenten sind versioniert, vergleichbar und im GitLab-UI visuell nachvollziehbar (vgl. Bild 5-28). Dies fördert nicht nur die Rückverfolgbarkeit, sondern erleichtert auch die Kommunikation zwischen Fachteams. Zudem ermöglicht die Integration von GitLab, *Jupyter Notebook* und dem SysML-Kernel eine nahtlose Verbindung zwischen Modellierung,

Dokumentation und Automatisierung. Dies verbessert die Effizienz insbesondere bei repetitiven Aufgaben, bei der Analyse von Modellständen sowie beim Review technischer Inhalte.

Die im Rahmen dieser Arbeit gewonnenen Erkenntnisse zeigen, dass die Integration von Git in die modellbasierte Systementwicklung mit SysML v2 nicht nur technisch möglich, sondern auch methodisch sinnvoll ist. Insbesondere die Nutzung der textuellen Notation von SysML v2 bietet neue Gestaltungsmöglichkeiten für die Verwaltung, Nachverfolgbarkeit und Automatisierung von Systemmodellen. Der modellierte Git-basierte Arbeitsablauf zeigt, wie Prinzipien aus der Softwareentwicklung, wie beispielsweise CI/CD, strukturiert in das MBSE übertragen werden können, um nachvollziehbare und wartbare Modellstände zu erzeugen.

Ein zentrales Ergebnis ist die Erkenntnis, dass die Offenheit der SysML v2-Textnotation in Kombination mit Git die technische Grundlage schafft, um Systemmodelle ähnlich wie Quellcode zu versionieren. Damit wird eine effizientere Zusammenarbeit über Teamgrenzen hinweg möglich – sowohl innerhalb eines Projekts als auch perspektivisch in interorganisationalen Konstellationen. Zugleich wird deutlich, dass diese Offenheit auch eine stärkere methodische und organisatorische Orchestrierung erfordert. Eine Git-basierte Kollaboration setzt voraus, dass klare Spielregeln zur *Branch*-Nutzung, zu Verantwortlichkeiten und zur Toolverwendung etabliert sind. Die in dieser Arbeit entwickelte RACI-Matrix, das *Swimlane*-Diagramm zur Rollenverteilung und die CI-Pipeline zeigen exemplarisch, wie ein solcher Rahmen gestaltet werden kann.

Die Erfahrungen aus den Testszenarien belegen zudem, dass insbesondere in der Einführungsphase ein erheblicher Konfigurations- und Betreuungsaufwand entsteht. Die Einrichtung der CI/CD-Pipeline, die Pflege der GitLab-Struktur sowie die Tool-Integration erfordern spezifisches Wissen und technisches Verständnis. Dieser initiale Aufwand sollte jedoch nicht als Hürde, sondern als Investition in eine langfristig skalierbare und transparente Modellierungskultur betrachtet werden. In einem interdisziplinären Umfeld wie dem DLR kann eine solche Infrastruktur die Zusammenarbeit zwischen SE-Management und technischen Teams deutlich strukturieren und beschleunigen – insbesondere, wenn sie auf konkrete, zuvor identifizierte Herausforderungen reagiert, wie sie in Kapitel 3.5 erhoben wurden.

Gleichzeitig sind einige Limitationen zu berücksichtigen. Die grafische Darstellung von Anforderungen in SysML v2 ist aktuell noch nicht auf dem Niveau von SysML v1 – insbesondere, wenn es um Matrixansichten oder die übersichtliche Navigation großer Anforderungsbäume geht. Eine hybride Lösung, etwa durch die Kopplung mit einem externen Anforderungsmanagement-Werkzeug wie Jira oder DOORS, erscheint gegenwärtig sinnvoll. Auch die Plattformabhängigkeit einzelner Tools (etwa des *Cameo Systems Modelers*) und Einschränkungen in der *Jupyter-Notebook*-Integration stellen technische Herausforderungen dar, die den Einsatz im

Alltag erschweren können. Zudem ist die Kollaboration in dieser Arbeit nur innerhalb eines Teams betrachtet worden. Die Erweiterung auf externe Partner oder Zulieferer bringt weitere organisatorische und sicherheitstechnische Fragestellungen mit sich, die in zukünftigen Arbeiten untersucht werden sollten.

Vor diesem Hintergrund ergeben sich mehrere Anschlussfragen für die weitere Forschung:

- Wie lassen sich Git-basierte Modellierungsprozesse in größeren Organisationen oder zwischen verschiedenen Unternehmen skalieren?
- Welche Rolle können API und *Services* von SysML v2 spielen, um eine modellzentrierte Integration über Toolgrenzen hinweg zu ermöglichen?
- Wie muss ein Organisationskonzept für modellbasierte Entwicklungsinfrastrukturen aussehen, das sowohl methodische Leitlinien als auch technische Unterstützung (z. B. in Form eines Konfigurationsmanagers) umfasst?

Auch die Weiterentwicklung der Toollandschaft – insbesondere hinsichtlich der Benutzerfreundlichkeit von SysML v2-Werkzeugen und der Interoperabilität zwischen Modellierungsumgebungen – bleibt ein zentrales Forschungsthema.

Insgesamt zeigen die Ergebnisse, dass die Kombination von Git und SysML v2 in der Lage ist, bestehende Lücken in der Nachvollziehbarkeit und Versionskontrolle in MBSE-Prozessen zu schließen. Die methodische Gestaltung solcher Prozesse bleibt jedoch ein kritischer Erfolgsfaktor. Die hier erarbeiteten Strukturen bieten dafür einen praxisnahen Ausgangspunkt – sowohl für die interne Optimierung im DLR als auch für die Weiterentwicklung von *Best Practices* in der modellbasierten Entwicklung sicherheitskritischer Systeme.

Literaturverzeichnis

Ahlbrecht, A., Lukić, B., Zaeske, W., & Durak, U. (2024). Exploring SysML v2 for Model-Based Engineering of Safety-Critical Avionics Systems. *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*, 1–8. https://doi.org/10.1109/DASC62030.2024.10749311

Alsaqqa, S., Sawalha, S., & Abdel-Nabi, H. (2020). Agile Software Development: Methodologies and Trends. *International Journal of Interactive Mobile Technologies (iJIM)*, *14*(11), 246. https://doi.org/10.3991/ijim.v14i11.13269

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., & Cunningham, W. (2001). *Manifesto for Agile Software Development*. https://agilemanifesto.org/

Costello, T. (2012). RACI—Getting Projects "Unstuck." *IT Professional*, *14*(2), 64–63. https://doi.org/10.1109/MITP.2012.41

Cui, J. (2024). *Research on DevOps Architecture Design and Git Flow Code Workflow Architecture Design: A case study*. https://doi.org/10.13140/RG.2.2.20367.60327

Dassault Systèmes. (2023, May 22). *Cameo Systems Modeler*. Dassault Systèmes. https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler

exapp.ca. (2024, March 25). *Agile software development: Everything you need to know*. https://www.nexapp.ca/en/blog/agile-software-development

Friedenthal, S. (2024, January). *INCOSE IW SysML v1 to SysML v2 Transition Information Session January 28, 2024*.

Friedenthal, S., Moore, A., & Steiner, R. (2009). *OMG Systems Modeling Language (OMG SysML™) Tutorial September, 2009*.

Ghodke, G. M., & Chavan, T. (2024). An Overview of Git. *International Journal of Scientific Research in Modern Science and Technology*, *3*(6), 17–23. https://doi.org/10.59828/ijsrmst.v3i6.216

Gowda, P. G. A. N. (2022). *Git branching and release strategies*. https://doi.org/10.5281/ZENODO.14221771

Gräßler, I., Thiele, H., Grewe, B., & Hieb, M. (2022). Responsibility Assignment in Systems Engineering. *Proceedings of the Design Society*, *2*, 1875–1884. https://doi.org/10.1017/pds.2022.190

Haberfellner, R., De Weck, O., Fricke, E., & Vössner, S. (2019). *Systems Engineering: Fundamentals and Applications*. Springer International Publishing. https://doi.org/10.1007/978-3-030-13431-0

Hevner, March, Park, & Ram. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75. https://doi.org/10.2307/25148625

Hick, H., Bajzek, M., & Faustmann, C. (2019). Definition of a system model for model-based development. *SN Applied Sciences*, *1*(9). https://doi.org/10.1007/s42452-019-1069-0

International Council on Systems Engineering. (2007, September). *SYSTEMS ENGINEERING VISION 2020*. International Council on Systems Engineering (INCOSE). https://sdincose.org/wp-content/uploads/2011/12/SEVision2020_20071003_v2_03.pdf

ISO/IEC & IEEE. (2023). *ISO/IEC/IEEE 15288:2023(en), Systems and software engineering—System life cycle processes*. https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:15288:ed-2:v1:en

Jayaraman, K. D., & Rastogi, D. (2025). *Best Practices for DevOps Integration in Enterprise Software Development*. https://doi.org/10.5281/ZENODO.14769328

Kaiser. (2013). *Kaiser—Rahmenwerk zur Modellierung plausibler Systemstrukturen.pdf*.

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The devOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution Press, LLC.

Krupa, G. P. (2019). Application of Agile Model-Based Systems Engineering in aircraft conceptual design—Full. *The Aeronautical Journal*, *123*(1268), 1561–1601. https://doi.org/10.1017/aer.2019.53

Li, Z., Faheem, F., & Husung, S. (2024). Collaborative Model-Based Systems Engineering Using Dataspaces and SysML v2. *Systems*, *12*(1), 18. https://doi.org/10.3390/systems12010018

Madni, A. M., Augustine, N., & Sievers, M. (Eds.). (2023). *Handbook of Model-Based Systems Engineering*. Springer International Publishing. https://doi.org/10.1007/978-3-030-93582-5

May, M., & Zerwas, T. (2025). *Enabling broader access to MBSE system models using collaborative engineering platforms and SysMLv2*.

Nyembe, F. H., Van Der Poll, J. A., & Lotriet, H. H. (2023). Formal Methods for an Agile Scrum Software Development Methodology. *Proceedings of the International Conference on Advanced Technologies*, ICAT23. https://doi.org/10.58190/icat.2023.35

OMG Systems Modeling Community. (n.d.). *Systems-Modeling/SysML-v2-Release: The latest incremental release of SysML v2. Start here*. Retrieved April 21, 2025, from https://github.com/Systems-Modeling/SysML-v2-Release

OMG Systems Modeling Language. (2024). *SysML v2.0, Part 1: Language Specification* (Version Version 2.0 Beta 2). https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF

prostep ivip Association. (2023). *Recommendation_SysML_WF-IF*. https://www.ps-ent-2023.de/fileadmin/prod-download/Recommendation_SysML_WF-IF.pdf

SAE Aerospace Recommended Practice. (2023a). *ARP4754B - Guidelines for Development of Civil Aircraft and Systems*.

SAE Aerospace Recommended Practice. (2023b). *ARP4761A - Guidelines for Conducting the Safety Assessment Process on Civil Aircraft, Systems, and Equipment*. SAE International.

Schwaber, K., & Sutherland, J. (2020, November). *Der Scrum Guide—Der gültige Leitfaden für Scrum: Die Spielregeln*. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-German.pdf

Spinellis, D. (2012). Git. *IEEE Software*, *29*(3), 100–101. https://doi.org/10.1109/MS.2012.61

Walden, D. D. & International Council on Systems Engineering (Eds.). (2023). *INCOSE systems engineering handbook* (Fifth edition). John Wiley & Sons Ltd.

Wouters, L., Creff, S., Bella, E. E., & Koudri, A. (2017). Collaborative systems engineering: Issues & challenges. *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 486–491. https://doi.org/10.1109/CSCWD.2017.8066742

Anhang

Inhaltsverzeichnis **Seite**

A1 Gesamtüberblick ISO/IEC 15288:2023-Prozesse

Cat.	ISO ID #	Prozess (Engl.)
AP	6.1.1	Acquisition Process
AP	6.1.2	Supply Process
OPEP	6.2.1	Life Cycle Model Management Process
OPEP	6.2.2	Infrastructure Management Process
OPEP	6.2.3	Project Portfolio Management Process
OPEP	6.2.4	Human Resource Management Process
OPEP	6.2.5	Quality Management Process
OPEP	6.2.6	Knowledge Management Process
TMP	6.3.1	Project Planning Process
TMP	6.3.2	Project Assessment and Control Process
TMP	6.3.3	Decision Management Process
TMP	6.3.4	Risk Management Process
TMP	6.3.5	Configuration Management Process
TMP	6.3.6	Information Management Process
TMP	6.3.7	Measurement Process
TMP	6.3.8	Quality Assurance Process
TP	6.4.1	Business or Mission Analysis Process
TP	6.4.2	Stakeholder Needs and Requirements Definition Process
TP	6.4.3	System Requirements Definition Process
TP	6.4.4	System Architecture Definition Process
TP	6.4.5	Design Definition Process
TP	6.4.6	System Analysis Process
TP	6.4.7	Implementation Process
TP	6.4.8	Integration Process
TP	6.4.9	Verification Process
TP	6.4.10	Transition Process
TP	6.4.11	Validation Process
TP	6.4.12	Operation Process
TP	6.4.13	Maintenance Process
TP	6.4.14	Disposal Process

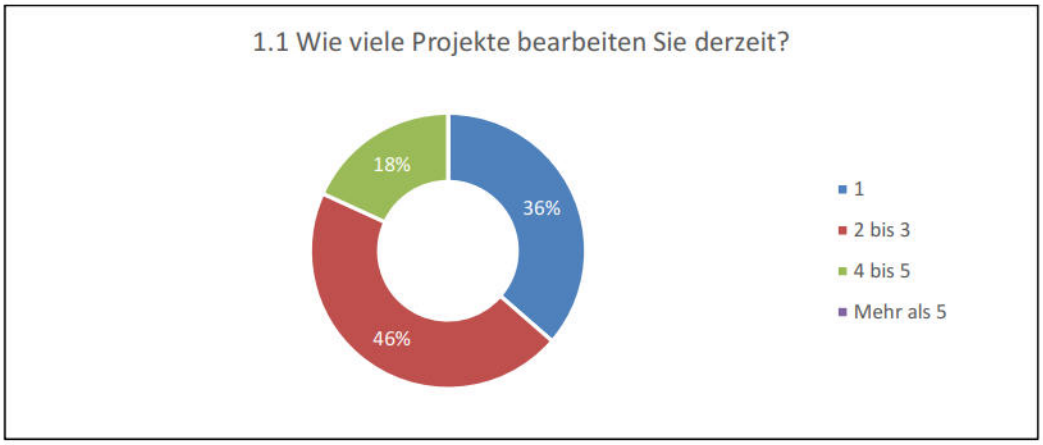
A2 Übersicht verschiedener Git-Befehle

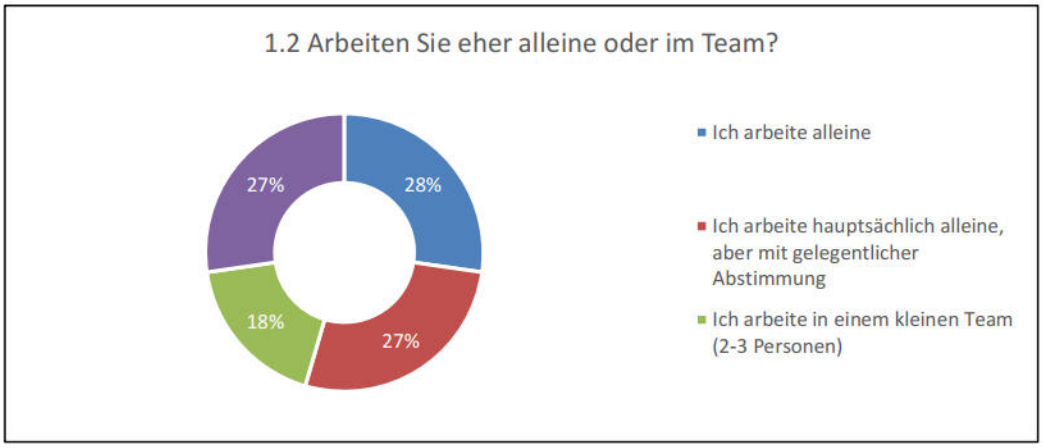
Git-Befehl	Funktionsbeschreibung
git init	Initialisiert ein neues Git-Repository im aktuellen Verzeichnis
´git clone <repo URL>´	Klont ein entferntes Repository lokal
´git status´	Zeigt den aktuellen Status des Repositories / Zeigt den Status der Dateien (staged, unstaged, untracked)
´git add <file>´	Stellt eine bestimmte Datei für den nächsten Commit bereit
´git add .´	Stellt alle Änderungen im aktuellen Verzeichnis bereit
´git commit -m "msg"´	Commit der vorgemerkten Änderungen mit einer Nachricht
´git log´	Zeigt die Historie der Commits an
´git diff´	Zeigt Änderungen zwischen Arbeitsverzeichnis und Index oder Commits
´git branch´	Listet alle lokalen Branches auf
´git branch <name>´	Erstellt einen neuen Branch mit dem angegebenen Namen
´git checkout <branch>´	Wechselt zu einem anderen Branch
´git checkout -b <name>´	Erstellt und wechselt zu einem neuen Branch
´`git merge <branch>´	Führt den angegebenen Branch in den aktuellen zusammen
´git pull´	Holt und integriert Änderungen vom Remote-Repository
´git push´	Überträgt lokale Commits zum Remote-Repository
´git remote -v´	Zeigt die URLs der konfigurierten Remotes an
´git fetch´	Holt Änderungen vom Remote-Repository ohne zu mergen
´git reset <file>´	Entfernt eine Datei aus dem Staging-Bereich
´git reset --hard´	Setzt den Stand auf den letzten Commit zurück (Vorsicht: destruktiv)
´git rm <file>´	Entfernt eine Datei und merkt die Löschung für den nächsten Commit vor
´git stash´	Speichert temporär nicht committete Änderungen

