Informatik Spektrum
https://doi.org/10.1007/s00287-026-01605-w

[SKP+26] A. Schweiger, H. Kausch, M. Pfeiffer, D. Raco, J. Lang, B. Rumpe:
Avionik-Software Engineering - Besonderheiten der Software-Entwicklung in der Luftfahrt.
In: Informatik Spektrum, Volume 49, DOI 10.1007/s00287-026-01605-w, Springer, Aug. 2026.

HAUPTBEITRAG

Avionik-Software Engineering – Besonderheiten der Software-Entwicklung in der Luftfahrt

Andreas Schweiger[1] · Hendrik Kausch[2] · Mathias Pfeiffer[2] · Deni Raco[2] · Jürgen Lang[1] · Bernhard Rumpe[2]

Eingegangen: 4. Februar 2026 / Angenommen: 6. März 2026

Zusammenfassung

Die Software-Entwicklung in der Luftfahrt unterscheidet sich wesentlich von der Software-Entwicklung in anderen Bereichen. Der Fokus dieses Beitrags ist die Analyse der dazu führenden Charakteristika, deren Auswirkungen und wie diese einer Mitigation zugeführt werden könnten. Die gekoppelte Verwendung von agilen Prozessen, modellgetriebenen Methoden und einheitlichen Modellierungssprachen sowie Werkzeugen kann zu einer Beherrschung der identifizierten Herausforderungen beitragen. Eine moderne, maßgeschneiderte Anpassung der bisher üblichen Software-Entwicklung unter Berücksichtigung von Zulassungsvorgaben kann Innovation im Luftfahrtbereich fördern und die Industrie stärken, indem eine ökonomischere Entwicklung ermöglicht wird. Mit dem Beitrag schärfen wir die avionik-spezifische Ausprägung des Software Engineering.

Einführung

In einem Luftfahrzeug ist je nach Ausprägung und Größe typischerweise eine dreistellige Anzahl von Bordrechnern eingerüstet. Diese tauschen Daten über für die Einsatzumgebung geeignete Kommunikationsmechanismen aus, die über adäquate Protokolle gesteuert werden. Die Gesamtheit der Rechnerknoten und ihrer Vernetzung kann somit als verteiltes System betrachtet werden. Beispielsweise bereiten Rechner die von Sensoren zur Verfügung gestellten Daten auf, um sie für die Anzeige an der Mensch-Maschine-Schnittstelle oder zur Weiterverarbeitung für Aktuatoren an andere Knoten zu versenden. Bei der Software für Luftfahrzeuge handelt es sich somit um sicherheitskritische, eingebettete Echtzeitsysteme, welche durch viele spezielle Normen begleitet werden. Die sich daraus ergebenden, besonderen Aspekte bei der Entwicklung solcher Software verschärfen die Herausforderungen, die in ähnlicher Form auch in anderen Anwendungsbereichen gegeben sind. Diese Charakteristika erschweren eine unmittelbare Anhebung der in der Luftfahrtindustrie vorherrschenden Entwicklungspraxis auf den Stand der Technik. In dem vorliegenden Beitrag werden diese besonderen Eigenschaften der Domäne identifiziert und beschrieben. Es folgt eine Darstellung von Vorschlägen zu deren Handhabung, um den luftfahrtbehördlichen, technischen und ökonomischen Anforderungen genügen zu können.

✉ Andreas Schweiger
andreas.schweiger@airbus.com

[1] Airbus Defence and Space GmbH, Manching, Deutschland

[2] RWTH Aachen University, Lehrstuhl für Software Engineering, Aachen, Deutschland

Besonderheiten der Software-Entwicklung

Im Folgenden werden die Charakteristika bei der Entwicklung von Software für Luftfahrzeuge dargestellt, die zu besonderen Herausforderungen im Vergleich zu anderen Anwendungsbereichen des Software Engineerings führen.

Lange Entwicklungszyklen

Die Entwicklung eines Luftfahrzeugs umfasst typischerweise vom Projektstart bis zu dessen Betriebsfreigabe einen Zeitraum von ca. ein bis zwei Jahrzehnten. Zum Zeitpunkt der Initiierung eines Entwicklungsprojekts erfolgt für die Auswahl der einzusetzenden Technologie eine Orientierung am Stand der Technik, allerdings unter Berücksichtigung der Zulassungsfähigkeit der Komponenten in Flugzeugen hinsichtlich der Belastung durch die Nutzungsbedingungen, wie Beschleunigungen, Vibrationen, elektromagnetische Felder oder kosmische Strahlung. Diese Fähigkeit ist gewöhnlich für die modernsten Komponenten noch nicht

Published online: 07 August 2026

geeignet nachgewiesen. Insofern liegen Komponenten, die in der Luftfahrt verwendet werden, bereits zu Beginn eines Projekts ggf. mehrere Generationen hinter den aktuellen Entwicklungen zurück. Demnach wird z. B. eine zum Projektstart geeignete Prozessorgeneration für den Einsatz in einem Bordrechner ausgewählt. Die Selektion einer bestimmten Hardware-Architektur hat die Auswahl zugehöriger Entwicklungswerkzeuge zur Folge. Auch eine sorgfältige Festlegung auf adäquate Technologien und Werkzeuge für die Software-Entwicklung kann einer unweigerlich einsetzenden Obsoleszenz von diesen nicht vollständig abhelfen. Deshalb sind die zu Beginn des Entwicklungsprozesses dem Stand der Technik entsprechenden Hardware-Komponenten, Methoden, Prozesse sowie Werkzeuge der Software-Entwicklung oftmals bereits am Ende der Systementwicklung obsolet. Teilweise wurde sogar die Erfahrung gemacht, dass die in der Designphase ausgewählten Hardware-Elemente aufgrund der raschen Entwicklungszyklen bereits zu Beginn der Entwicklungsphase obsolet sind. Nach der Entwicklung schließen sich noch Wartungs- und Weiterentwicklungsphasen mit ähnlicher zeitlicher Ausdehnung wie die initiale Entwicklung an, die durch den Einsatz obsoleter Hardware, Methoden, Prozesse und Werkzeuge verlangsamt werden, weil diese zum Zeitpunkt des Projektbeginns selektiert und damit nicht mehr dem Stand der Technik entsprechen. Ein Wechsel auf neuere Versionen und aktuellere Technologien erfordert eine zusätzliche Zertifizierung (Zulassung von Entwicklungsartefakten) bzw. Qualifizierung (Zulassung von Entwicklungswerkzeugen), welche mit entsprechend hohem Aufwand verbunden sind.

Starre Entwicklungsprozesse

Etablierte und teilweise von Auftraggebern oder Zulassungsbehörden vorgeschriebene Prozesse, die z. B. dem V-Modell oder dessen Nachfolger V-Modell XT folgen, können, wenn starr angewandt, die Entwicklungszeit verlängern, da Korrekturzyklen für die Integration, Verifikation und Validierung dabei besonders umfangreich ausfallen können. Dabei ist zu beachten, dass Änderungen an der Software ausschließlich über Änderungsaufträge vorgenommen werden dürfen. Jegliche Änderung einer zugelassenen Software – und sei es auch nur für minimale Wartungsarbeiten – führt aufgrund der Zulassungsvorgaben unmittelbar zum Erlöschen der Zertifizierung.

