Integration von Informationen aus Testprozessen im Supply Chain Management

Vom Fachbereich Produktionstechnik

der

UNIVERSITÄT BREMEN

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

Dr.-Ing. –

genehmigte

DISSERTATION

von

Dipl.-Inf. Marco Franke

Hauptreferent: Prof. Dr.-Ing. habil. Klaus-Dieter Thoben

Korreferent: Prof. Dr. rer. nat. Otthein Herzog

Tag der mündlichen Prüfung: 05.08.2024

Zusammenfassung

Die Entwicklung von komplexen mechatronischen Systemen - wie Flugzeuge, Autos oder Züge - erfolgt in einer Zulieferpyramide. Im Rahmen der Produktentwicklung werden ein Testprozess beim Originalgerätehersteller (OEM) und mehrere Testprozesse bei den Lieferanten durchgeführt. Das Ziel des Verbunds aus Testprozessen ist die Verifikation des mechatronischen Systems.

Die vorliegende Arbeit betrachtet Funktionstest. Das Hauptziel eines jeden Testprozesses ist die Überprüfung der Funktionalität, der Zuverlässigkeit und der Betriebssicherheit des zu testenden Produktes anhand von Anforderungen. Hierfür werden Testfälle spezifiziert, implementiert, ausgeführt und ihre Ergebnisse als Nachweis für die Verifikation eingesetzt. Zusätzlich werden Testfälle zwischen Testprozessen ausgetauscht, wenn es zu unterschiedlichen Ergebnissen in den Testprozessen der Zulieferpyramide kommt. Dementsprechend nehmen Testfälle eine zentrale Rolle als Informationsträger in den Testprozessen ein.

Der Einsatz von Testfällen als Informationsträger wird durch die Heterogenität in den Testprozessen innerhalb der Zulieferkette erschwert. Jeder Testprozess verfolgt spezifische Zielsetzungen. Dies führt je nach Fall zur Auswahl unterschiedlicher Testskriptsprachen und Prüfstände. Deswegen werden Testfälle in der Zulieferpyramide in unterschiedlichen Testskriptsprachen implementiert, obwohl sie ähnliche Sachverhalte beschreiben. Diese Ausgangssituation reduziert die Bedeutung der Testfälle vom Informationsträger zum Datenträger und verursacht einen Informationsverlust beim Austausch der Testfälle in den Testprozessen.

Das Ziel dieser Arbeit ist die Erforschung der informationstechnischen Grundlagen für die Integration von Testfällen als Informationsträger in der Zulieferpyramide. Der Ansatz fokussiert auf die Interoperabilität von Testfällen. Die Ausgangssituation ist, dass zwei Testprozesse für die Fehlersuche ihre Testfälle in unterschiedlichen Testskriptsprachen austauschen müssen. Das hierfür entwickelte Konzept ist die kollaborative Fehlersuche und ermöglicht sowohl den interoperablen Austausch von Testfällen als auch die Nutzung als Informationen für die Fehlersuche.

Die kollaborative Fehlersuche benötigt als Vorarbeit die Identifizierung von relevanten Informationen, welche in den Testfällen enthalten sind. Zusätzlich wird die Heterogenität der Testskriptsprachen herausgearbeitet, um die notwendigen Datenintegrationsansätze für die Interoperabilität von Testfällen zu identifizieren.

Das Ergebnis sind drei Fehlersuchmethoden als Bestandteil der Fehlersuche, mit dessen Hilfe die Interoperabilität von Testfällen für den direkten Austausch von Testfällen und für die Suche nach Informationen innerhalb von Testfällen hergestellt wurde. Für beide Fälle wurden Informationsmodelle für Testfälle und entsprechende Datenintegrationslösungen entwickelt. Die Evaluation wurde in den Testprozessen für Hochauftriebsklappen und Kabinentüren von Flugzeugen durchgeführt.

Abstract

The development of complex mechatronic systems such as aircraft, cars or trains takes place in a supplier pyramid. As part of product development, a test process is carried out at the original equipment manufacturer (OEM) and several test processes are carried out at the supplier. The aim of the network of test processes is to verify the mechatronic system.

This thesis considers functional testing. The main goal of each test process is to check the functionality, reliability, and operational safety of the product to be tested based on requirements. For this purpose, test cases are specified, implemented, executed and their results used as evidence for verification. In addition, test cases are exchanged between test processes if there are different results in the test processes of the supplier pyramid. Accordingly, test cases play a significant role as information carriers in the test processes.