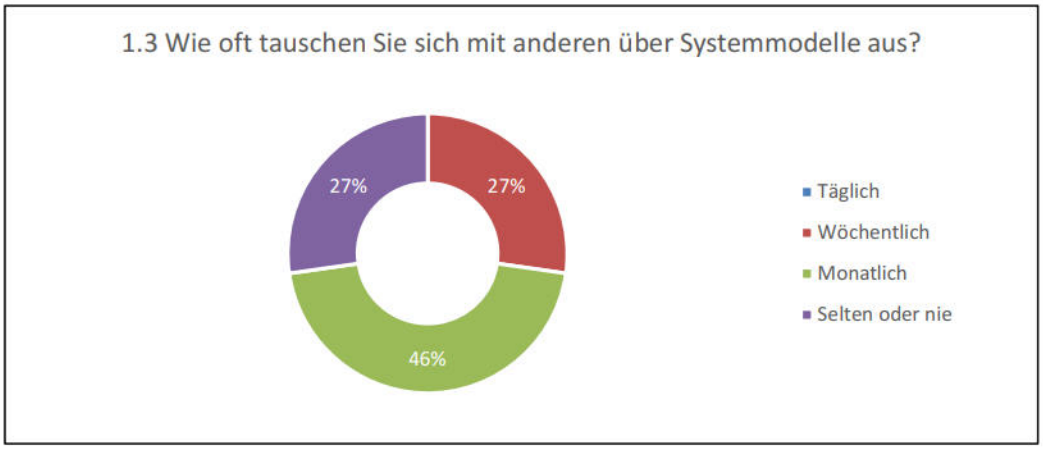
´git stash pop´	Wendet die zuletzt gespeicherten Änderungen wieder an und entfernt sie
´git tag´	Zeigt alle Tags im Repository an
´git tag <name>´	Erstellt einen neuen Tag
´git config´	Zeigt oder ändert die Git-Konfiguration (z. B. Benutzername, E-Mail)

A3 Ergebnisse der Teamumfrage

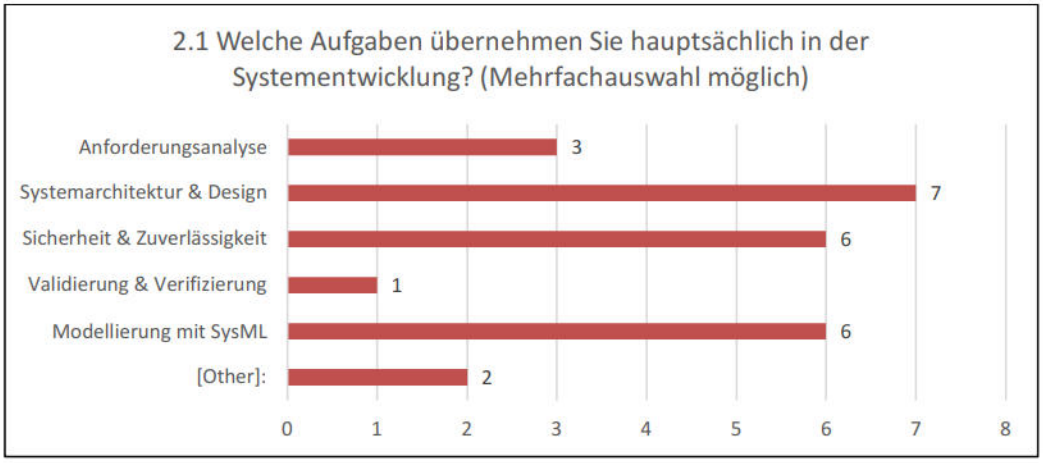
1. Team- und Projektinformationen



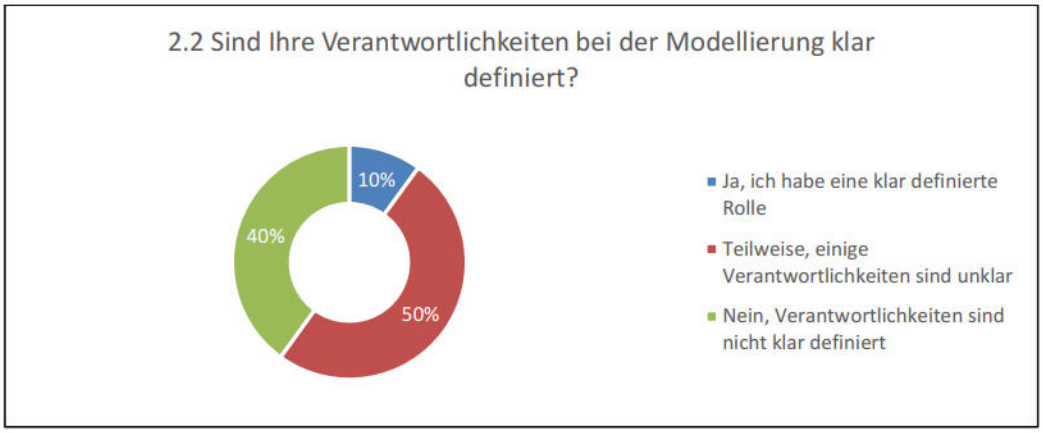


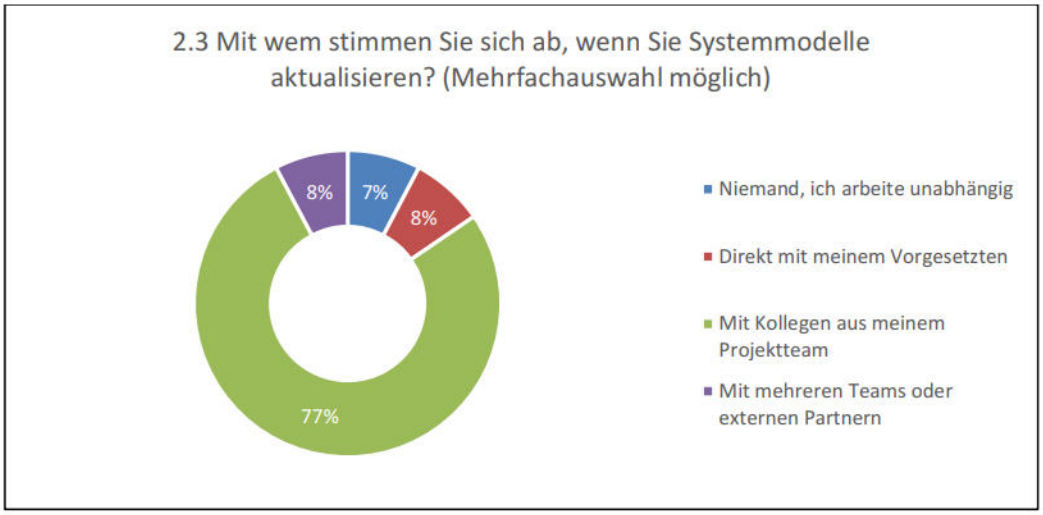


2. Rollen und Verantwortlichkeiten in der Systementwicklung

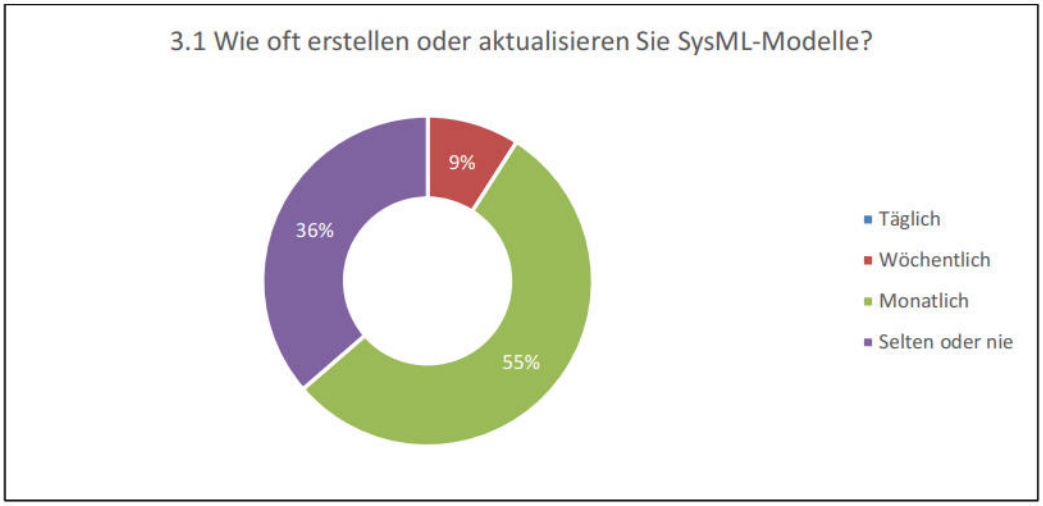


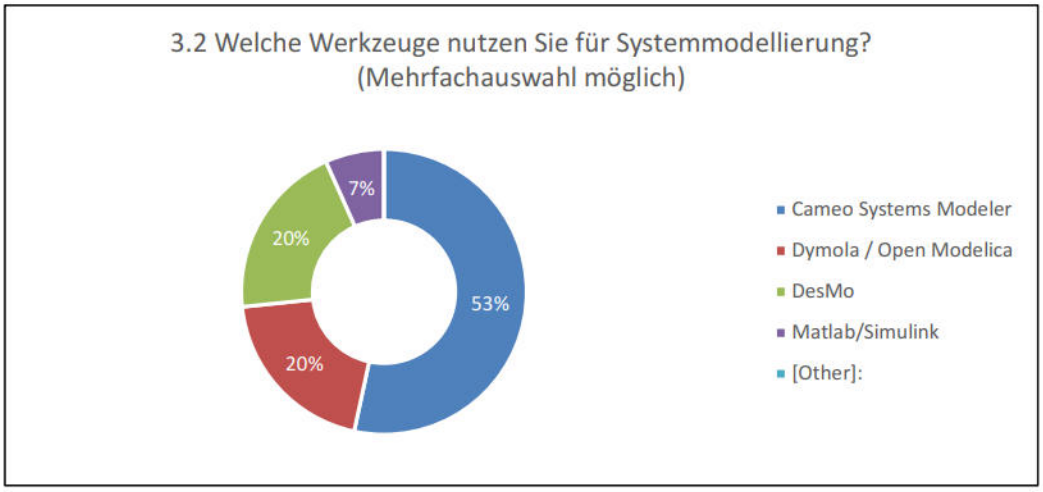
Other: Klappen-system Design und Integration + Entwurfssprache/Ontologien