Agile Methoden können aus den folgenden Gründen [1] nicht unmittelbar um Einsatz kommen:

- Die nach dem agilen Manifesto weniger wichtigen Elemente (Prozesse und Werkzeuge, Dokumentation, Vertrag und Projektplan) sind essenziell für die Zertifizierung.
- Die ausdrückliche Berücksichtigung von Änderungen kann den Auftraggeber von der Verantwortung lösen, die Anforderungen beim Projektstart präzise zu definieren. Dies kann während der verbleibenden Projektlaufzeit zu zusätzlichem Aufwand, insbesondere für die Zertifizierung führen, da die anfänglich fehlende Präzision nachgeliefert werden muss.
- Kontinuierliche Verbesserung kann eine geänderte Vorgehensweise bei der Implementierung implizieren, was dem Zertifizierungsansatz widerspricht, strikt der Ausführung des vereinbarten Plans zu folgen.
- Die Formulierung von Anforderungen in der Form von in Prosa geschriebenen User Stories kann unvollständig, inkonsistent oder unpräzise sein, was wiederum zu nicht ausreichender Qualitätssicherung z. B. durch Testen und Verifikationslücken führen kann. Damit können die in für die Domäne typischen Standards formulierten Ziele, deren Erreichung für eine Zertifizierung unabdingbar nachzuweisen sind, potenziell nicht erfüllt werden.
- Sich während der Implementierungsphase wandelnde Architekturen, die bewusst antizipierte Veränderungen ausschließen, um Aufwand zu reduzieren, können sich im späteren Projektverlauf als unterdimensioniert erweisen.

Umfangreiche Qualifizierungs- und Zertifizierungsvorgaben

Die Entwicklung von Software für den Einsatz in einem Luftfahrzeug unterliegt strikten Regularien (manifestiert z. B. im Standard EUROCAE ED-12C) und erfordert eine Nachweisführung gegenüber den Zulassungsbehörden, um eine Zulassung für das Gesamtprodukt zu erhalten. Hierzu ist ein Entwicklungsvorgehen verpflichtend, das auf den von den Behörden geforderten Standards beruht. Letztere führen zu einem gemeinsamen Verständnis der Behörden und der Industrie, wodurch eine Vereinheitlichung des Zulassungsprozesses ermöglicht wird, auf dessen Basis eine Zulassung eines Luftfahrzeugs erfolgen kann.

Standards dieser Art sind oftmals konservativ in ihren Vorgaben, weil die Betriebssicherheit eines Luftfahrzeugs unbedingt zu gewährleisten ist. Ergo werden darin neue Technologien nicht oder erst nach geraumer Zeit ihrer Etablierung in anderen Bereichen berücksichtigt. Dies führt dazu, dass neue Technologien und Methoden möglicherweise erst verzögert in Luftfahrzeugen bzw. bei deren Entwicklung zum Einsatz kommen.

Außerdem entstehen bei der Anwendung der Standards sehr hohe Aufwände [2] in der Nachweisführung als Voraussetzung für die Zertifizierung der Software – insbesondere bei hoher, die Flugsicherheit betreffender Kritikalität

der Software. Dabei umfasst die Phase der Verifikation den größten Teil des Gesamtentwicklungsaufwands [3].

Steigende Qualifizierungs- und Zertifizierungsvorgaben sorgen dafür, dass Testen in besonders sicherheitskritischen Bereichen zur Qualitätssicherung nicht ausreichend sein kann. Entweder kann eine vollständige Testabdeckung aufgrund des Umfangs nicht erreicht werden, oder spezifizierte Eigenschaften (z. B. zur Gewährleistung der Lebendigkeit) können mit dem Testen gar nicht überprüft werden.

Tazites System- und Prozesswissen

Die umfangreichen Entwicklungs- und die sich daran anschließenden Wartungszeiträume können sich über einen wesentlichen Anteil des Arbeitslebens eines Ingenieurs erstrecken. Herausforderungen entstehen dann, wenn das Wissen nicht an die nächste Generation der Entwickler in geeigneter Form weitergegeben werden kann, bevor die Wissensträger aus dem Arbeitsleben ausscheiden.

Komplexe Testumgebungen zur Qualitätssicherung

Die umfangreiche Entwicklungsdauer des Gesamtsystems wird auch durch die Entwicklung, den Aufbau und die Qualifizierung von komplexen Testumgebungen bedingt. Beispielsweise müssen beim isolierten Testen eines einzelnen Bordrechners seine Kommunikationspartner und die zugehörige Infrastruktur vollständig in Hardware oder auch ggf. in Software abgebildet werden, falls die Hardware noch nicht vorliegt. Die Bereitstellung der Umgebung ist deshalb erforderlich, weil die Software von Bordrechnern üblicherweise die Ausgaben anderer Rechner als eigene Eingaben weiterverarbeitet. Wird ein Rechner isoliert von seiner Umgebung betrieben, fehlen die Stimuli der anderen Rechner, weshalb die Software des betrachteten Rechners keine Verarbeitung vornimmt. Aufgrund dieser Abhängigkeiten entwickelt sich ein Testsystem zu einer Komplexität, die derjenigen des zu testenden Systems ähnelt.

Eine weitere Erhöhung der Komplexität entsteht durch die Notwendigkeit, den Einsatz des entwickelten Systems unter realistischen Bedingungen zu testen. Dazu werden aufeinander abgestimmte Flug- und Umgebungssimulationen benötigt. Beispielsweise darf es nicht vorkommen, dass für einen Sensor ein Steigflug simuliert wird, während bei einem anderen die Flughöhe konstant bleibt.

Hoher Software-Umfang und bedeutende Komplexität

Die beständige Zunahme der Komplexität und des Umfangs von Software-Systemen betrifft auch die Luftfahrt [4; S. 166]. Die Ursachen des besonders hohen Umfangs und der Komplexität einer fliegenden Plattform liegen in der Vielzahl von systeminternen und -externen Schnittstellen, den steigenden Anforderungen in den Bereichen Automatisierung während des Flugbetriebs eines Luftfahrzeugs, der Systemsicherheit sowie bei den Passagieransprüchen z. B. in Bezug auf Bordunterhaltung oder die Integration von mobilen Endgeräten in das Bordunterhaltungssystem.

Nischenartige Programmiersprachen

Die Software wird oftmals in Ada entwickelt. Obwohl diese Programmiersprache mit ihrer strengen Typsicherheit dediziert für die Entwicklung von sicherheitskritischer Software entworfen[1] und stetig weiterentwickelt[2] wurde, existiert eine entsprechende Community nur in geringem Ausmaß, die in ihrem Umfang wesentlich kleiner zu denjenigen von weit verbreiteten Sprachen, wie z. B. C, C++ oder Rust, ist. Dies erschwert den Austausch unter Entwicklern z. B. in Foren wie Stack Overflow, um (ähnliche) Programmieraufgaben oder besondere Spracheigenheiten zu diskutieren, wodurch eine Verlangsamung der Entwicklung bedingt wird.

Deterministische Hardware für Bordrechner

Die bisher im Einsatz befindlichen Einkernprozessoren stoßen aufgrund der Zunahme der Komplexität und der steigenden Anforderungen der zu implementierenden Software-Funktionalitäten an ihre Leistungsgrenze, die bisher durch die Erhöhung von Taktfrequenzen verschoben wurde, und sind oftmals der limitierende Faktor für Funktionserweiterungen, die sich auf die gesamte Lebensdauer eines Luftfahrzeugs beziehen. Damit und aufgrund des in absehbarer Zeit erwarteten Verschwindens vom Markt sind Einkernprozessoren voraussichtlich nicht zukunftsfähig, weshalb zukünftig die Notwendigkeit für den Einsatz von Mehrkernprozessoren zunimmt.

Die Erarbeitung von Standards zur Anwendung in der Luftfahrt basiert oftmals auf Erfahrungen. Jedoch liegen für moderne Technologien, wie z. B. Hyperthreading (parallele Ausführung von Software auf einem Prozessorkern), eine dynamische Allokation von Ressourcen (z. B. Prozessor oder Speicher) zur Nutzung durch verschiedene Software-Komponenten oder der Einsatz von Mehrkernprozessoren bisher geringe Erfahrungen für deren Einsatz in der Luftfahrt vor. Zudem werden Technologien wie die oben genannten von aktuellen Standards (EUROCAE ED-80 für Hardware) noch nicht abgedeckt. Das im Jahr 2016 vom Certification Authorities Software Team (CAST, eine in-

[1] Ada 83 spezifiziert nach MIL-STD-1815 und standardisiert nach ANSI/MIL-STD-1815A sowie ISO-8652:1987.