The challenge when using test cases as information carriers is their heterogeneity between the test processes of the supplier pyramid. Each test process pursues specific goals, which means that the test script languages and test benches used are specific. For this reason, test cases in the supplier pyramid are implemented in different test script languages, although they describe similar issues. This initial situation reduces the importance of the test cases from the information carrier to the data carrier and causes a loss of information when the test cases are exchanged in the test processes.

The aim of this work is to research the information technology bases for the integration of test cases as information carriers in the supplier pyramid. The chosen approach focuses on the interoperability of test cases. The initial situation is that two test processes must exchange their test cases in different test script languages for troubleshooting. The concept developed for this is collaborative troubleshooting and describes both the interoperable exchange of test cases and their use as information for troubleshooting.

As a preliminary work, the developed collaborative troubleshooting requires the identification of the relevant information contained in the test cases. In addition, the heterogeneity of test script languages is explored to identify the necessary data integration approaches for test case interoperability.

As a result, three test case debugging methods have been developed, which have been used to establish test case interoperability for the direct exchange of test cases and for the search for information within test cases. For both cases, information views for test cases and corresponding data integration solutions were developed. The evaluation was carried out in the test processes for high-lift flaps and cabin doors on aircraft.

Danksagung
Zunächst möchte ich mich bei meinem Doktorvater, Herrn Prof. Dr.-Ing. Klaus-Dieter Thoben, für die Möglichkeit bedanken, an meiner Dissertation zu forschen. Ohne seine kontinuierliche Betreuung, Unterstützung, Motivation und das Korrekturlesen der Dissertation wäre die Fertigstellung dieser Arbeit nicht möglich gewesen. In den vielen Gesprächen, die wir in den letzten Jahren geführt haben, hat er viele herausfordernde und inspirierende Fragen gestellt, wodurch das Verständnis für die Problemstellung und die darauf basierende Lösung projekt- und domänenübergreifend reifen konnte. Prof. Dr. Otthein Herzog bin ich sehr dankbar, dass er sich bereit erklärt hat, mein Zweitgutachter in der Promotionskommission zu sein.

Mein Dank gilt meinen Kollegen vom Bremer Institut für Produktion und Logistik GmbH (BIBA) für ihre kontinuierliche Unterstützung. Mein Dank gilt Quan Deng, Konstantin Klein, Stefan Wellsandt und Meike Wienholdt für ihre Expertise.

Außerdem möchte ich mich bei meinen Freunden Martin Doliwa, Matthias Erhardt und Hendrik Orlovius für ihre Hilfe bedanken.

Vor allem bin ich meiner geliebten Frau, meinem geliebten Sohn und meinen geliebten Schwestern zutiefst dankbar, für ihre bedingungslose Unterstützung, Ermutigung und unerschütterliche Liebe. Abschließend möchte ich diese Dissertation meinem Vater widmen, dessen Leitspruch „Das Ding an sich gibt es nicht“ auch eine gute Zusammenfassung meiner Problemstellung sein könnte.

Tabellenverzeichnis

Abbildungsverzeichnis

Akronyme & Definitionen

AGILE-VT	Agile Virtual Testing: Harmonisierung von Testumgebungen (AGILE-VT) ist ein Forschungsprojekt für die horizontale Durchgängigkeit von Testprozessen zum Zwecke der Optimierung beim funktionalen Testen von Luftfahrtzeugen.
API	Application Programming Interface (API) ist eine Programmierschnittstelle.
APPU	Asymmetry Position Pick Off Unit APU (APPU) meldet Positions- und Winkelasymmetrien.
ARINC	Steht stellvertretend für die verschiedenen Aeronautical Radio Incorporated (ARINC) Standards, welche Standards für Bussysteme in der Luftfahrt sind.
BOL	Beginning-of-Life (BOL) ist eine Phase im Produktlebenszyklusmanagement.
CCDL	Check Case Definition Language (CCDL) ist eine Testskriptsprache und wird von Razorcat vertrieben.
CIDS	Cabin Intercommunication Data System (CIDS) ist das Kabinensteuerungssystem von Flugzeugen des Herstellers Airbus.
DSL	Domänenspezifische Sprache (DSL) ist eine Programmiersprache oder eine ausführbare Spezifikationssprache, welche auf eine Domäne zugeschnitten ist.
EOL	End-of-Life (EOL) ist eine Phase im Produktlebenszyklusmanagement.
GAV	Global as View (GAV) ist eine Art der Datenintegration, in der die globale Sicht die Referenz für Abfrage und Transformation sind.
HIL	Hardware in the Loop (HIL) ist ein Testverfahren, in dem ein physisches Testobjekt in einer simulierten Umgebung getestet wird.
IDSA	International Data Spaces (IDSA) ist eine Initiative für den domänenübergreifenden Austausch und Handel mit Daten.
IDS-RAM	Reference Architecture Model (IDS-RAM) ist die Softwarearchitektur des IDSA.