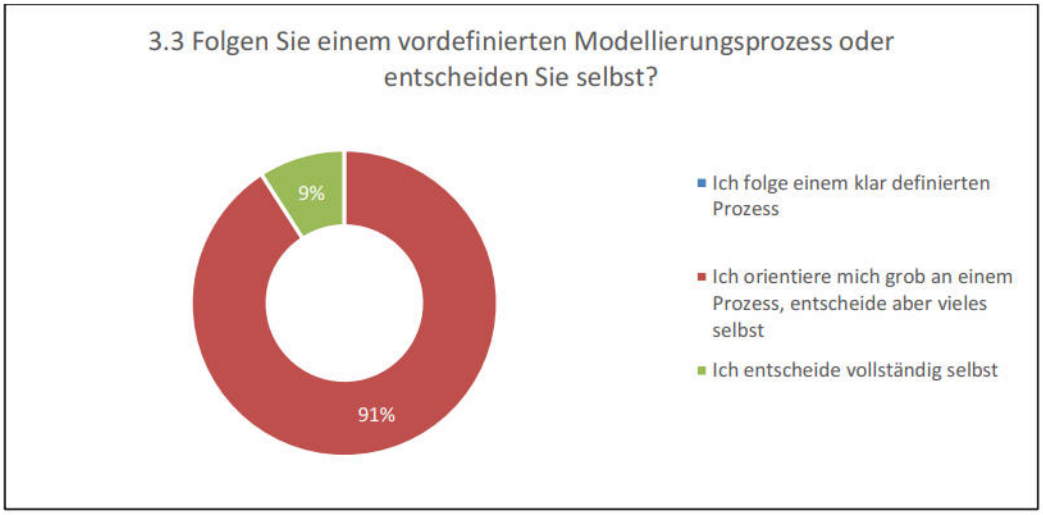


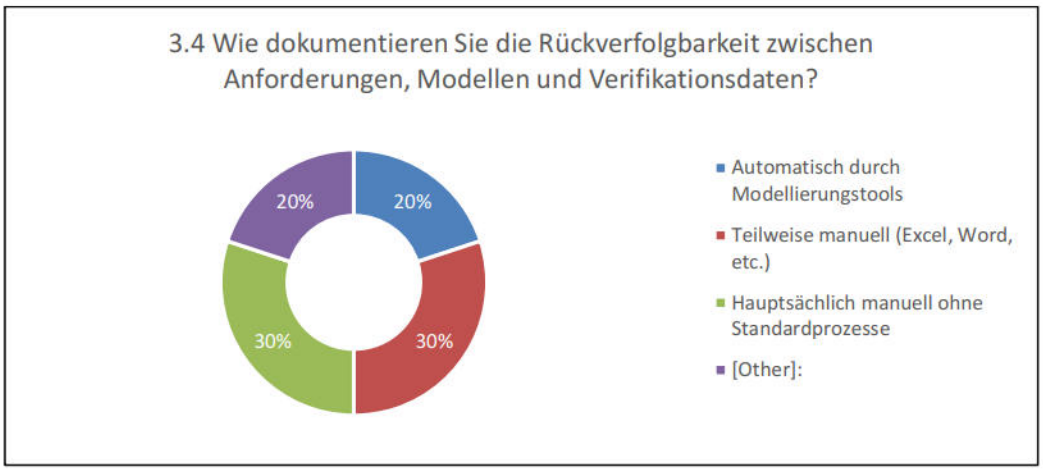


3. Modellierungspraxis und SysML-Nutzung



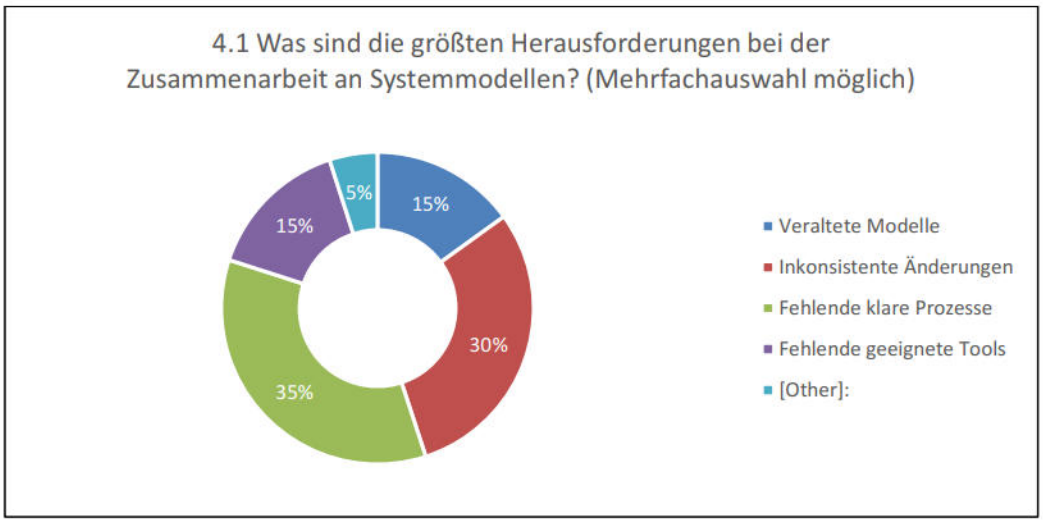




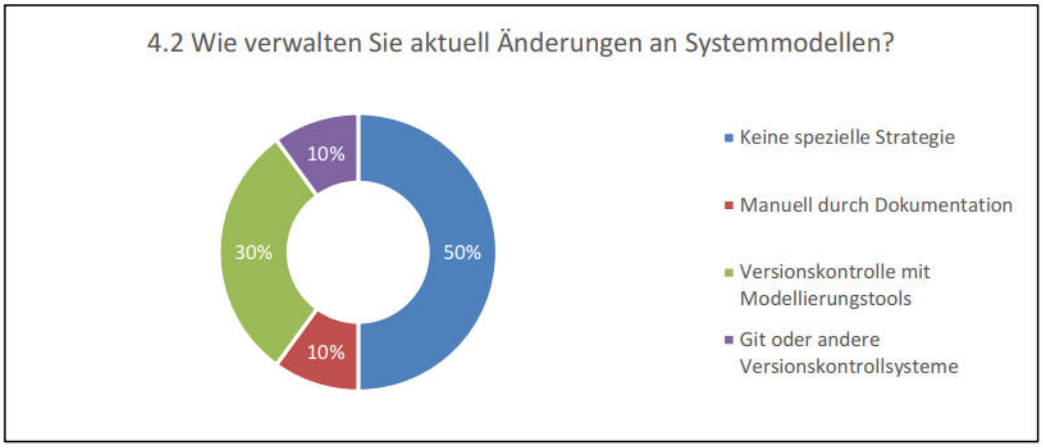


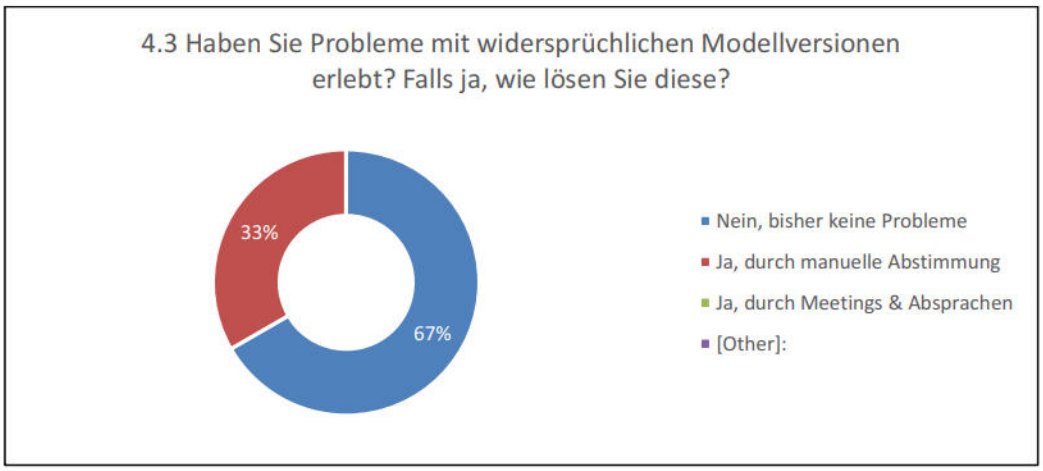
Other: unautomatisch durch Modellierungstools + Durch das manuelle Hinzufügen von Satisfy-, Derive-, refine-, Verify-, Trace-Verbindungselementen zwischen Anforderungen, Modellelementen, etc.

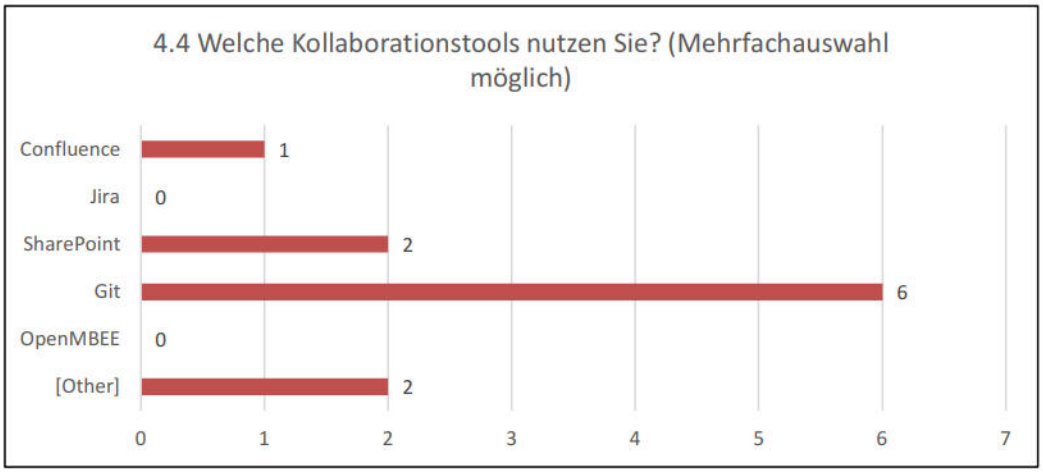
4. Kollaboration und Versionskontrolle



Other: Zeitlicher Mehraufwand im Forschungsbereich

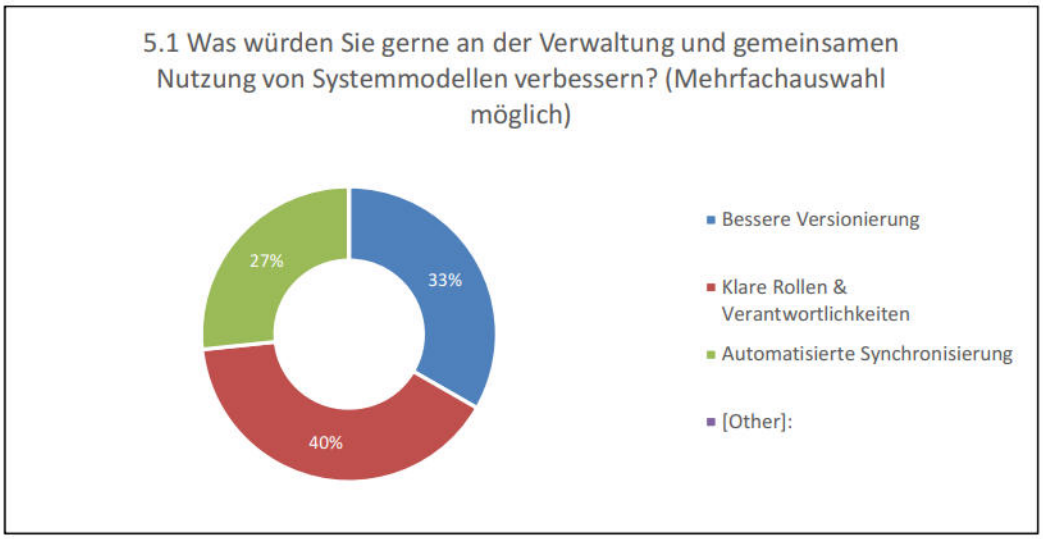


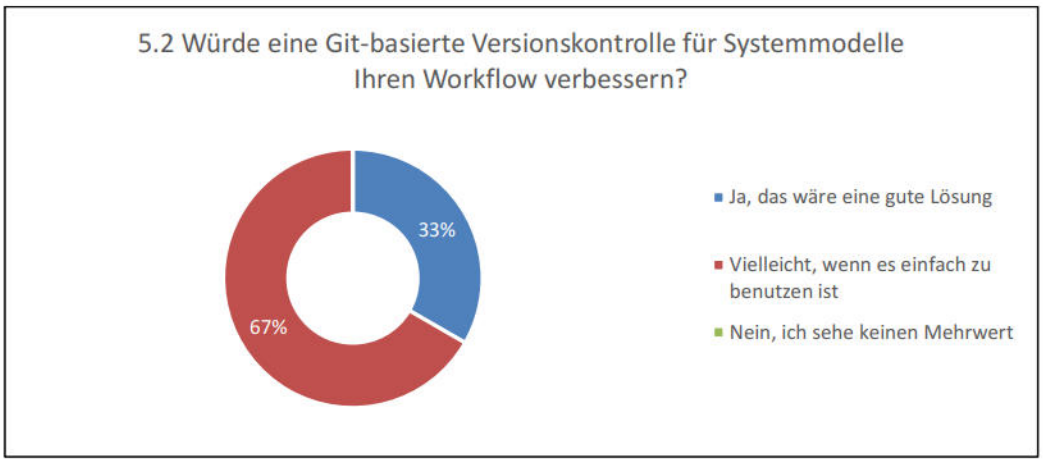


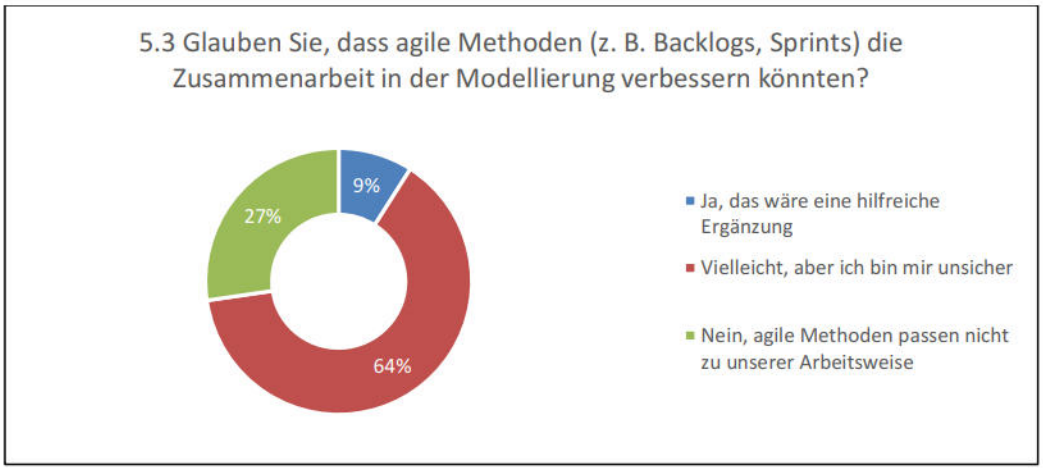


Other: Mattermost, GitLab- Git Instanz + Für Cameo bislang die Teamwork-Cloud

5. Erwartungen und Verbesserungsideen







Antwort #	**5.4 Kennen Sie weitere Lösungen zur Versionskontrolle oder Synchronisation von Systemmodellen neben Git? Falls ja, welche und welche Erfahrungen haben Sie damit gemacht?**
1	Team Work Cloud
2	TWC (Hauseigene "Cloud" für Cameo Systems Modeler. Versionierung eher rudimentär, fehlende Funktionalität. Okay zum sharen von modellen, medium zum kollaborieren in meiner Erfahrung)
3	Cameo (Team Work Cloud), Interessant wäre eine Bearbeitung von Modellen im Stil von Google Docs, die das parallele Arbeiten an Modellen ermöglicht und vereinfacht.
4	Neben Git hat die Verwendung der Teamworkcloud für die Versionskontrolle von SysML-Modellen gut funktioniert. Diese habe ich bislang verwendet. Mit dem Update von Cameo von 2021 aufwärts bin ich mir aber nicht sicher, ob die Teamworkcloud noch genutzt werden kann. Hierzu müsste man sich nochmal mit Malte Rahm absprechen.