[2] Ada 95 nach ISO-8652:1995, Ada 2005 nach ISO/IEC-8652:1995/AMD 1:2007, Ada 2012 nach ISO/IEC-8652:2012 und Ada 2022 nach ISO/IEC 8652:2023 standardisiert.

ternationale Arbeitsgruppe von Vertretern von Zulassungsbehörden) vorgelegte Papier CAST-32A stellte seine damalige Perspektive auf den Einsatz von Mehrkernprozessoren in der Luftfahrt dar. Abgelöst wurde das Dokument 2022 im Zuständigkeitsbereich der europäischen (European Union Aviation Safety Agency, EASA) durch AMC 20-193 und 2024 in demjenigen der nordamerikanischen Zulassungsbehörde (Federal Aviation Administration, FAA) durch AC 20-193. Dennoch werden Technologien wie Hyperthreading oder dynamische Reallokation von Prozessen darin nicht dediziert betrachtet. Ergo existieren Lücken, die eine vollumfängliche Zertifizierung von Mehrkernprozessoren erschweren. Dies ist damit begründet, dass kein Konsens darüber etabliert werden konnte, welche Designschritte erforderlich sind sowie welcher Zuverlässigkeitsgrad von durchgeführten Tests nötig ist.

Aus technischer Sicht wird in Mehrkernprozessoren potenziell nicht deterministisches Verhalten durch Race Conditions oder Zugriffskonflikte auf gemeinsam genutzte Ressourcen (z. B. Caches) verursacht. Aufgrund dessen konnte bisher keine Zertifizierung erfolgen, weshalb bis dato nahezu ausschließlich Einkernprozessoren in der Luftfahrt eingesetzt oder beim Einsatz von Mehrkernprozessoren sehr konservative Ansätze verfolgt werden, welche z. B. durch die Deaktivierung aller bis auf einen Kern gekennzeichnet sind. Solche Ansätze für Mehrkernprozessoren können zur Folge haben, dass die verfügbare Leistung beträchtlich hinter der durchschnittlichen Leistung zurückbleibt oder teilweise geringer ausfällt als bei der Verwendung von Einkernprozessoren.

Diese beschriebene Problematik hängt eng mit den aktuellen Vorgaben hinsichtlich der geforderten Betriebssicherheit der Systeme zusammen: rein deterministisches Verhalten und eine hohe Sicherheit gegenüber Fehlern. In der höchsten Sicherheitsklasse sind häufig etwa 10^{-9} Fehler pro Flugstunde vorgegeben.

Ansätze zur Beherrschung der identifizierten Herausforderungen

Im Folgenden werden Ansätze vorgeschlagen, um die oben beschriebenen Herausforderungen einer geeigneten Mitigation zuzuführen. Eine geeignete Umsetzung der nachfolgenden Maßnahmen durch adäquate, auf die Avionik zugeschnittene Prozesse, Methoden und Techniken führt zu einer eigenständigen Variante des Software Engineering, die entsprechend auch im Berufsbild, der Ausbildung und notwendigem Skill-Set reflektiert wird.

Definition 1 [Avionik-Software Engineering] Avionik-Software Engineering ist die Spezialisierung und Anwendung von Software Engineering-Techniken speziell für Software im Fluggerät und zur vernetzten Unterstützung von Flügen unter Berücksichtigung der geltenden Normen, Standards und Gesetze.

Beherrschung langer Entwicklungszyklen

Für die Beherrschung von obsoleter Hardware und veralteten Methoden, Prozessen sowie Werkzeugen müssen jeweils Potenziale für adäquate Effizienzsteigerungen identifiziert und umgesetzt werden. Obwohl neue Methoden erst nach der Verfügbarkeit entsprechender Standards zum Einsatz kommen können, ist die Integration von Werkzeugen für die Gewinnung von Effektivitäts- und Effizienzvorteilen auch unabhängig davon geeignet, um die Entwicklung in laufenden Projekten zu beschleunigen. Für ihre Integration in den Entwicklungsprozess ist jedoch ihre Qualifizierung in Abhängigkeit der Nutzungsart der von den Werkzeugen generierten Ergebnisse erforderlich. Die Werkzeuge müssen dabei hinsichtlich ihrer Funktionsqualität und Fehleranfälligkeit ein Qualifizierungsniveau besitzen, das der Sicherheitsrelevanz der Software für das Luftfahrzeug entspricht. Weil die Qualifizierung zusätzlichen Aufwand verursacht, müssen die durch die Einführung gewonnenen Effektivitäts- und Effizienzvorteile die damit verbundenen ökonomischen Kosten mindestens amortisieren können. Deshalb ist eine sorgfältige Evaluierung von Nutzen und Kosten erforderlich. Die Kosten sind dabei nicht isoliert in Bezug auf ihre unmittelbare, sondern ihre langfristigen Auswirkungen zu betrachten. Denn durch den Einsatz eines Werkzeugs kann das Fundament für die Einführung einer permanenten und durchgängigen CI/CD-Kette gebildet werden, obwohl die initiale Einführung mit umfangreichem Aufwand verbunden sein kann. Damit wird die Grundlage auch für zukünftige Effektivitäts- und Effizienzpotenziale gelegt, durch welche die anfängliche Kostenamortisierung signifikant befördert werden kann [5, 6].

Anhand verschiedener Vorgehensweisen wird mit folgendem Beispiel beschrieben, welche Effizienzpotenziale gehoben werden könnten: Nach funktionalen Änderungen in der Software können die von der Anpassung betroffenen Tests selektiv vorgenommen werden. Dies ist gewöhnlich mit einer No-Impact-Analyse für die anderen Bereiche verbunden. Zu beachten ist, dass beim selektiven Vorgehen die Analyse oftmals manuell vorgenommen wird. Eine Effizienzsteigerung ist dann möglich, wenn die durchzuführenden Tests mit einer automatisierten Test-Gap-Analyse [7] identifiziert und gleichzeitig die notwendige Information für die No-Impact-Analyse generiert werden können. Damit wird neben der Effizienz zugleich die Effektivität erhöht, weil die relevanten Tests sowie die erforderlichen Regressionstests identifiziert werden. Alternativ zum selektiven Vorgehen kann die Gesamtheit der Tests ausgeführt werden, wobei einerseits die Testdauer signifikant zunimmt und ei-

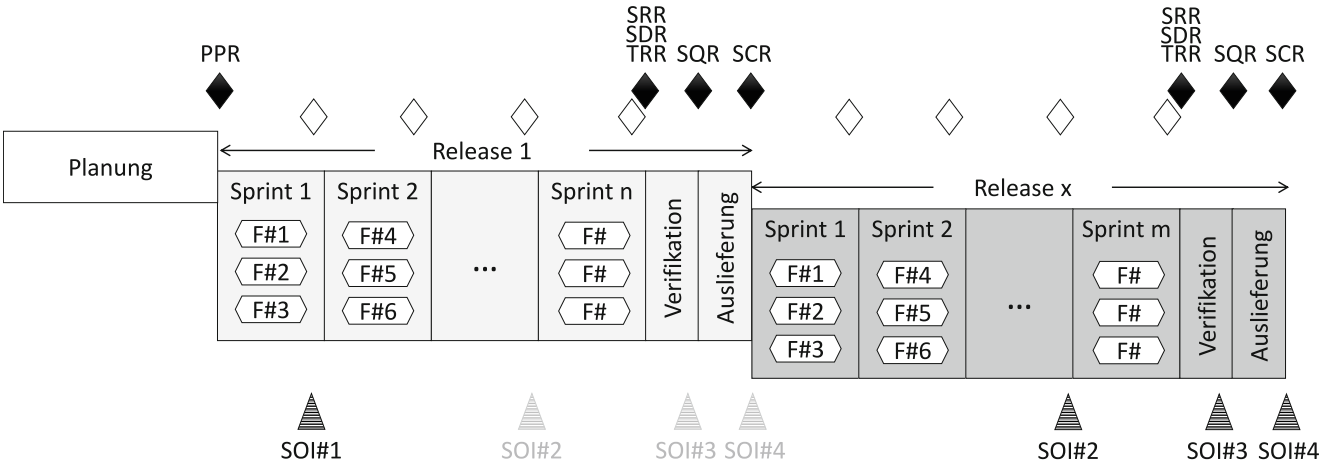

Abb. 1 Vorschlag für einen agilen Prozess für die Anwendung bei der Entwicklung von Software für die Luftfahrt (in Anlehnung an [1]).

ne Verlangsamung des Entwicklungsprozesses riskiert wird, andererseits aber die No-Impact-Analyse abkömmlich wird. Deshalb ist eine geeignete Analyse vorzunehmen, um für beide Varianten die entstehenden Aufwände zu quantifizieren, bevor eine Entscheidung zum Testvorgehen bzw. für die Einführung einer Werkzeugunterstützung getroffen werden kann. Dabei ist zu berücksichtigen, dass eine geeignete Werkzeug-Qualifizierung vorzunehmen ist, insofern bei der Test-Gap-Analyse der Testumfang automatisiert definiert wird.

Als Beispiel für mögliche Auswirkungen eines langen Entwicklungszyklus kann Erosion in der ursprünglich definierten Software-Architektur bzw. im -Design angeführt werden. Erosion entsteht beispielsweise durch das Kopieren und Einfügen von Software-Teilen, Code Clones, durch die wiederum das Risiko von Qualitätsmängeln erhöht wird [8], weil durch die kopierten und ggf. individuell modifizierten Teile potenziell Fehler verbreitet werden, ohne diese in ihrem vollen Umfang an den jeweiligen Positionen nachverfolgen zu können. Moderne Entwicklungswerkzeuge können die Identifikation und Beseitigung von Erosion durch Code Clones in der Software unterstützen. Eine Qualifizierung ist dabei nicht erforderlich, weil die bereits bestehende Qualitätssicherung vom Werkzeugeinsatz unberührt bleibt.

Weitere Effizienzsteigerungen können mit einer durchgängigen Werkzeugintegration erreicht werden [9], welche oftmals nicht vorherrschend ist. Eine solche gelingt besonders vorteilhaft durch die Anwendung von modellbasierten Konzepten, um darüber die Integrationsbasis zu schaffen. Solche Konzepte sind in dem Standard EUROCAE ED-218 beschrieben, welcher als Ergänzung zu EUROCAE ED-12C zu betrachten ist. Eine Qualifizierung ist in Abhängigkeit der Auswirkungen auf die Qualitätssicherung vorzunehmen.

Angepasste Entwicklungsprozesse

Marsden et al. [1] schlagen einen agilen Prozess (Abb. 1) vor, der für die Anwendung bei der Entwicklung von Software in der Luftfahrt geeignet ist. Dabei wurde z. B. die mit den Vorgaben aus den standardisierten Entwicklungsprozessen konfligierende Priorisierung der agilen Elemente Individuen und Interaktionen, funktionsfähige Software, Kundenzusammenarbeit und Reaktion aufgelöst. Außerdem werden die in der Domäne typischen Quality Gates Project Planning Review (PPR), Software Requirements Review (SRR), Software Design Review (SDR), Test Readiness Review (TRR), Software Qualification Review (SQR) und Software Certification Review (SCR) explizit berücksichtigt. Da die für die Gates SRR, SDR und TRR erforderlichen Artefakte Anforderungen, Design und Testprozeduren erst nach der vollständigen Implementierung vorliegen, müssen diese Gates gleichzeitig durchlaufen werden. Alternativ können nach jedem Sprint Reviews (weiße Rauten in Abb. 1) durchgeführt werden, die in ihrem Umfang reduziert sind. Mit diesen Vorschlägen finden insbesondere Änderungen bei den Anforderungen, der Vorgehensweise bei der Implementierung, die kontinuierliche Präzisierung von Anforderungen[3] und die permanente Weiterentwicklung von Architekturen Berücksichtigung. Weiterhin sind die in den Sprints zu implementierenden Funktionalitäten (F#1...6 und F#) fein granular genug zu definieren, um von den Vorteilen des agilen Prozesses profitieren zu können. Für eine erfolgreiche Zulassung ist es empfehlenswert, die zuständigen Behörden zu sinnvollen Zeitpunkten im Entwicklungsprozess in ähnlicher Weise wie bei Vorgehens-

[3] Bei der Erstellung der Anforderungen ist immer zu berücksichtigen, wie deren korrekte Implementierung nachgewiesen werden kann. Anforderungen, die nicht nachweisbar sind – weder per Test noch per Analyse oder formaler Verifikation – sind unbrauchbar.

modellen nach dem V-Modell zu involvieren (SOI, Stages of Involvement). Auch diese Zeitpunkte sind geeignet zu wählen, um die Zulassung zu unterstützen (hellgraue SOI-Symbole in Abb. 1). Mit dem beschriebenen Prozess konnten Marsden et al. folgende Verbesserungen in der Entwicklung nachweisen:

- Reduktion der Entwicklungszeit um den Faktor 1,7 bis 2
- Reduktion der Anzahl von Fehlern um den Faktor 3,8
- Erhöhung der Produktivität um den Faktor 1,7 bis 2

Ein adäquater Prozess für die Software-Entwicklung muss nicht nur die Konstruktion des zu entwerfenden Systems, sondern auch die Anfertigung der Verifikationsartefakte begleiten. Besonderes Augenmerk liegt auf den frühen Phasen der Entwicklung, da dort entstehende Fehler besonders hohe Kosten nach sich ziehen. Die Grundlage für geeignete Anpassungen des Entwicklungsprozesses wird im Standard EUROCAE ED-216 gebildet, welcher als Erweiterung von EUROCAE ED-12C den Einsatz formaler Methoden für Software in der Luftfahrt beschreibt. Moy et al. [10] konnten zeigen, dass die Anwendung formaler Methoden für die Verifikation von Software in der Luftfahrt Effizienzsteigerungen bewirkt. In dem genannten Standard nimmt die Entwicklung der Artefakte System Requirement (SRs), High-Level Requirement (HLRs) und des Designs bestehend aus Architektur und Low-Level Requirement (LLRs) einen zentralen Stellenwert ein. Dabei sind die Compliance, Konsistenz, Verifizierbarkeit und Nachverfolgbarkeit zwischen diesen Entwicklungsphasen und den entsprechenden Artefakten sicherzustellen. In Abb. 2 sind diese Zusammenhänge der Phasen dargestellt. Im grau hinterlegten Kasten sind die frühen Phasen und ihre Relationen hervorgehoben. Diese Phasen stehen deshalb im Fokus, weil dort entstehende Fehler besonders hohe Kosten nach sich ziehen, wenn sie erst in späten Phasen entdeckt werden.

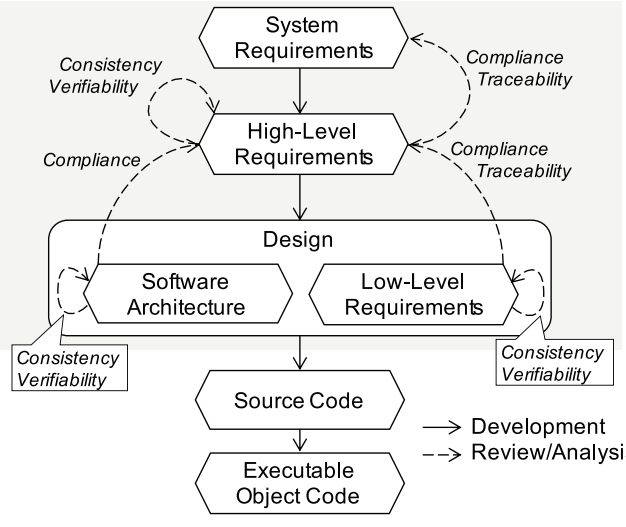

Abb. 2 Ein geeigneter Prozess sollte die Compliance, Konsistenz, Verifizierbarkeit und Rückverfolgbarkeit zwischen SRs, HLRs, Software-Architektur und LLRs im systematischen Entwurf gewährleisten, wie er im Standard EUROCAE ED-216 beschrieben ist.