A4 Übersicht der SysML-Kernel-Kommandos in Jupyter Notebook

Befehl	**Funktionsbeschreibung**
%eval	Eine gegebene Expression auswerten
%export	Eine Datei mit der JSON-Repräsentation des abstrakten Syntaxbaums eines benannten Elements speichern
%help	Eine Liste verfügbarer Befehle anzeigen oder Hilfe zu einem bestimmten Befehl aufrufen
%list	Geladene Bibliothekspakete oder die Ergebnisse einer Abfrage auflisten
%show	Den abstrakten Syntaxbaum eines benannten Elements ausgeben
%publish	Die Modellelemente, die in einem benannten Element verwurzelt sind, in das *Repository* veröffentlichen
%view	Die durch die benannte *View Usage* definierte Ansicht rendern
%viz	Die benannten Modellelemente visualisieren

A5 Screenshots zu Testszenario #1 – Export und Commit-Validierung

CSM mit SysML v2-Plugin:

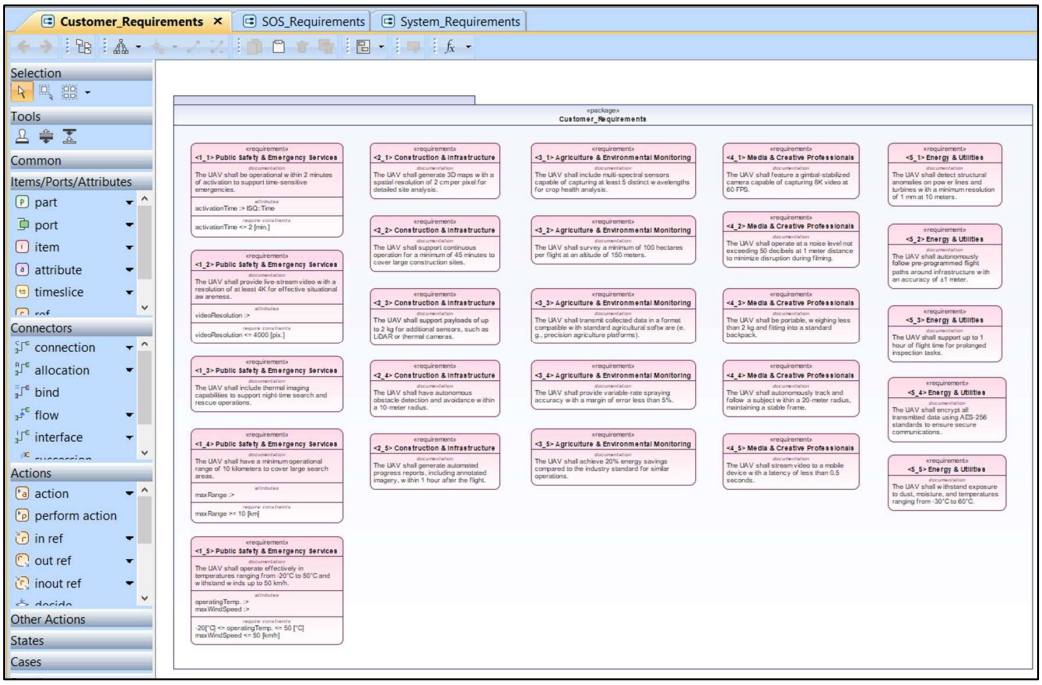


Bild A5-1 Customer Requirements

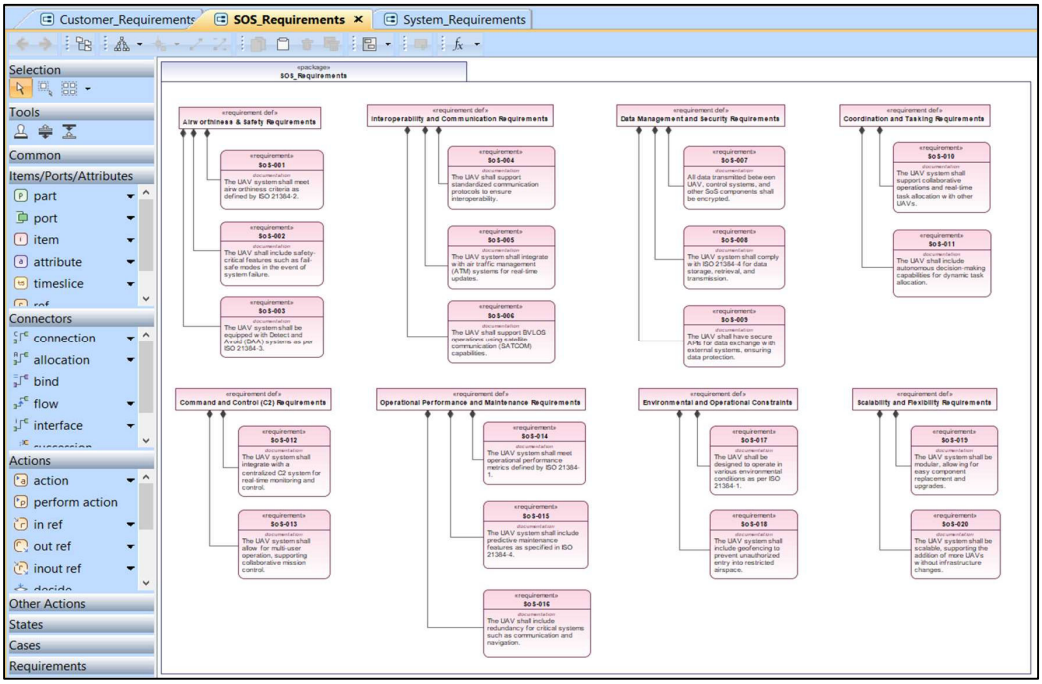


Bild A5-2 SOS Requirements

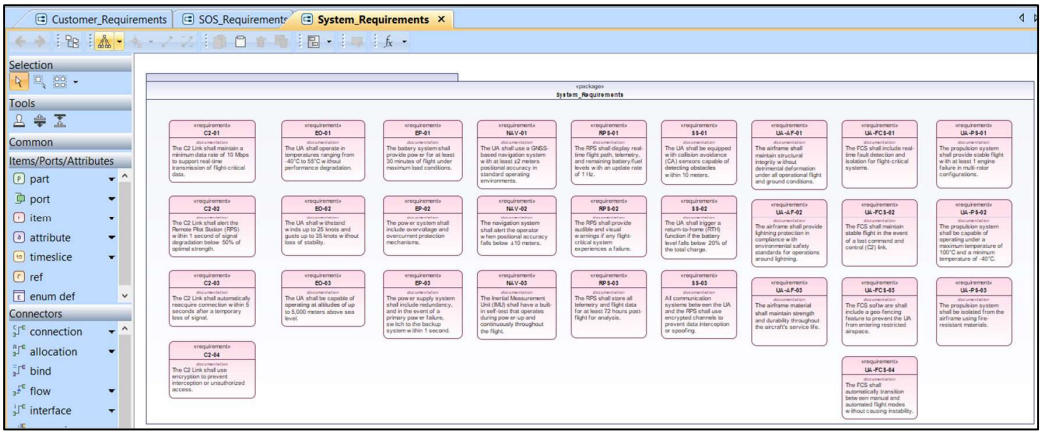


Bild A5-3 System Requirements

Git CLI:

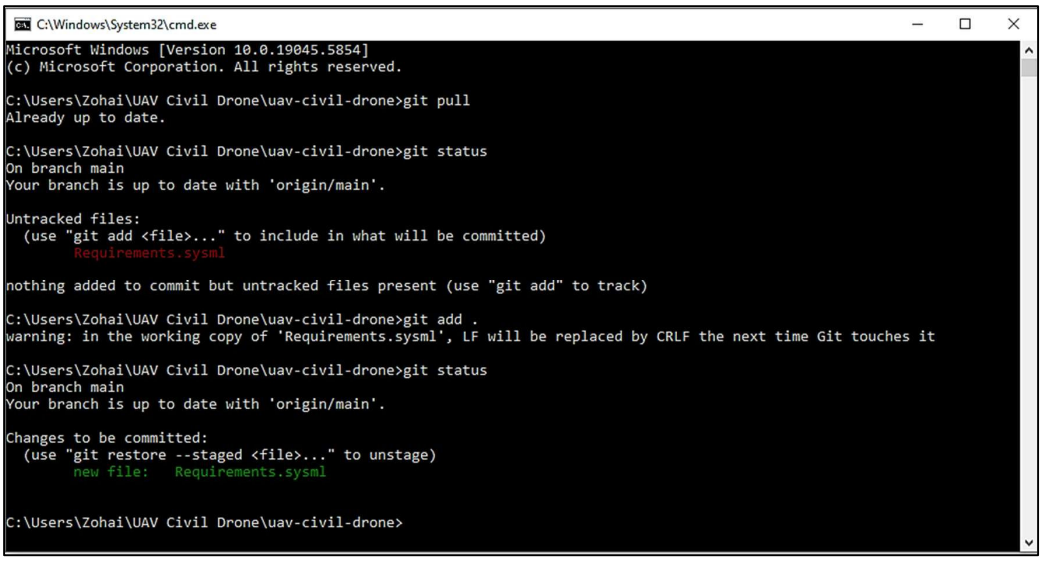


Bild A5-4 Git CLI (1)

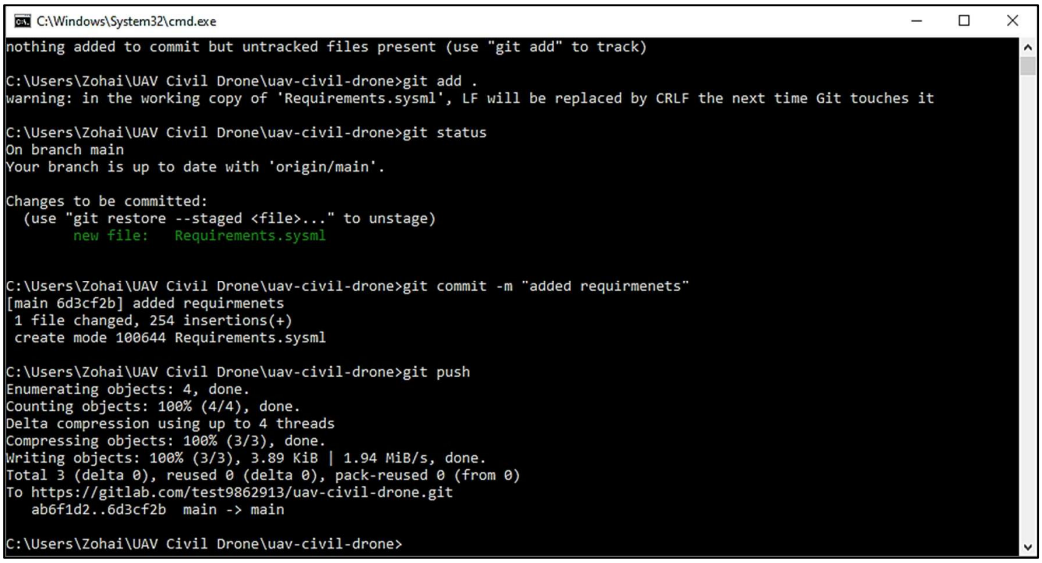


Bild A5-5 Git CLI (2)

GitLab UI:

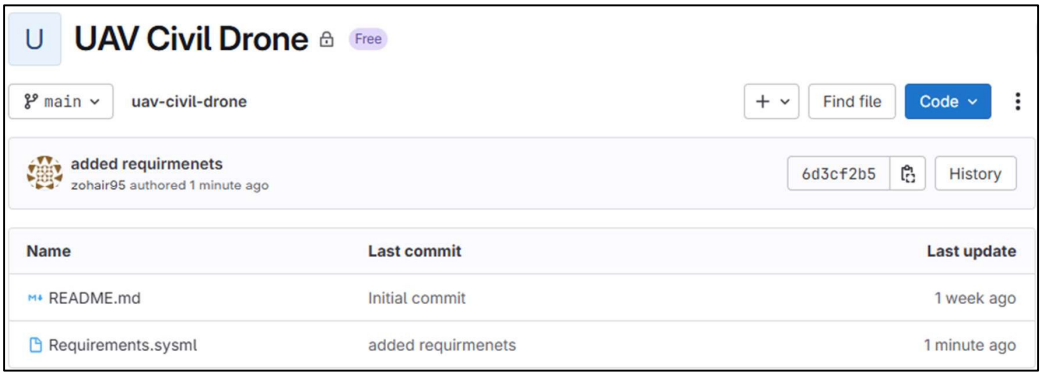


Bild A5-6 GitLab UI nach Testabschluss

A6 Screenshots zu Testszenario #2 – Anzeige und Bearbeitung im JN

JN SysML-Kernel:

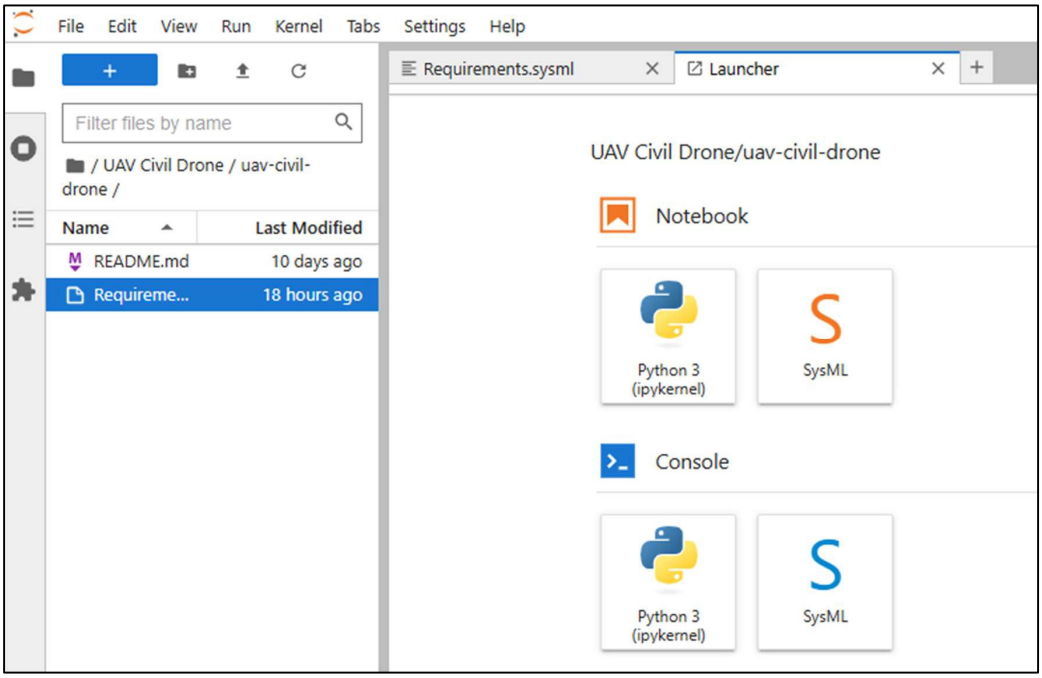


Bild A6-1 SysML-Kernel in JN

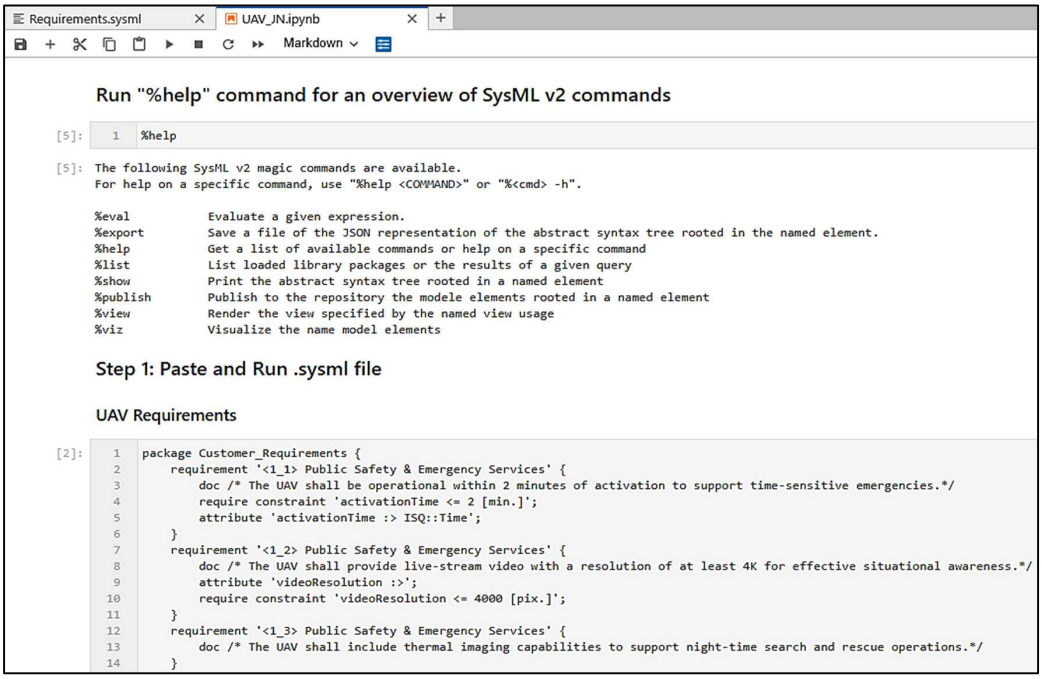


Bild A6-2 UAV Requirements in JN (1)

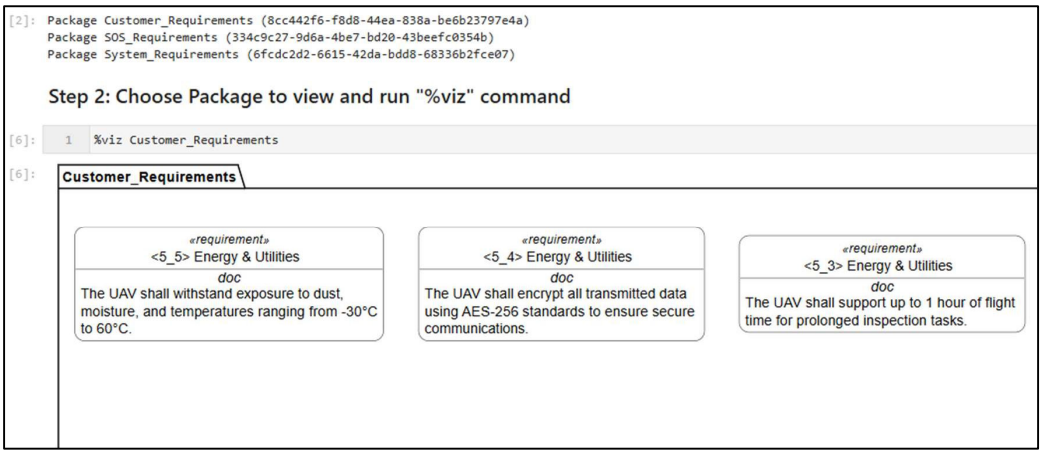


Bild A6-3 UAV Requirements in JN (2)

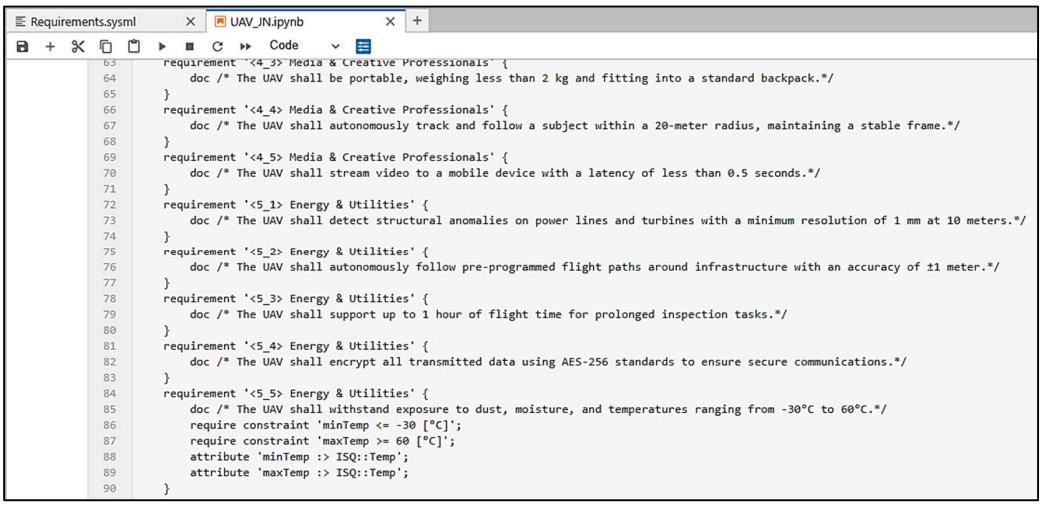


Bild A6-4 Manuelle Eingaabe der Constraints und Attribute für „requirement <5_5> (Zeile 86-89)“

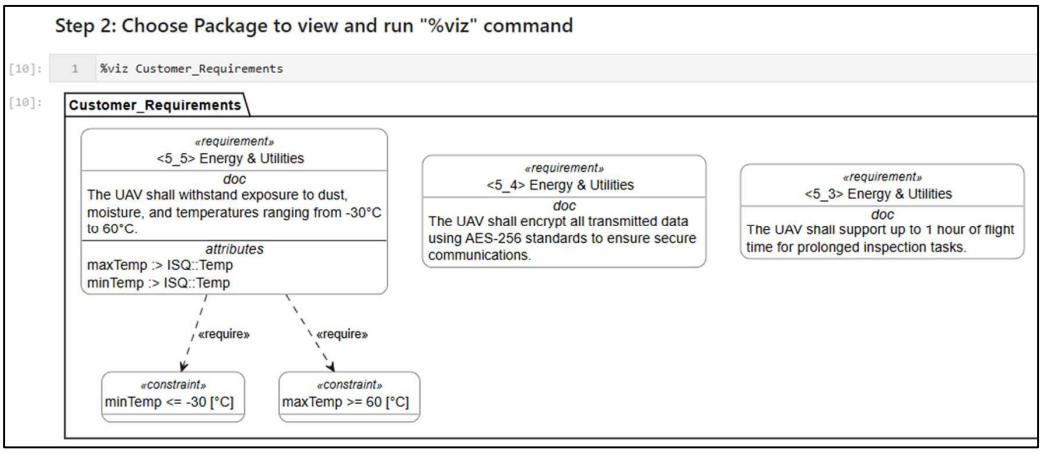


Bild A6-5 Ergebnis nach der manuellen Anpassung von „requirement <5_5>“

Git CLI:

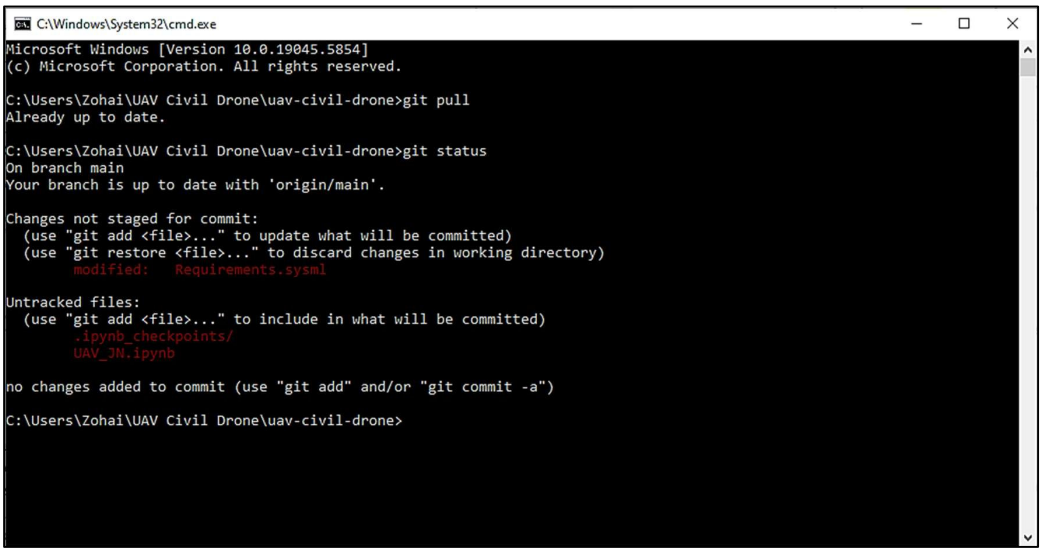


Bild A6-6 Git CLI (1)

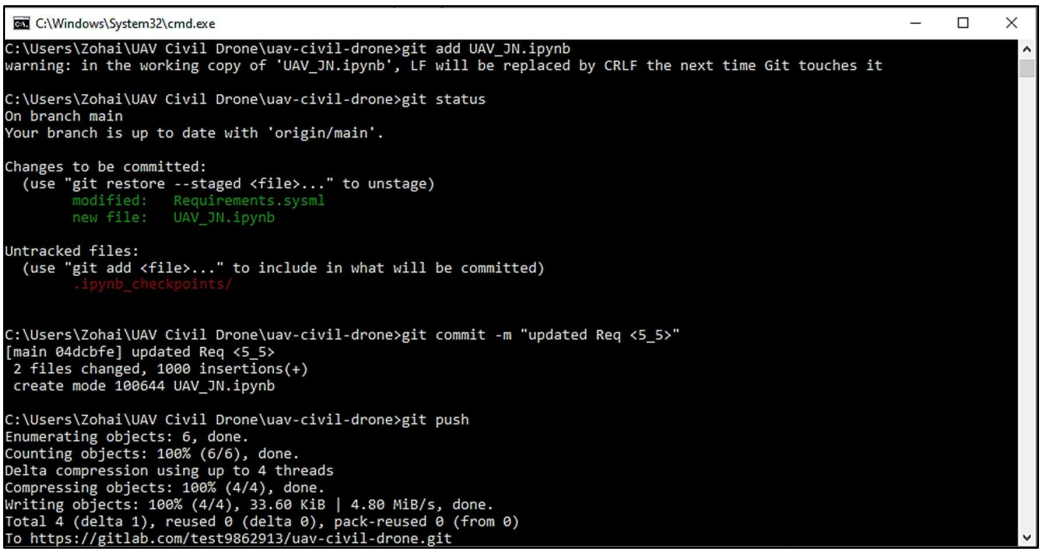


Bild A6-7 Git CLI (2)

GitLab UI:

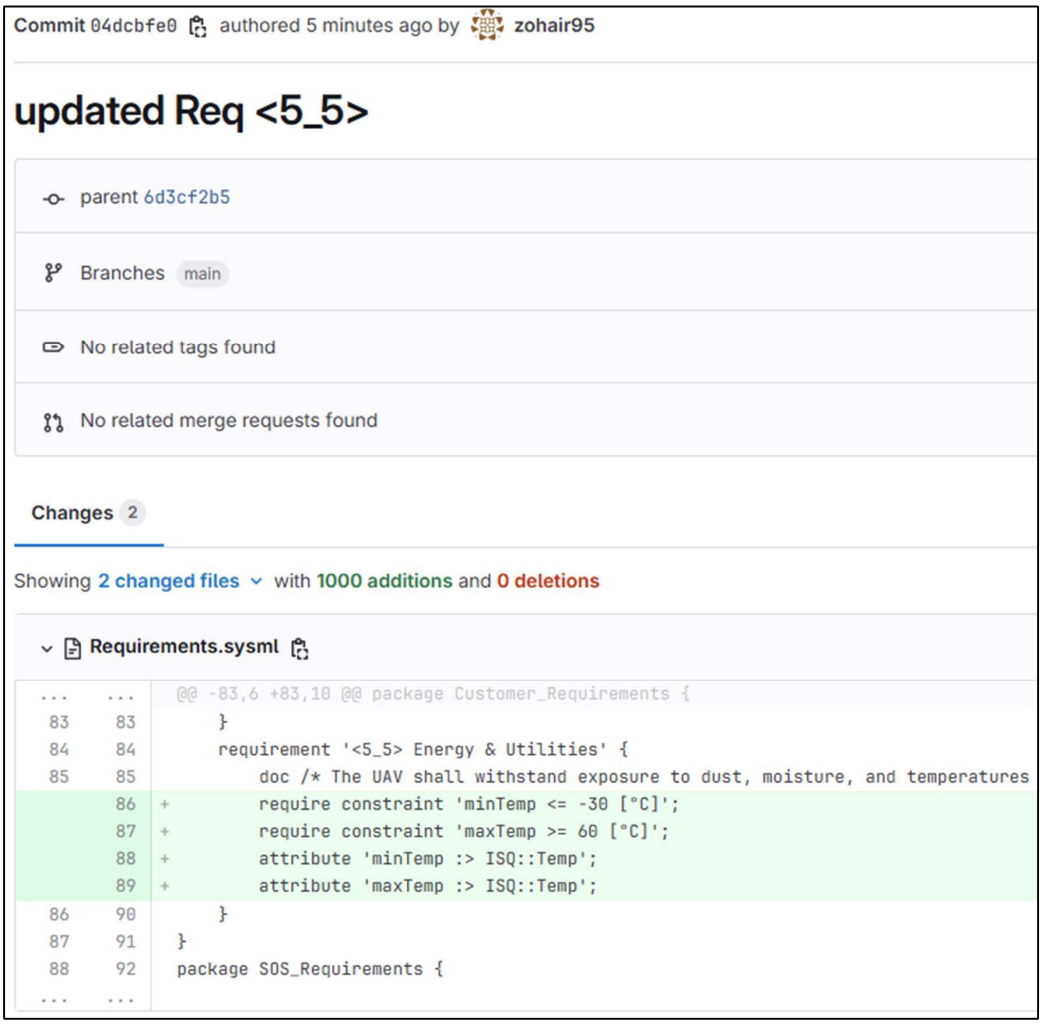


Bild A6-8 Versionierung der Änderungen von „requirement <5_5>“ in GitLab

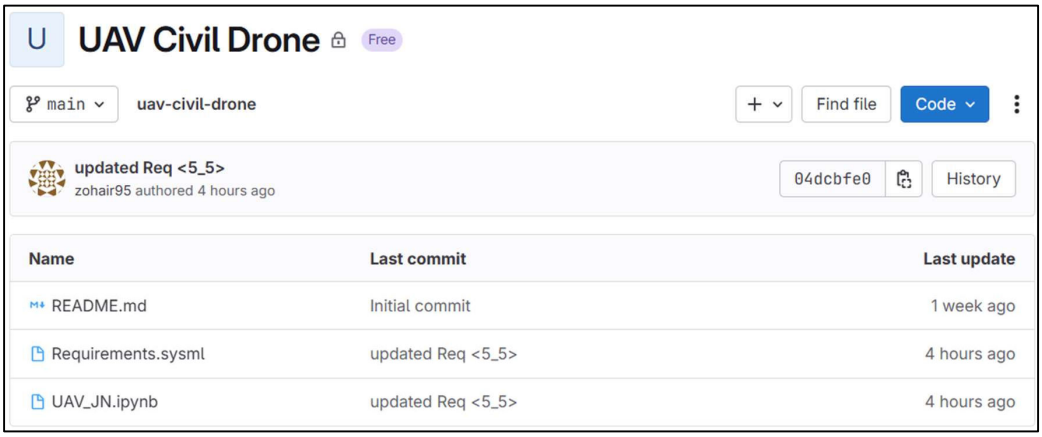


Bild A6-9 GitLab UI nach Testabschluss

A7 Screenshots zu Testszenario #3 – Multi-Tool-Kompatibilität (CSM und JN)

CSM mit SysML v2-Plugin:

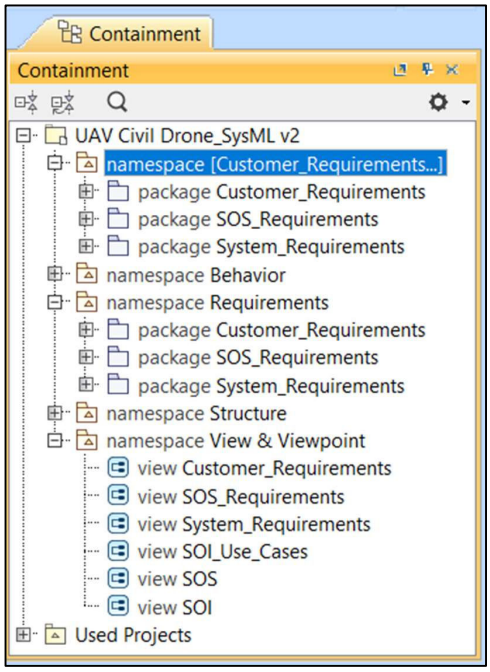


Bild A7-1 Containment-Baum nach import der „Requirements.sysml“-Datei

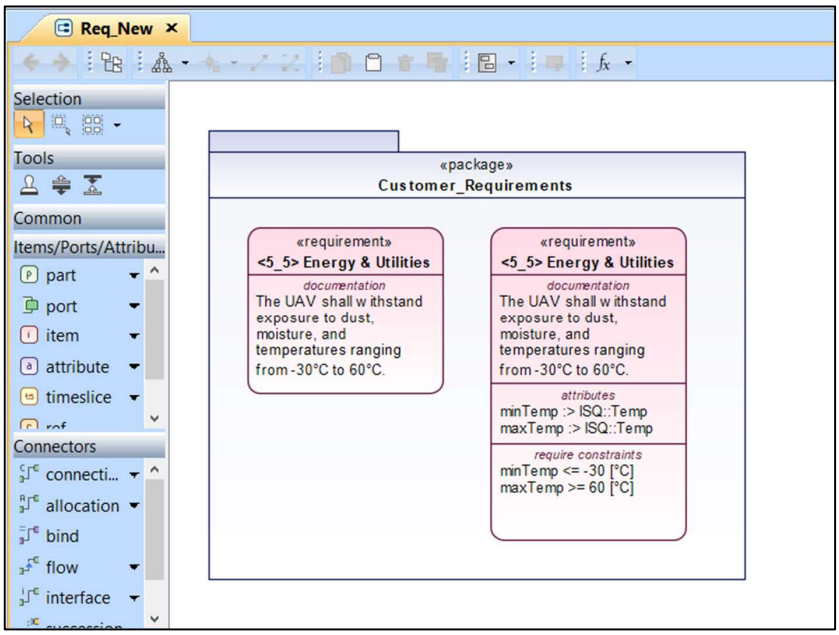


Bild A7-2 Vergleich der ursprünglichen (Links) und modifizierten (Rechts) Anforderung „Requirement <5_5>“ im View-Diagramm „Req_New“