Die Compliance gewährleistet eine korrekte Weiterentwicklung der Anforderungen phasenübergreifend. Konsistenz stellt sicher, dass keine Widersprüche in den Anforderungen gegeben sind. Verifizierbarkeit erlaubt die Überprüfung der Artefakte. Nachverfolgbarkeit dient der Gewährleistung der Vollständigkeit und der ausschließlichen Implementierung der anfänglichen Requirements. Nachverfolgbarkeit kann auch Agilität unterstützen, indem sie Abhängigkeiten sichtbar und analysierbar macht. Dadurch kann beispielsweise bei Änderungswünschen abgeschätzt werden, welche benachbarten Systeme oder Anforderungen betroffen sind. Potenzielle Probleme können damit einfacher auf ihren Ursprung zurückgeführt werden, wodurch Störungen im Entwicklungsprozess reduziert werden.

Beherrschung der Qualifizierungs- und Zertifizierungsvorgaben

Im Standard EUROCAE ED-216 wird der Einsatz formaler Methoden für die Sofware-Entwicklung in der Luftfahrt beschrieben. Dadurch wird die Entfaltung von deren Mächtigkeit ermöglicht. Für ihren Einsatz ist aber die nötige Expertise in abstrakter Interpretation [11], im Model Checking [12] oder in deduktiven Methoden (z. B. Theorembeweisern [13]) erforderlich. Gleichzeitig benötigen formale Methoden eine formale, eindeutige Semantikabbildung der zu analysierenden Systeme. Dabei muss bei dieser Abbildung eine geeignete semantische Domäne für den Anwendungsfall ausgewählt werden [14]. Für sicherheitskritische Echtzeitsysteme in der Luftfahrt ist beispielsweise die ereignisbasierte Verarbeitung [15, 16] geeignet, welche auf Strömen basiert [17, 18].

Formale Verifikation birgt das Risiko, hohe Kosten zu verursachen. Ein methodischer Umgang mit diesem mächtigen Werkzeug ist ebenso nötig wie eine anwenderfreundliche Schnittstelle. Modellbasierte Methoden, deren Anwendung in EUROCAE ED-218 beschrieben sind, können hier Abhilfe schaffen. In der Industrie relevante Sprachen wie SysML v2 und geeignete Modellprofile können Verwendung finden, um Anforderungen an Systeme zu spezifizieren [19]. Mechanisierte Übersetzer überführen Modelle einer solchen Sprache, die ggf. durch die Verwendung eines passenden Profils eingeschränkt werden, z. B. für SysML v2 [20], in ein semantisches Backend. Im Backend sind Bedeutungen von und Beziehungen zwischen verschiedenen Entitäten in einem System erfasst und analysierbar, beispielsweise in einem Theorembeweiser [21]. Dieses Vorgehen kann hoch automatisierte Analysen des modellierten Systems liefern, beispielsweise zum Zweck der Verifikati-

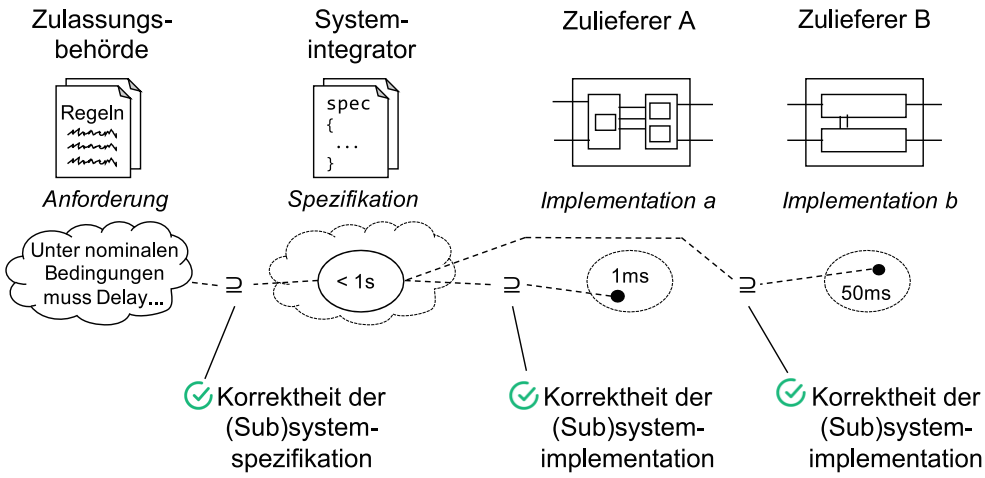

Abb. 3 Beispielhafte Entwicklung eines funkbasierten Datenübertragungssystems für die Luftfahrt. Das System besitzt eine vorgegebene maximale Übertragungsverzögerung. Ein Systemintegrator muss nachweisen, dass seine Spezifikation den Anforderungen (links) entspricht. Zulieferer müssen nachweisen, dass ihre Implementierung die Spezifikation des Systemintegrators erfüllt.

on. Die so geprüften Modelle können gleichermaßen zur weiteren Entwicklung wie zur Synthese von Code verwendet werden [22].

Zusätzlich ließen sich die Verifikationsergebnisse wiederverwenden, wenn eine durchgängige Verifikation (im Gegensatz zur Inselverifikation bestimmter Teile oder stichpunktartigen Verifikation nur zu bestimmten Meilensteinen) angestrebt wird. Die Übernahme von Verifikationen erfordert jedoch, dass sie auf identischen Entitäten erfolgt sind. Zudem müssen die Verifikationen auch mit den für das Gesamtsystem geforderten Prozessvorgaben und Methoden erzeugt sein oder zumindest mit akzeptablen Alternativen.

Beispielsweise zeigt Abb. 3, dass Integratoren von Korrektheitsnachweisen der Zulieferersysteme profitieren können. Gleichzeitig müssen Zulieferer keine Korrektheitsnachweise für die Integration ihrer Systeme erbringen. Zulieferersysteme können stattdessen gemäß dem Prinzip Design by Contract [23] an einer (formalen) Spezifikation gemessen werden. Dieses Vorgehen kann auch teure Testaufbauten reduzieren. Dem Integrationstest obliegt es dann, die Korrektheit der Schnittstellen und der Kommunikation zu überprüfen, was unabhängig von den Zulieferern auf Basis der formalen Spezifikationen geschehen kann.

Verstärkung des Kompetenzfeldes Wissensmanagement

Es kann beobachtet werden, dass das Berufsbild des Informatikers starken Änderungen unterliegt. Brenner/Broy/Leimeister schlagen deshalb für die Industrie die dafür geeigneten Ausprägungen „konventionelle“ Informatiker, Informatikführungskräfte, Entrepreneurship-Informatiker sowie innovative Informatiker vor [24]. Ergänzend dazu und für alle genannten Ausprägungen im selben Ausmaß ist besonderer Fokus auf das kontinuierliche Wissensmanagement zu richten. Dabei sollen bereits während der Entwicklung einzelner Bereiche die grundlegenden Ideen umfassend dokumentiert werden. Etablierte Standards wie ISO/IEC/IEEE 15288:2015 greifen diese Notwendigkeit auf, um zu gewährleisten, dass tazites bzw. implizites Wissen in explizites transformiert wird, um die gesammelte Expertise auf weitere Wissensträger transferieren zu können, damit diese nicht verloren geht.

Qualitätssicherung durch modellbasiertes Testen und formale Verifikation

In der Industrie und in Zulassungsrichtlinien ist das Testen zur Verifikation weit verbreitet. Modellbasiertes Testen erlaubt es, bereits früh im Entwicklungsprozess eine Qualitätssicherung vorzunehmen und Fehler früh zu detektieren sowie zu beheben. Dies ist besonders wichtig, da später gefundene Fehler überproportional teuer sind [25, 26] und gute Modelle ein Indikator für gute Software-Produktqualität sind [27, 28]. Dabei kann automatisierte Testgenerierung die Konstruktion von Testfällen mechanisieren [29].

Formale Verifikation kann die Qualitätssicherung durch Testen nicht ersetzen et v.v. Beiden Methoden sollten deswegen koexistieren und in einer Weise entwickelt werden, die den Wert des Testens verbessert, komplementiert und ausnutzt. Eine Synergie mit formaler Verifikation ist anzustreben, denn Testen erfordert ebenso eine präzise Semantik wie die Verifikation. Durch eine geeignete Semantik, die beispielsweise Unterspezifikation beherrscht [30, 31], können auch abstrakte Modelle getestet werden. So existieren Ansätze, die Theorembeweiser als Testplattform einsetzen [32]. Ein solches Vorgehen kann teure Testsysteme zwar nicht ersetzen, jedoch den entstehenden Testaufwand reduzieren, indem Fehler früher gefunden werden oder auch Teilsysteme durch virtuelle (z. B. simulierte) Systeme (Digital Twin) ersetzt werden, die aus Modellen synthetisiert wurden. Ebenso hilft Testen dabei, Fehlerquellen vor einer relativ teuren Verifikation zu reduzieren und so einen gewissen Reifegrad der Modelle zu gewährleisten.

Die formale Verifikation von Software in der Sprache C kann durch den Einsatz des Frameworks Frama-C unterstützt werden. Dieses Framework bietet die vertragsbasierte (Design by Contract) Programmierung an. Für Ada ist ein entsprechendes Pendant mit der Sprache SPARK[4] gegeben.

Bei den vorgeschlagenen Methoden ist zu beachten, dass für deren Nutzung auch ein geeigneter Nachweis ihrer Fähigkeiten bzw. eine Zulassung erforderlich ist, wenn die Artefakte für eine Zulassung benötigt werden. Dabei entsteht zusätzlicher Aufwand, der mit den erwarteten Einsparungen gerechtfertigt werden kann.

Beherrschung des Software-Umfangs und der -Komplexität

Komplexität lässt sich allgemein mit Methoden zur Abstraktion und Konkretisierung begegnen. Abstraktionen erlauben es, Übersicht zu gewinnen, während Konkretisierungen diesen Abstraktionen an entscheidenden Stellen Details hinzufügen, die eine feinere Kontrolle über die modellierten Konzepte erlauben. Dieses Verfahren sieht auch der in Abb. 2 skizzierte Entwicklungsprozess vor: Abstrakte System Requirements (SR) werden über HLRs bis hin zu LLRs einschließlich Architektur konkretisiert. Modellgetriebene Methoden bieten diese Abstraktionen per definitionem an, da ein Modell eine Abstraktion eines Originals darstellt [33]. Wesentlich bei der Auswahl einer Modellierungssprache und eines methodischen Frameworks ist die Fähigkeit zur Bildung von mehreren Abstraktionsebenen. Nur so lassen sich die verschiedenen Ebenen (SRs, HLRs, LLRs und Architektur) abdecken. Entscheidend ist dabei, dass diese Artefakte nachverfolgbar verbunden sind und die Compliance gesichert werden kann. SysML v2 erlaubt es beispielsweise, Komponenten bzw. deren Verhalten deklarativ durch *Constraints* und imperativ durch *States* (Automaten [34, 35]) anzugeben. Ein einfach erweiterbares Konzept von *Spezialisierung* kann zur Nachverfolgbarkeit genutzt werden. Spezialisierungen setzen zwei oder mehrere Spezifikationen in Relation und können dadurch die Weiterentwicklung (Verfeinerung) der Spezifikation beschreiben, womit auch die Nachverfolgbarkeit gegeben ist.

Eine weitere Möglichkeit, Komplexität zu begegnen, ist die Anwendung des Prinzips, zu teilen und zu herrschen (lat. divide et impera). Dabei werden das Problem und ggf. auch die Lösung in leichter zu verstehende und realisierende Teile zerlegt [17]. Auch hier ist darauf zu achten, solche Sprachen und Methoden einzusetzen, die (De-)Komposition beherrschen. SysML v2 beispielsweise benutzt *Composite Parts* und das Konzept der Definition und Instanz, um wiederverwendbare, parametrisierbare Komponenten abzubilden. Ebenso lassen sich dieselben Modelle für verschiedene Plattformen einsetzen, indem Transformatoren ausgetauscht werden. Solches Vorgehen bietet typischerweise eine geringere Fehleranfälligkeit und höhere Agilität, da die effektiven Quelldaten, die Modelle, einen geringeren Umfang besitzen als der lauffähige und aus den Quelldaten generierte Code. Dabei ist die geplante einer ungeplanten Wiederverwendung [36] vorzuziehen, da letztere über ein Reverse Engineering erst in eine geplante Wiederverwendung überführt werden muss, womit zusätzliche Kosten anfallen.

Nota bene, dass Wiederverwendung in den in der Luftfahrt eingesetzten Standards nicht notwendigerweise explizit erwähnt wird und damit nicht ohne Weiteres zum Einsatz kommen kann. Gegebenenfalls sind geeignete Zertifizierungsschritte erforderlich, um eine Zulassung der wiederverwendeten Artefakte in einem anderen Kontext (i.e. für ein anderes Produkt) zu ermöglichen. Eine wesentliche Voraussetzung zur Wiederverwendung ist eine vollständige Information über Nutzungsrandbedingungen, Limitierungen, Annahmen, Näherungen, Randbedingungen und ähnliches. Häufig ist es schwierig, zu überprüfen, ob die heranzuziehende Information tatsächlich vollständig ist.

Programmiersprachen der Hauptströmung

Die herausragende Eigenschaft der für die Implementierung von eingebetteten Systemen häufig eingesetzten Programmiersprache Ada ist deren starke Typsicherheit. Im Vergleich dazu zeichnen sich die Sprachen C und C++ zwar durch eine aufgrund der Hardware-Nähe gegebene Performanz, jedoch auch durch eine Fehleranfälligkeit aus, die z. B. durch die häufig zum Einsatz kommende Zeigerarithmetik verursacht wird. Deshalb wurde für C und C++ jeweils eine Teilmenge des vollständigen Sprachschatzes vom Konsortium MISRA (The Motor Industry Software Reliability Association) definiert, um die Zuverlässigkeit der mit diesen Sprachen für sicherheitskritische Systeme implementierten Software zu erhöhen. Dieser Weg wurde auch mit der Spezifikation von JSF++ [37] für die Implementierung von Software für Luftfahrzeuge auf der Basis von C++ beschritten.

Die von der Mozilla Foundation spezifizierte Sprache Rust besitzt hohe Laufzeitperformanz und gleichzeitig hohe Typsicherheit, da diese explizit für einen Einsatz in eingebetteten Systemen entwickelt wurde. Zum aktuellen Zeitpunkt arbeitet die der Standardisierungsorganisation SAE International® zugeordnete Arbeitsgruppe SAfEr Rust Task Force an einer Spracheinschränkung, um den Anforderungen für Software für sicherheitskritische Systeme noch besser zu genügen. Außerdem ist aufgrund ihrer weiten Verbreitung eine gute Unterstützung in der Community gege-

[4] Für SPARK 2014 siehe https://www.adacore.com/about-spark, zugegriffen am 29.02.2024.

ben, da z. B. der Browser Mozilla Firefox in dieser Sprache implementiert ist.