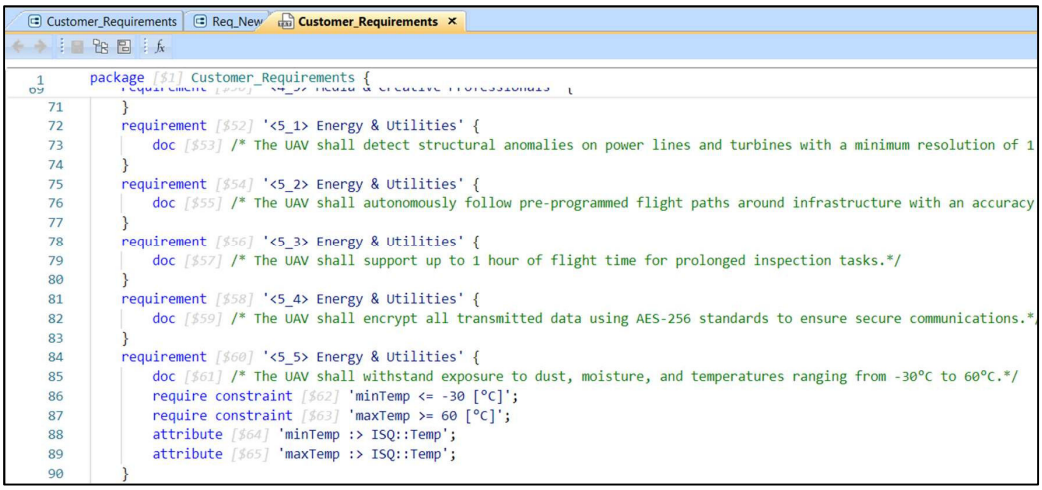


Bild A7-3 Importierte „Requirements.sysml“-Datei als textuelle Notation in CSM

A8 Screenshots zu Testszenario #4 – Versionierung und Rollback

CSM mit SysML v2-Plugin:

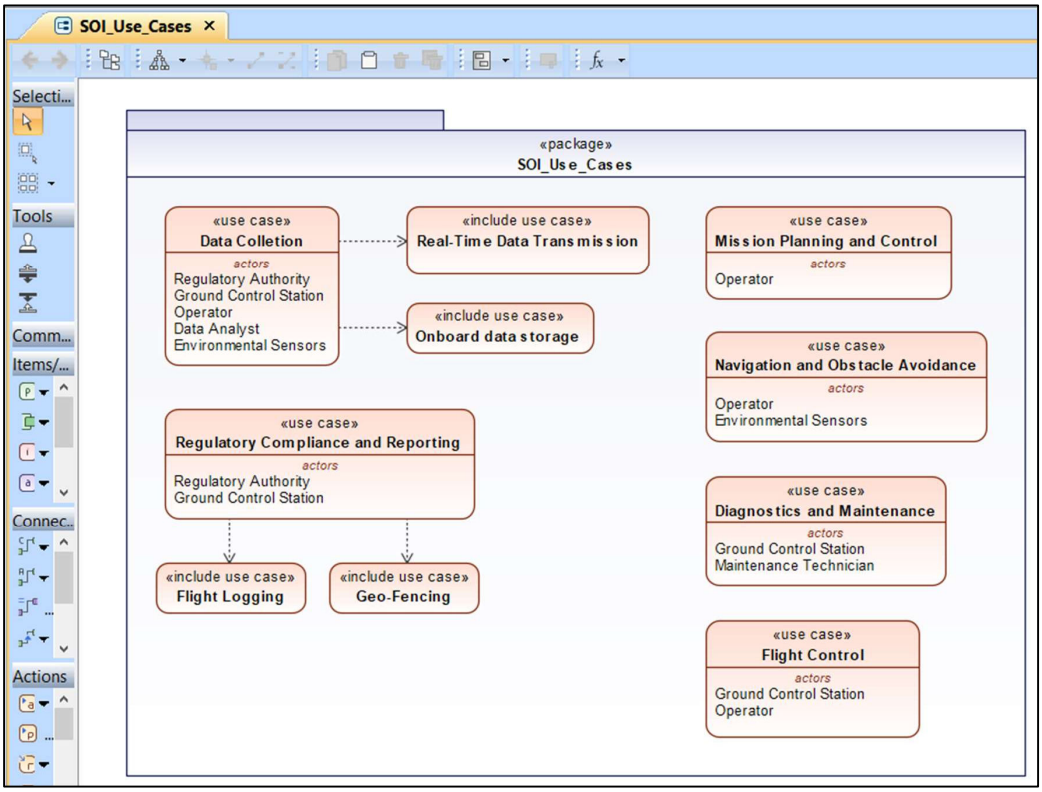


Bild A8-1 SOI Use Cases in Behavior.sysml

Git CLI:

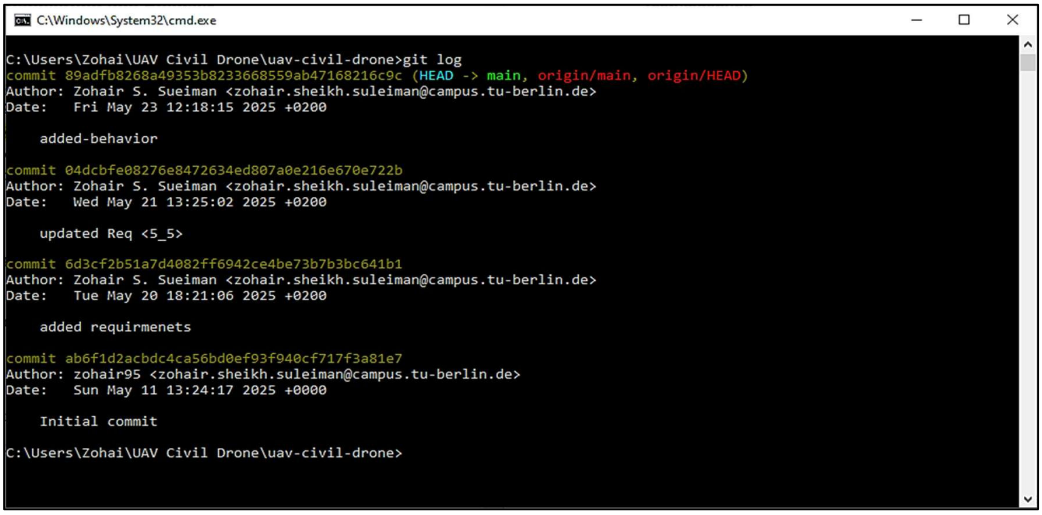


Bild A8-2 `git log`-Befehl

GitLab UI:

Bild A8-3 Commit-Verlauf nach Testabschluss

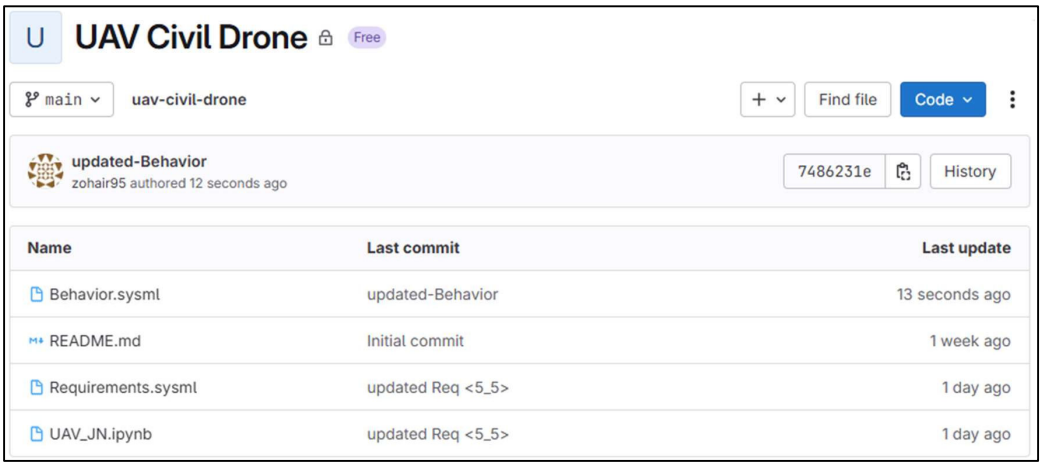


Bild A8-4 GitLab UI vor ´git revert´

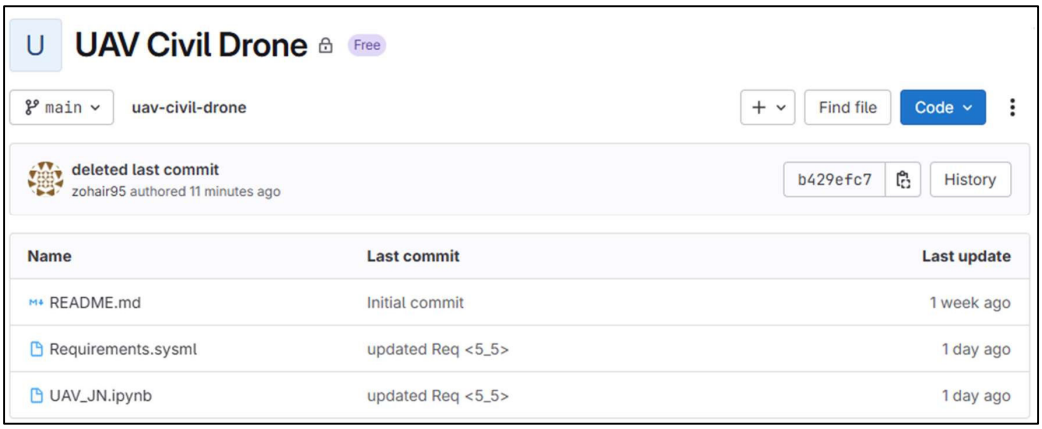


Bild A8-5 GitLab UI nach ´git revert´

A9 Screenshots zu Testszenario #5 – GitHub Flow Test (Kollaborationsworkflow)

GitLab UI:

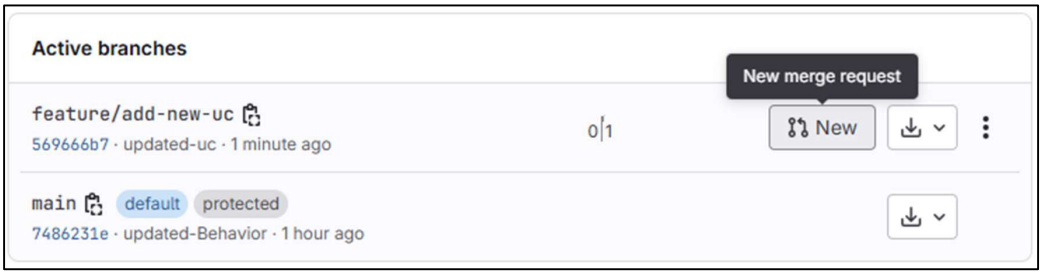


Bild A9-1 Branches-Übersicht in GitLab Repository

Bild A9-2 GitLab Merge Request (1)

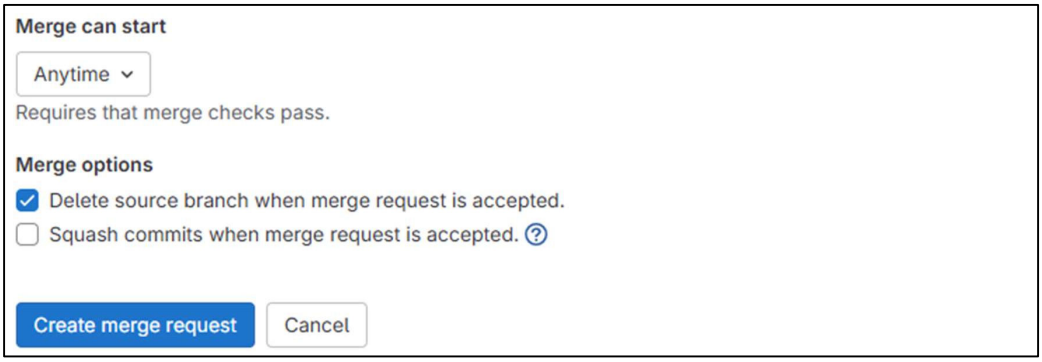


Bild A9-3 GitLab Merge Request (2)

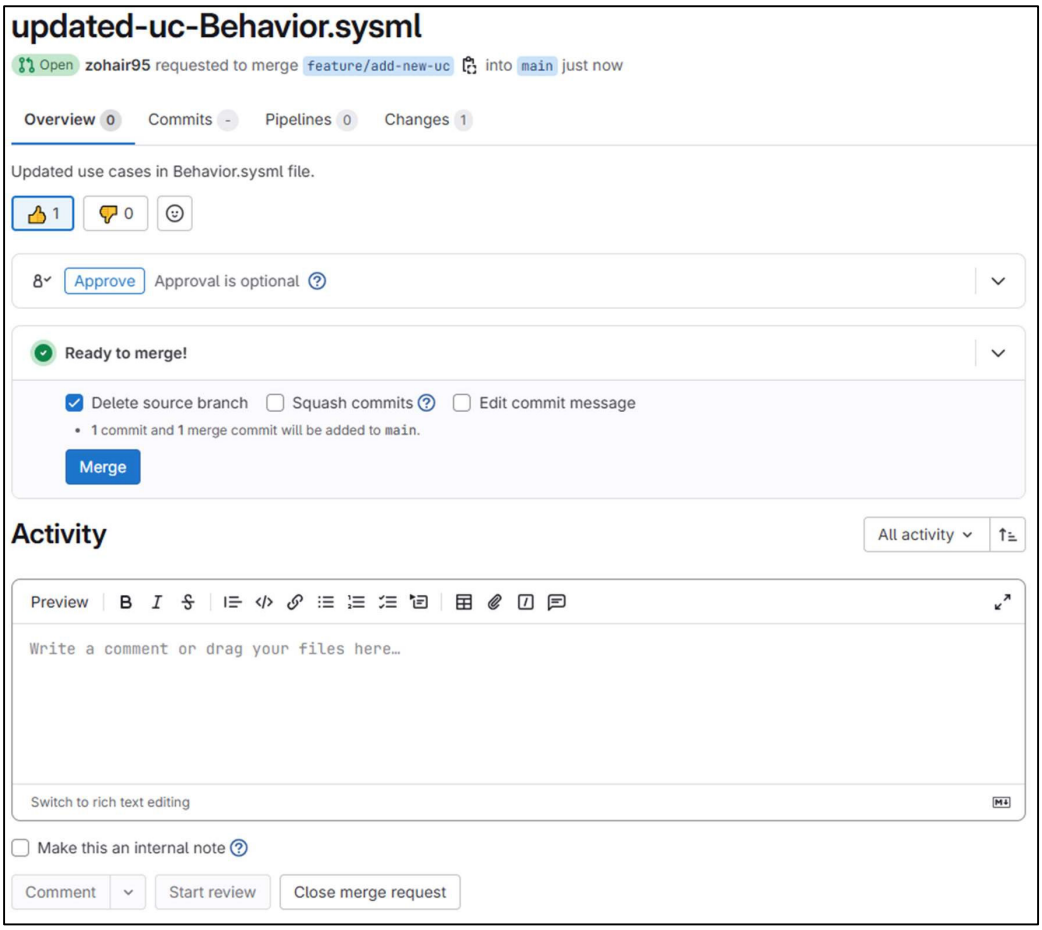


Bild A9-4 GitLab Merge Request (3)