Damit auch bereits laufende Projekte, in denen die Sprache Ada zum Einsatz kommt, von den oben genannten Vorteilen profitieren können, könnte Legacy-Software einmalig mit geeigneten Transformatoren von Ada in die gewünschte Zielsprache[5] übersetzt und auf dieser Grundlage weiterentwickelt sowie gewartet werden. Es kann bei diesem Vorgehen davon ausgegangen werden, dass zunächst Anwendungen mit niedrigen Anforderungen hinsichtlich Sicherheit (niedriger Design Assurance Level) zugelassen werden. Anwendungen mit höheren Kritikalitätsanforderungen würden dann nach erfolgreicher Nutzung jenen Implementierungen folgen.

Die Verwendung einer in der Industrie weit verbreiteten Modellierungssprache wie SysML kann Teile der Software-Entwicklung ersetzen. Die aktuelle Fassung SysML v2 bietet eine klare Semantik und die Möglichkeit, auch in Textform ausdrucksstark zu modellieren. Dem Prinzip von Model-driven Engineering (MDE) [22] folgend, könnte durch die Verwendung von geeigneten Transformatoren die Architektur und sogar das Verhalten der Software direkt aus Modellen synthetisiert werden. Vorteile der Verwendung von Modellen sind Konzentration auf die Problemdomäne statt auf den Lösungsraum. Diese Details ließen sich durch Transformatoren einmalig generisch lösen und könnten so wiederverwendet werden. Dem Beispiel der Einschränkung des Sprachschatzes von modernen Programmiersprachen folgend, kann die gewählte Modellierungssprache auf ein Profil eingeschränkt werden. So kann beispielsweise auch eine Verwendung von formalen Backends (siehe Abschn. 3.3) zur Modellanalyse ermöglicht werden. Generell ist es deshalb von Vorteil, Methoden und Werkzeuge einzusetzen, die die Modellierung von Software sowie die Generierung von Code unterstützen. Im Zusammenhang mit einer geeigneten Qualifizierung der eingesetzten Werkzeuge, welche eine unmittelbare Zulassung des generierten Codes ermöglicht, ist dadurch ein wesentlicher Beitrag zur Automatisierung identifiziert.

Zulassung von Mehrkernprozessoren

Bei Mehrkernprozessoren kann zwischen zwei Architekturansätzen differenziert werden: Bei AMP-Architekturen (Asymmetric Multi-Processing) verfügt jeder Kern über einen eigenen, dedizierten Adressbereich (insbesondere Cache und Speicher). Oft sind auch auf jedem Kern mehrere Instanzen eines Betriebssystems installiert, die fast vollständig unabhängig betrieben werden können. Für die Gewährleistung eines für die Zulassung vorausgesetzten, möglichst deterministischen Betriebs werden typischerweise die Zugriffsraten von dedizierten Prozessorkernen auf die gemeinsam genutzten Ressourcen eingeschränkt, was zu einer Reduktion der Varianz, jedoch auch zu einem Anstieg der Ausführungszeit führt.

Bei SMP-Architekturen (Symmetric Multi-processing) teilen sich die Kerne den Adressbereich. Typischerweise läuft ein Betriebssystem auf allen Kernen, welches die Allokation der verfügbaren Ressourcen vornimmt. Für die Sicherstellung des erforderlichen Determinismus übernimmt ein Prozess die Koordination der Zugriffe auf Ressourcen und verhindert so Zugriffskonflikte, wodurch jedoch Ausführungszeiten erhöht werden. Zudem werden durch verschiedene Regelungen für Cache- und Speicherzugriffe die Varianzen auf Kosten der durchschnittlichen Ausführungszeiten reduziert.

Die Anwendung solcher Techniken führt zu einer deterministischen und zulassungsfähigen Architektur, jedoch auf Kosten einer oftmals nicht mehr praktikablen Leistungsfähigkeit. Aktuell ist noch kein genereller Ansatz verfügbar, der der Zulassungs- und Leistungsfähigkeit in gleichem Maß gerecht wird. Herausforderung ist es, einen sinnvollen Kompromiss zwischen der beschriebenen Zulassungsfähigkeit sowie der gewünschten Performanz zu erzielen. Dazu sollten die Ansätze in hohem Maße querschnittlich sein, um die hohen Kosten der Nachweisführung auf mehrere Entwicklungsprojekte zu verteilen. Eine Standardisierung könnte in Form einer generischen Interpretation der Grundsätze von AMC 20-193 und AC 20-193 erfolgen, bisher sind jedoch solche Ansätze an der Verschiedenartigkeit der potenziellen Anwendungen gescheitert.

Die Anwendung einer neuen Betrachtungsweise, z. B. probabilistische Ansätze anstatt der starren Bestimmung einer „Worst Case Execution Time“ (WCET), kann einen Beitrag auf dem Weg zu einer Zertifizierung von Mehrkernprozessoren leisten. Dazu muss bereits in der Planungsphase eines Projekts eine Abstimmung mit der Zulassungsbehörde erfolgen. Weil dafür noch keine Erfahrungen vorliegen, sind aufwendige Untersuchungen und Nachweise im Vorfeld zu erbringen. Einer frühzeitigen Abstimmung mit den Zulassungsbehörden über Alternativansätze kommt somit eine ausgesprochen hohe Bedeutung zu. Auch ist der Weg zu zulassungsfähigen Ansätzen intensiv von der Industrie zu forcieren.

Zusammenfassung und Ausblick

Die festgelegte Definition 1 des spezialisierten „Avionik-Software Engineering“ ist aus den Randbedingungen der Avionik heraus notwendig. Die diskutierten Randbedingungen zeigen, dass zwar die Probleme im Software Engineering immer dieselben sind, aber die technischen, projekt-

[5] Siehe z. B. das Produkt Ada-C/C++ Changer, https://mapusoft.com/ada-to-c-changer/, zugegriffen am 11.06.2024.

basierten und juristischen Randbedingungen zumeist spezifische Lösungen und damit auch eine avionik-angepasste Werkzeuglandschaft erfordern.

Wiederverwendung hat das Potenzial einer schnelleren Entwicklung bzw. von geringeren Kosten [38–40]. Eine sinnvolle Wiederverwendung muss dabei alle Phasen des Entwicklungsprozesses umfassen [41]. Damit ist eine vollständige Integration der eingesetzten Werkzeugkette unverzichtbar. Eine Vorgehensweise hierzu wie in der SPES-Projektreihe [42–45] vorgestellt, kann dabei auch für die Avionik als Ideengeber wirken. Software in neueren Sprachen wie Rust kann hohe Performanz und gleichzeitig Typsicherheit bieten. Legacy Software muss einmalig in eine gewünschte Zielsprache übersetzt werden, um auch laufenden Projekten die Vorteilhaftigkeit von Programmiersprachen der Hauptströmung zu erschließen. Entsprechende Übersetzer existieren bereits. Durch ein industrielles Engagement in Foren werden die Verbreitung sowie Unterstützung der gewählten Programmiersprachen unterstützt.

Zusammenfassend können modellgetriebene Methoden mit einem formalen Backend entscheidende Beiträge zur Komplexitätsreduktion und Erhöhung der Zulassungskapazität leisten. Gekoppelt werden diese mit einem geeigneten Prozess, der mit Agilität die Stärken der Abstraktion und (De-)Komposition bespielt. Die Verwendung einer industrieverbreiteten Modellierungssprache wie der SysML v2 in Verbindung mit geeigneten und vollständig integrierten Werkzeugen mit Analysefähigkeiten ermöglichen signifikante Effektivitäts- und Effizienzsteigerungen.

Verfügbarkeit der Daten Im Rahmen dieser Arbeit wurden keine neuen Datensätze erzeugt oder analysiert. Die dargestellten Inhalte beruhen auf Fachwissen der Autoren sowie auf bereits veröffentlichten und zitierten Quellen.