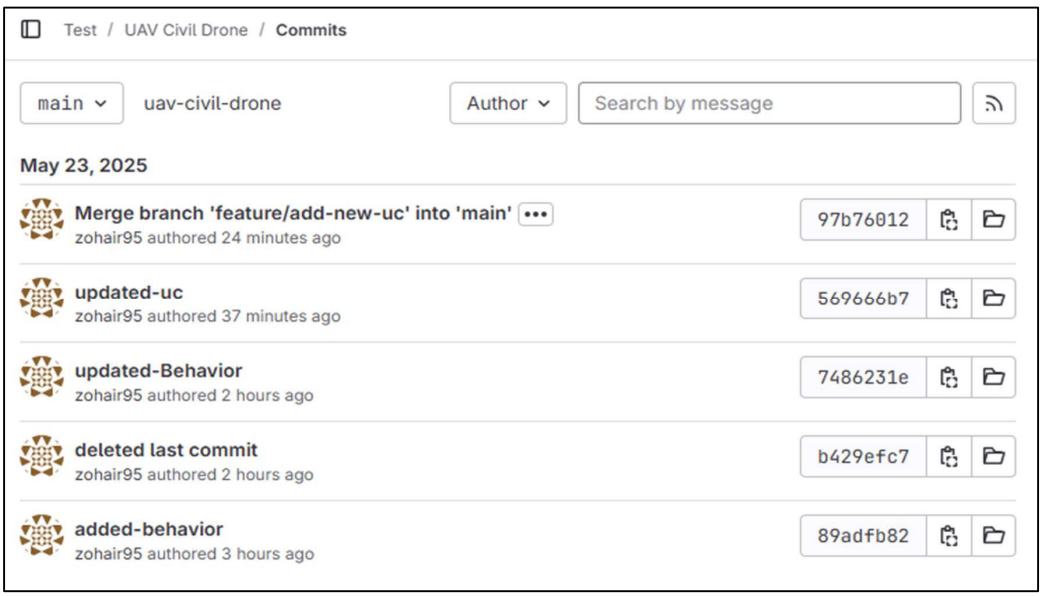


Bild A9-5 Commit-Verlauf nach Testabschluss

A10 Screenshots und Python-Skript zu Testszenario #6 – Automatisierte Konfigurationsprüfung

CSM mit SysML v2-Plugin:

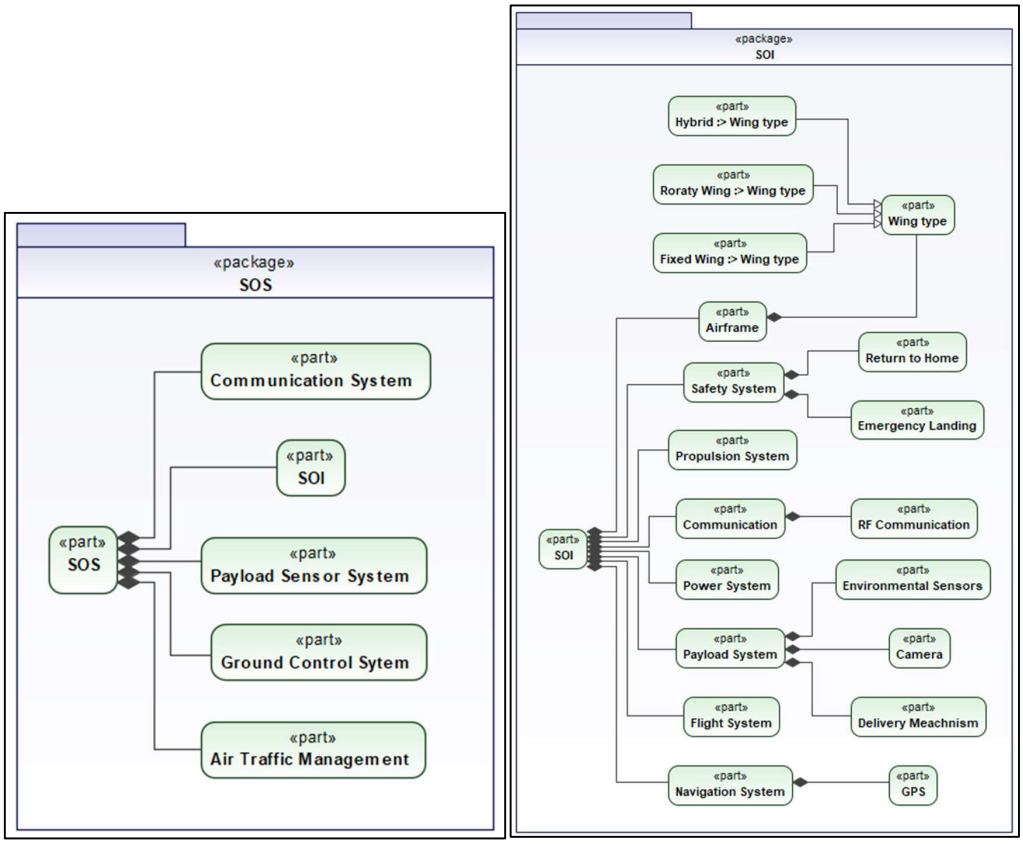


Bild A10-1 SOS und SOI aus Structure.sysml in CSM mit SysML v2-Plugin

JN mit Python 3-Kernel:

```
Scanning 'Behavior.sysml'...
No issues found.

Scanning 'bracket_issue.sysml'...
 Line 2: unknown term 'Unknown term: 'nospace''
 → requirement nospace;
 Line 4: unknown term 'Unknown term: 'nospace''
 → part nospace;

Scanning 'Requirements.sysml'...
No issues found.

Scanning 'Structure.sysml'...
No issues found.
```

Bild A10-2 Ergebnisse nach lokaler Ausführung von fix_config.ipynb in JN

GitLab CI/CD mit *.gitlab-ci.yml*:

```yaml
check_config:
  stage: check-config
  image: python:3.10                    # Leichtgewichtiges Python-Image
  before_script:
    - pip install jupyter               # Jupyter wird für die Konvertierung benötigt
  script:
    - jupyter nbconvert --to script fix_config.ipynb     # Notebook in .py-Datei konvertieren
    - python fix_config.py                               # Korrigiertes Skript ausführen
  only:
    changes:
      - System_Models/*.sysml           # Trigger nur bei Änderungen an SysML-Dateien
```

Bild A10-3 Ausschnitt aus .gitlab-ci.yml

GitLab UI:

Bild A10-4 Screenshot nach erfolgreicher CI-Pipeline Funktion

Python-Skript *fix_config.ipynb*:

Python-Skript `**fix_config.ipynb**` zur automatischen Formatierung der SysML-Dateien

```
import os

import re

# Pfad zum Verzeichnis, das die zu überprüfenden SysML-Modelldateien enthält

FOLDER_PATH = "./System_Models"

# Gültige Schlüsselwörter (SysML-spezifisch), die ohne Anführungszeichen verwendet werden dürfen

VALID_TERMS = {

    'part', 'block', 'requirement', 'constraint', 'package', 'import', 'diagram', 'property', 'flow', 'port', 'interface', 'association', 'value', 'type', 'connector', 'relationship', 'actor', 'signal', 'state', 'transition', 'event', 'operation', 'input', 'output', 'use', 'case', 'subject', 'include', 'first', 'then', 'require', 'attribute', 'def', 'doc', 'view', 'private', 'in', 'out', 'library', 'item', 'action', 'inout', 'allocation', 'ref', 'end', 'analysis', 'objective', 'loop', 'assert', 'calc', 'stakeholder', 'assume', 'concern', 'connection', 'do', 'exit', 'enum', 'occurrence', 'time', 'timeslice', 'exhibit', 'metadata', 'perform', 'assign', 'self', 'send', 'null', 'if', 'while', 'true', 'false',
```

```
'join', 'done', 'start', 'fork', 'merge', 'decide', 'verification',
'to', 'accept', 'satisfy', 'connect', 'bind', 'frame', 'for', 'i',
'ISQ', 'dependency', 'comment', 'locale', 'rep', 'language',
'about', 'alias', 'abstract', 'variation', 'variant', 'subsets',
'redefines', 'specializes', 'references', 'ordered', 'nonunique',
'individual', 'snapshot', 'message', 'succession', 'via', 'from',
'entry', 'parallel', 'verify', 'filter', 'render', 'expo-
se','viewpoint'

}

# Regulärer Ausdruck zum Erkennen korrekt gesetzter Anführungszei-
chen

QUOTE_REGEX = re.compile(r"'[^']*'")

def quote_identifier_if_needed(token):

    # Prüft, ob ein Begriff in Anführungszeichen gesetzt werden muss

    if token in VALID_TERMS or QUOTE_REGEX.fullmatch(token):

        return token

    return f"'{token}'"

def fix_colon_arrow_line(line): # Korrigiert fehlerhafte Zeilen mit
dem Operator ':>', wobei Einrückungen beibehalten und doppelte Kor-
rekturen vermieden werden.

    if ':>' not in line or ';' not in line:

        return line  # Zeilen ohne ':>' werden übersprungen

    # Einrückung ermitteln

    indent_match = re.match(r"^(\s*)", line)

    indentation = indent_match.group(1) if indent_match else ""

    # Regulärer Ausdruck für korrekte Syntax

    valid_term_pattern = '|'.join(re.escape(term) for term in
VALID_TERMS)

    correct_pattern = rf"^\s*({valid_term_pat-
tern})\s+'[^']+'\s+:>\s+'[^']+'(?:::'[^']+')*;\s*$"

    if re.match(correct_pattern, line.strip(), re.IGNORECASE):

        return line  # Format bereits korrekt

    # Struktur extrahieren und neu zusammensetzen

    match = re.match(r"^\s*(\w+)\s+([^:]+)\s*:>\s*([^;]+);",
line.strip())

    if not match:

        return line  # Zeile passt nicht zum erwarteten Muster

    keyword = match.group(1).strip()

    lhs = match.group(2).strip().replace("'", "")
```

```
    rhs = match.group(3).strip().replace("'", "")
    lhs_quoted = f"'{lhs}'"
    rhs_quoted = "::".join(f"'{seg.strip()}'" for seg in
rhs.split("::"))
    return f"{indentation}{keyword} {lhs_quoted} :> {rhs_quoted};\n"
def fix_line(line):
    # Hauptfunktion zur Zeilenkorrektur
    stripped = line.strip()
    if stripped.startswith("doc /*") or "doc /*" in stripped:
        return line  # Kommentare bleiben unverändert
    # Ignoriere bestimmte "first ... then ...;" Konstrukte
    if stripped.startswith("first '") and " then " in stripped and
stripped.endswith("';"):
        return line
    # Versuche zuerst die ':>'-Struktur zu korrigieren
    fixed = fix_colon_arrow_line(line)
    if fixed != line:
        return fixed
    # Korrektur von falsch geschriebenen 'include use case'
    line = re.sub(r"include\s+'use'\s+case", "include use case",
line)
    # Quoting von Namespace-Elementen (z.B. package::element)
    def replace_namespaced(match):
        parts = match.group().split("::")
        return "::".join(quote_identifier_if_needed(p) for p in
parts)
    line = re.sub(r"\b(?:[A-Za-z_][\w\-]*::)+[A-Za-z_][\w\-]*\b",
replace_namespaced, line)
    # Setzt Anführungszeichen für unbekannte Begriffe nach bekannten
Schlüsselwörtern
    def keyword_replacer(match):
        keyword, token = match.group(1), match.group(2)
        if token in VALID_TERMS or QUOTE_REGEX.fullmatch(token):
            return f"{keyword} {token}"
        return f"{keyword} '{token}'"
```

```
    pattern = r"\b(" + "|".join(sorted(VALID_TERMS, key=len, re-
verse=True)) + r")\s+([^\s';{}\[\]:]+)"

    line = re.sub(pattern, keyword_replacer, line)

    # Verschachtelte Anführungszeichen korrigieren

    line = re.sub(r"'([^']*)'([^']*)'([^']*)'", lambda m:
f"'{m.group(1)}{m.group(2)}{m.group(3)}'", line)

    return line

def process_file(file_path):

    # Führt die Zeilenkorrektur für eine einzelne Datei durch

    with open(file_path, "r", encoding="utf-8") as f:

        lines = f.readlines()

    fixed_lines = []

    changed = False

    fix_count = 0

    for line in lines:

        fixed_line = fix_line(line)

        fixed_lines.append(fixed_line)

        if fixed_line != line:

            changed = True

            fix_count += 1

    if changed:

        # Überschreibt die Datei mit den korrigierten Zeilen

        with open(file_path, "w", encoding="utf-8") as f:

            f.writelines(fixed_lines)

        print(f"Fixed {fix_count} issues in: {os.path.base-
name(file_path)}")

    else:

        print(f"No changes in: {os.path.basename(file_path)}")

def run_fix():

    # Durchläuft das Modellverzeichnis und verarbeitet alle .sysml-
Dateien

    for filename in os.listdir(FOLDER_PATH):

        if filename.endswith(".sysml"):

            process_file(os.path.join(FOLDER_PATH, filename))

run_fix()# Start der automatisierten Korrektur
```

A11 Screenshots und Python-Skript zu Testszenario #7 – Automatisierte Syntaxprüfung

JN mit Python 3-Kernel:

```
Scanning 'Behavior.sysml'...
No issues found.

Scanning 'Customer_Requirements.sysml'...
No issues found.

Scanning 'SoS_Requirements.sysml'...
No issues found.

Scanning 'Structure.sysml'...
 Line 4: unknown term 'Unknown term: 'patt''
→ patt 'UAV Civil Drone';
 Line 13: unknown term 'Unknown term: 'past''
→ past 'Payload System : EO';
```

Bild A11-1 Konsolenausgabe des Syntaxprüfskripts

GitLab CI/CD mit *.gitlab-ci.yml*:

```yaml
syntax_check:
  stage: syntax-check
  image: python:3.10
  before_script:
    - pip install jupyter
  script:
    - jupyter nbconvert --to script syntax_check.ipynb   # Notebook in ausführbares Skript umwandeln
    - python syntax_check.py                             # Syntaxprüfung starten
  only:
    changes:
      - System_Models/*.sysml
```