Literatur

1. J. Marsden, A. Windisch, R. Mayo, J. Grossi, J. Villermin, L. Fabre, C. Aventini, ED-12C/DO-178C vs. Agile Manifesto – A Solution to Agile Development of Certifiable Avionics Systems, in ERTS 2018 (2018)
2. B. Annighoefer, M. Halle, A. Schweiger, M. Reich, C. Watkins, S. van der Leest, S. Harwarth, P. Deiber, Challenges and Ways Forward for Avionics Platforms and their Development in 2019, in 38th Digital Avionics SystemConference (DASC) (2019)
3. A. Brahmi, D. Delmas, M.H. Essoussi, F. Randimbivololona, A. Atki, T. Marie, Formalise to Automate: Deployment of a Safe and Cost-efficient Process for Avionics Software, in 9th European Congress on Embedded Real TimeSoftware and Systems (ERTS 2018) (Toulouse, France, 2018)
4. Judas PA, Prokop LE (2011) A Historical Compilation of Software Metrics with Applicability to NASA's Orion Spacecraft Flight Software Sizing. Innovations in Systems and Software Engineering 7(3):161–170
5. Booch G (1990) Object Oriented Design with Applications. Benjamin-Cummings Publishing Co., Inc., USA
6. Duvall P, Matyas S, Glover A (2007) Continuous Integration: Improving Software Quality and Reducing Risk. Pearson International
7. Eder S, Hauptmann B, Junker M, Juergens E, Vaas R, Prommer KH (2013) Did we Test our Changes? Assessing Alignment between Tests and Development in Practice, in 2013 8th International Workshop on Automation of Software Test (AST), pp. 107–110
8. Juergens E, Deißenböck F, Hummel B, Wagner S (o.J.) Do Code Clones Matter?, in 31st International Conference on Software Engineering (IEEE, Piscataway, NJ.), pp. 485–495
9. Broy M, Feilkas M, Herrmannsdoerfer M, Merenda S, Ratiu D (2010) Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. Proceedings of the IEEE 98(4):526–545
10. Moy Y, Ledinot E, Delseny H, Wiels V, Monate B (2013) Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. IEEE Software 30(3):50–57
11. Cousot P, Cousot R (1977) Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Association for Computing Machinery, New York, NY, USA), POPL '77, p. 238–252
12. Queille JP, Sifakis J (1982) Specification and Verification of Concurrent Systems in CESAR. In: Dezani-Ciancaglini M, Montanari U (Hrsg.) International Symposium on Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 337–351
13. Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G (2014) Comprehensive Formal Verification of an OS Microkernel. ACM Trans. Comput. Syst. 32(1)
14. Harel D, Rumpe B (2004) Meaningful Modeling: What's the Semantics of „Semantics". IEEE Computer Journal 37(10):64–72
15. Kausch H, Pfeiffer M, Raco D, Rumpe B, Schweiger A (2022) Correct and Sustainable Development Using Model-based Engineering and Formal Methods, in 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC) (IEEE)
16. Kausch H, Pfeiffer M, Raco D, Rath A, Rumpe B, Schweiger A (2023) A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System. In: Groher I, Vogel T (eds) Software Engineering 2023 Workshops. Gesellschaft für Informatik e.V., pp 169–188
17. Broy M, Rumpe B (2007) Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik-Spektrum 30(1):3–18
18. Kriebel S, Raco D, Rumpe B, Stüber S (2019) Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? CEUR Workshop Proceedings 2308:87–94
19. Kausch H, Michael J, Pfeiffer M, Raco D, Rumpe B, Schweiger A (2021) Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications, in Aerospace Europe Conference 2021 (AEC 2021) (Council of European Aerospace Societies (CEAS), 2021)
20. Kausch H, Pfeiffer M, Raco D, Rumpe B (2021) Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams, in Proceedings of the Software Engineering 2021 Satellite Events, vol. 2814, ed. by S. Götz, L. Linsbauer, I. Schaefer, A. Wortmann (CEUR, 2021)
21. Bürger JC, Kausch H, Raco D, Ringert JO, Rumpe B, Stüber S, Wiartalla M (2020) Towards an Isabelle Theory for distributed, interactive systems - the untimed case. Aachener Informatik Berichte, Software Engineering, vol 45. Shaker Verlag
22. Whittle J, Hutchinson J, Rouncefield M (2014) The State of Practice in Model-Driven Engineering. IEEE Software 31(3):79–85
23. Meyer B (2002) Design by Contract. Prentice Hall, Upper Saddle River

24. Brenner W, Broy M, Leimeister JM (2017) Auf dem Weg zu einer Informatik neuer Prägung in Wissenschaft, Studium und Wirtschaft. Informatik Sprektrum 40
25. Boehm BW (1976) Software Engineering. IEEE Transactions on Computers C-25(12):1226–1241
26. Baziuk W (1995) BNR/NORTEL: Path to Improve Product Quality, Reliability and Customer Satisfaction, in Proceedings of Sixth International Symposiumon Software Reliability Engineering. ISSRE'95, pp. 256–262
27. Fieber F, Huhn M, Rumpe B (2008) Modellqualität als Indikator für Softwarequalität: eine Taxonomie. Informatik-Spektrum 31(5):408–424
28. Kausch H, Pfeiffer M, Raco D, Rumpe B, Schweiger A (2024) Enhancing System-model Quality: Evaluation of the MontiBelle Approach with the Avionics Case Study on a Data Link Uplink Feed System. In: Avionics Systems and Software Engineering Workshop of the Software Engineering 2024 - Companion Proceedings (AvioSE). Gesellschaft für Informatik e.V., pp 119–138
29. Rushby J (2008) Automated Test Generation and Verified Software (Springer Berlin Heidelberg, Berlin, Heidelberg), pp. 161–172
30. Kausch H, Pfeiffer M, Raco D, Rumpe B (2020) An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems, in Combined Proceedings of the Workshops at Software Engineering 2020, vol. 2581, ed. by R. Hebig, R. Heinrich (CEUR Workshop Proceedings)
31. Kausch H, Pfeiffer M, Raco D, Rumpe B (2020) MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety, in AIAA Scitech 2020 Forum (American Institute of Aeronautics and Astronautics, 2020)
32. Brucker AD, Wolff B (2012) On Theorem Prover-based Testing. Formal Aspects of Computing 25:683–721
33. Stachowiak H (1973) Allgemeine Modelltheorie. Springer
34. Paech B, Rumpe B (1994) A new Concept of Refinement used for Behaviour Modelling with Automata. Proceedings of the Industrial Benefit of Formal Methods (FME'94) LNCS 873:154–174
35. Grosu R, Klein C, Rumpe B, Broy M (1996) State Transition Diagrams. Technischer Bericht., Technische Universität München.
36. Balzert H (1998) Software-Management, Software-Qualitätssicherung,Unternehmensmodellierung. Spektrum Akademischer Verlag
37. Lockhead Martin Corporation (2005) Joint Strike Fighter – Air Vehicle C++Coding Standards for the System Development and Demonstration Program.
38. Clements P, Northrop L (2002) Software Product Lines. Addison-Wesley, Boston
39. Haber A, Rendel H, Rumpe B, Schaefer I (2012) Evolving Delta-oriented Software Product Line Architectures. In: Large-Scale Complex IT Systems.Development, Operation and Management, 17th Monterey Workshop 2012. LNCS, Vol. 7539. Springer, pp 183–208
40. Pohl K, Böckle G, Van Der Linden F (2005) Software Product Line Engineering: Foundations, Principles, and Techniques, Vol. 1. Springer
41. Keswani R, Joshi S, Jatain A (2014) Software Reuse in Practice, in 2014 Fourth International Conference on Advanced Computing & Communication Technologies, pp. 159–162
42. Pohl K, Hönninger H, Achatz R, Broy M (eds) (2012) Model-Based Engineering of Embedded Systems. Springer, Berlin, Heidelberg
43. Pohl K, Hönninger H, Daembkes H, Broy M (eds) (2016) Advanced Model-Based Engineering of Embedded Systems. Springer, Cham
44. Böhm W, Broy M, Klein C, Pohl K, Rumpe B, Schröck S (eds) (2021) Model-Based Engineering of Collaborative Embedded Systems. Springer
45. SpesML-Projektseite. https://spesml.github.io