Bild A11-2 Ausschnitt aus .gitlab-ci.yml

```
5.1 traitlets-5.14.3 types-python-dateutil-2.9.0.20250516 typing_extensions-4.13.2 uri-template-1.3.0 u
rllib3-2.4.0 wcwidth-0.2.13 webcolors-24.11.1 webencodings-0.5.1 websocket-client-1.8.0 widgetsnbextens
ion-4.0.14
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with
the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.i
o/warnings/venv
[notice] A new release of pip is available: 23.0.1 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
$ jupyter nbconvert --to script syntax_check.ipynb
[NbConvertApp] Converting notebook syntax_check.ipynb to script
[NbConvertApp] Writing 4334 bytes to syntax_check.py
$ python syntax_check.py
Scanning 'Structure_v2.sysml'...
No issues found.
Scanning 'Behavior.sysml'...
No issues found.
Scanning 'Requirements.sysml'...
No issues found.
Scanning 'Structure.sysml'...
No issues found.
Cleaning up project directory and file based variables                    00:01
Job succeeded
```

Bild A11-3 Ausschnitt Pipeline-Skript aus GitLab CI

Python-Skript *syntax_check.ipynb*:

Python-Skript `**syntax_check.ipynb**` zur automatischen Qualitätsprüfung der SysML-Dateien

```
import os

import re

import sys

# Definierte Liste aller gültigen Begriffe in SysML v2 (Referenz für Syntaxprüfung)

VALID_TERMS = {

    'part', 'block', 'requirement', 'constraint', 'package', 'import',    'diagram', 'property', 'flow', 'port', 'interface', 'association',    'value', 'type', 'connector', 'relationship', 'actor', 'signal',    'state', 'transition', 'event', 'operation', 'input', 'output', 'use',    'case', 'subject', 'include', 'first', 'then', 'require', 'attribute', 'def', 'doc', 'view', 'private', 'in', 'out', 'library', 'item', 'action',    'inout', 'allocation', 'ref', 'end', 'analysis', 'objective', 'loop',    'assert', 'calc', 'stakeholder', 'assume', 'concern', 'connection', 'do', 'exit', 'enum', 'occurrence', 'time', 'timeslice', 'exhibit', 'metadata', 'perform', 'assign', 'self', 'send', 'null', 'if', 'while', 'true','false',    'join', 'done', 'start', 'fork', 'merge', 'decide', 'verification', 'to', 'accept', 'satisfy', 'connect', 'bind', 'frame', 'for', 'i', 'ISQ', 'dependency', 'comment', 'locale', 'rep', 'language', 'about', 'alias', 'abstract', 'variation', 'variant', 'subsets', 'redefines', 'specializes', 'references', 'ordered', 'nonunique', 'individual', 'snapshot', 'message', 'succession', 'via', 'from', 'entry', 'parallel', 'verify', 'filter', 'render', 'expose', 'viewpoint'

}

# Funktion zum Entfernen von Text innerhalb von doc-Kommentarblöcken oder Anführungszeichen

def remove_ignored_sections(line, in_doc_block):

    if 'doc /*' in line:

        in_doc_block = True

    if in_doc_block:

        if '*/' in line:

            in_doc_block = False

        return '', in_doc_block  # Inhalt wird ignoriert

    line = re.sub(r"'[^']*'", '', line)  # Inhalte in einfachen Anführungszeichen werden entfernt

    return line, in_doc_block
```

```
# Funktion zur Identifikation unbekannter oder falsch geschriebener Begriffe
def find_unknown_terms(file_path, valid_terms):
    unknown_terms = []
    in_doc_block = False
    with open(file_path, 'r') as f:
        lines = f.readlines()
        for i, line in enumerate(lines, 1):
            clean_line, in_doc_block = remove_ignored_sections(line, in_doc_block)
            if not clean_line.strip():
                continue
            # Zerlege Zeile in einzelne Wörter
            words = re.findall(r'\b\w+\b', clean_line)
            for word in words:
                word_lower = word.lower()
                if word_lower in valid_terms:
                    # Groß-/Kleinschreibungsfehler erkennen
                    if word != word_lower:
                        unknown_terms.append((i, f"Capitalization error: '{word}' should be lowercase", line.strip()))
                elif word_lower.isalpha():
                    # Unbekannter Begriff
                    unknown_terms.append((i, f"Unknown term: '{word}'", line.strip()))
    return unknown_terms
# Funktion zur Überprüfung auf Klammernfehler und fehlende Semikolons
def check_braces_and_semicolons(file_path):
    issues = []
    brace_stack = []  # Stack zur Überprüfung von geschachtelten Klammern
    in_doc_block = False

    with open(file_path, 'r') as f:
        lines = f.readlines()
```

```
        for i, line in enumerate(lines, 1):
            raw_line = line.strip()
            clean_line, in_doc_block = remove_ignored_sections(line,
in_doc_block)
            if not clean_line.strip():
                continue
            # Überprüfung auf geschlossene/geöffnete Klammern
            for char in clean_line:
                if char == '{':
                    brace_stack.append((i, '{'))
                elif char == '}':
                    if brace_stack:
                        brace_stack.pop()
                    else:
                        issues.append((i, "Unmatched closing brace
'}'", raw_line))
            # Überprüfung auf fehlende Semikolons bei bestimmten
Schlüsselwörtern
            if re.match(r'\s*(actor|subject|include)\b', clean_line,
re.IGNORECASE):
                if not clean_line.strip().endswith(';'):
                    issues.append((i, "Missing semicolon",
raw_line))
        # Noch offene geschweifte Klammern melden
        for brace in brace_stack:
            issues.append((brace[0], "Unmatched opening brace '{'",
lines[brace[0]-1].strip()))
    return issues
# Flag zur Kennzeichnung, ob ein Fehler gefunden wurde
any_issues_found = False
# Hauptfunktion zur Durchsuchung aller .sysml-Dateien im angegebenen
Verzeichnis
def scan_sysml_files(folder):
    global any_issues_found  # Ermöglicht globale Änderung des Feh-
ler-Flags
    for filename in os.listdir(folder):
        if filename.endswith(".sysml"):
```

```
            file_path = os.path.join(folder, filename)
            print(f"\nScanning '{filename}'...")
            # Durchführung der Prüfungen
            unknown_term_issues = find_unknown_terms(file_path,
VALID_TERMS)
            syntax_issues = check_braces_and_semicolons(file_path)
            # Auswertung der Ergebnisse
            if unknown_term_issues or syntax_issues:
                any_issues_found = True  # Fehler gefunden
                for line_num, word, line in unknown_term_issues:
                    print(f"Line {line_num}: unknown term '{word}'")
                    print(f"    → {line}")
                for line_num, msg, line in syntax_issues:
                    print(f"Line {line_num}: {msg}")
                    print(f"    → {line}")
            else:
                print("No issues found.")
# Ordner mit den SysML-Modellen
FOLDER_PATH = "./System_Models"
# Start der Analyse
scan_sysml_files(FOLDER_PATH)
# Abbruch der Pipeline mit Fehlercode, falls Probleme gefunden wur-
den
if any_issues_found:
    sys.exit(1)
```

A12 Screenshots und Python-Skript zu Testszenario #8 – Automatisierte Dokumentenerstellung

GitLab CI/CD mit .gitlab-ci.yml:

```
# === GitLab CI/CD-Pipeline für SysML-Modelle ===
# Diese Pipeline besteht aus drei Stufen:
# 1. Konfigurationsprüfung
# 2. Syntaxprüfung
# 3. Berichtserzeugung in Jupyter Notebook

stages:
  - check-config       # Stufe 1: Prüfung und Korrektur von Konfigurationsfehlern
  - syntax-check       # Stufe 2: Prüfung der SysML-Syntax
  - generate-report    # Stufe 3: Automatische Erstellung eines Jupyter-Berichts
```

Bild A12-1 Ausschnitt aus .gitlab-ci.yml (1)

```
generate_report:
  stage: generate-report
  image: jupyter/scipy-notebook          # Jupyter-Umgebung mit vorinstallierten Paketen (z. B. Pandas, Matplotlib)
  before_script:
    - pip install nbformat               # Paket zur Bearbeitung von .ipynb-Dateien
  script:
    - jupyter nbconvert --to notebook --execute generate_render_notebook.ipynb  # Ausführung des Notebooks
  artifacts:
    paths:
      - SysML_Reports/sysml_report_v*.ipynb  # Exportiertes Notebook als Artefakt bereitstellen
  only:
    changes:
      - System_Models/*.sysml
```

Bild A12-2 Ausschnitt aus .gitlab-ci.yml (2)

GitLab UI:

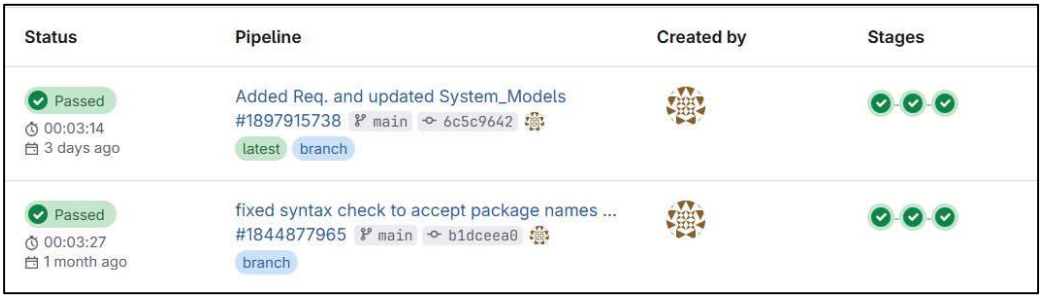
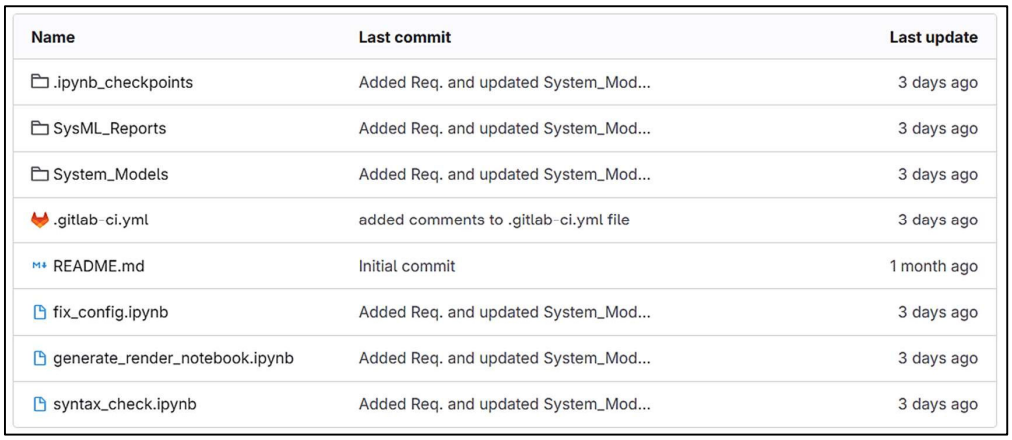


Bild A12-3 Die drei Pipelinestufen nach erfolgreichem CI-Durchlauf



Name	Last commit	Last update
.ipynb_checkpoints	Added Req. and updated System_Mod...	3 days ago
SysML_Reports	Added Req. and updated System_Mod...	3 days ago
System_Models	Added Req. and updated System_Mod...	3 days ago
.gitlab-ci.yml	added comments to .gitlab-ci.yml file	3 days ago
README.md	Initial commit	1 month ago
fix_config.ipynb	Added Req. and updated System_Mod...	3 days ago
generate_render_notebook.ipynb	Added Req. and updated System_Mod...	3 days ago
syntax_check.ipynb	Added Req. and updated System_Mod...	3 days ago

Bild A12-4 Aufbau des Repositories nach der Testdurchführung

Python-Skript *generate_render_notebook.ipynb*:

Python-Skript `**generate_render_notebook.ipynb**` zur automatischen Berichterstellung der SysML-Dateien in Jupyter Notebook

```
import os

import re

import nbformat

import hashlib

import json

from nbformat.v4 import new_notebook, new_markdown_cell,
new_code_cell

# === KONFIGURATION ===

model_dir = "System_Models"         # Eingabeverzeichnis mit .sysml-
Modellen

output_dir = "SysML_Reports"        # Ausgabeverzeichnis für gene-
rierte Notebooks

os.makedirs(output_dir, exist_ok=True)  # Erstelle das Ausgabever-
zeichnis, falls nicht vorhanden

version_file = os.path.join(output_dir, "version_metadata.json")  #
Datei mit Versionsinformationen

base_output = os.path.join(output_dir, "sysml_report")  # Basisname
für generierte Reports

# Funktion zur Erzeugung eines Hash-Werts für den Inhalt einer Datei

def hash_file_content(content):

    return hashlib.sha256(content.encode("utf-8")).hexdigest()

# === VORHERIGE VERSION LADEN (falls vorhanden) ===
```

```
if os.path.exists(version_file):
    with open(version_file, "r") as f:
        prev_state = json.load(f)
    prev_version = prev_state["version"]
    prev_files = prev_state["files"]
else:
    prev_version = "v1.0.0"      # Initiale Version, wenn keine vorhanden
    prev_files = {}
# === AKTUELLE .sysml-DATEIEN EINLESEN UND HASHEN ===
current_files = {}               # Datei-Hashs zur Änderungserkennung
sysml_models = []                # Gesammelte Inhalte aller SysML-Dateien
for filename in os.listdir(model_dir):
    if filename.endswith(".sysml"):
        path = os.path.join(model_dir, filename)
        with open(path, "r", encoding="utf-8") as f:
            content = f.read()
        sysml_models.append(content)
        current_files[filename] = hash_file_content(content)
# === ÄNDERUNGEN ZWISCHEN VERSIONEN DETEKTIEREN ===
added = set(current_files) - set(prev_files)           # Neue Dateien
deleted = set(prev_files) - set(current_files)         # Gelöschte Dateien
modified = {f for f in current_files if f in prev_files and current_files[f] != prev_files[f]}  # Geänderte Dateien
# === NEUE VERSION BESTIMMEN (Semantische Versionierung) ===
major, minor, patch = map(int, prev_version.lstrip("v").split("."))
if added or deleted:
    minor += 1  # Änderung in Struktur → Minor-Bump
    patch = 0
elif modified:
    patch += 1  # Nur inhaltliche Änderung → Patch-Bump
new_version = f"v{major}.{minor}.{patch}"
```

```
print(f"Version updated: {prev_version} → {new_version}")

# === NOTEBOOK ZUSAMMENSTELLEN ===

nb = new_notebook()

nb.cells.append(new_markdown_cell(f"# SysML Report\nVersion:
{new_version}"))  # Titelseite

joined_models = "\n\n".join(sysml_models)  # Alle Modelle in eine
Code-Zelle einfügen

nb.cells.append(new_code_cell(joined_models))

nb.cells.append(new_markdown_cell("## Visualized Models"))  # Trenn-
überschrift für Visualisierungen

# === PAKETNAMEN AUS MODELLEN EXTRAHIEREN ===

package_pattern = re.compile(r"package\s+(?:'([^']+)'|(\w+))",
re.IGNORECASE)

package_names = set()

for model_text in sysml_models:

    matches = package_pattern.findall(model_text)

    for quoted, plain in matches:

        package_names.add(quoted or plain)  # Entweder der in Anfüh-
rungszeichen oder der einfache Name

# Für jedes Paket eine Visualisierungszelle (%viz)

for package in sorted(package_names):

    nb.cells.append(new_code_cell(f"%viz --view=DEFAULT --style=DE-
FAULT {package}"))

# === NOTEBOOK MIT VERSIONSNUMMER SPEICHERN ===

output_notebook = f"{base_output}_{new_version}.ipynb"

with open(output_notebook, "w", encoding="utf-8") as f:

    nbformat.write(nb, f)

print(f"Saved: {output_notebook}")

# === AKTUALISIERTEN ZUSTAND DER VERSION SPEICHERN ===

with open(version_file, "w") as f:

    json.dump({

        "version": new_version,

        "files": current_files

    }, f, indent=2)
```