IO	Ein-/Ausgabe (I/O) definiert die Ein- Und die Ausgabe eines Informationssystems.
IPR	Intellectual Property Rights (IPR) ist das Urheberrecht und regelt den Schutz des geistigen Eigentums.
LAV	Local as View (LAV) ist eine Art der Datenintegration, in der die lokalen Sichten die Referenz für Abfrage und Transformation sind.
MBSE	Modellbasiertes Systems Engineering (MBSE) ist ein modellbasierter Ansatz der Produktentwicklung.
MBT	Modellbasiertes Testen ist ein modellbasierter Ansatz der Testfallgenerierung als Bestandteil der Produktentwicklung.
MOL	Middle-of-Life (MOL) ist eine Phase im Produktlebenszyklusmanagement.
OCL	Object Constraint Language (OCL) ist eine Sprache, um Restriktionen für UML-Modelle zu definieren.
OEM	Originalgerätehersteller (OEM) bezeichnet den Hersteller eines Produktes und steht stellvertretend als OEM an der Spitze der Zulieferpyramide, deren Rest durch Lieferketten gebildet wird.
RCP	Rich Client Platform (RCP) ist ein Framework für die Entwicklung von Desktop-Anwendungen.
PKW	Personenkraftwagen (PKW) ist ein Fahrzeug, welches für den Straßenverkehr zugelassen ist.
PLM	Das Produktlebenszyklus-Management (PLM) ist das Management eines Produktes von der Entwicklung bis zur Entsorgung.
RTT	Real Time Testing (RTT) ist eine auf Python basierte Testskriptsprache für die Echtzeittestautomatisierung.
SCM	Supply Chain Management (SCM) ist das Management von Lieferketten.
SCXML	State Chart XML (SCXML): State Machine Notation for Control Abstraction ist ein XML basierter Standard für die Darstellung von Zustandsübergangsdiagramme.

SFCC	Slat Flap Control Computer (SFCC) ist ein Flugsteuerungs-Computer für die Steuerung der Klappen am Flugzeug.
SPI	Serial Peripheral Interface (SPI) ist ein serieller Datenbus und dient der Kommunikation zwischen elektronischen Bauteilen.
SRD	System Requirement Document (SRD) ist ein Dokument mit System und Systemelement Anforderungen.
SUT	System Under Test (SUT) bezeichnet das zu testende System.
STEVE	System-Technik und Virtuelle Erprobung (STEVE) ist ein Forschungsprojekt für die die Entwicklung eines umweltfreundlichen Luftverkehrssystems.
TTCN-3	Testing and Test Control Notation, (TTCN-3) ist Testskriptsprache, welche für kommunikationsbasierte Systeme zugeschnitten ist.
UML	Unified Modeling Language ist eine Modellierungssprache für die Spezifikation von Software.
W3C	World Wide Web Consortium (W3C) ist ein Standardisierungsgremium für das World Wide Web.
WTB	Wing Tip Brakes (WTB) ist eine Bremse und wird beispielsweise im Hochauftriebssystem von Flugzeugen eingesetzt.
XML	Extensible Markup Language (XML) ist eine maschinenlesbare Auszeichnungssprache mit dessen Hilfe Texte hierarchisch formatiert werden können.

1 Einleitung

Hersteller agieren im globalen Wettbewerb und verbessern ihre Wettbewerbsposition unter anderem sowohl durch die Integration neuer Technologien in ihre Produkte als auch durch die Verteilung von Entwicklungsaktivitäten auf Zulieferpyramiden. Komplexe mechatronische Produkte wie ein Auto oder ein Flugzeug werden bereits heute in Zulieferpyramiden entwickelt. Dabei übernehmen sowohl der OEM als auch die Lieferanten Entwicklungs- und Testaufgaben. Ein Beispiel für ein solches mechatronisches System ist das Hochauftriebssystem eines Flugzeugs. Es ist in Abbildung 1 vereinfacht dargestellt und wird im Verlauf der Arbeit als wiederkehrendes Beispiel genutzt.

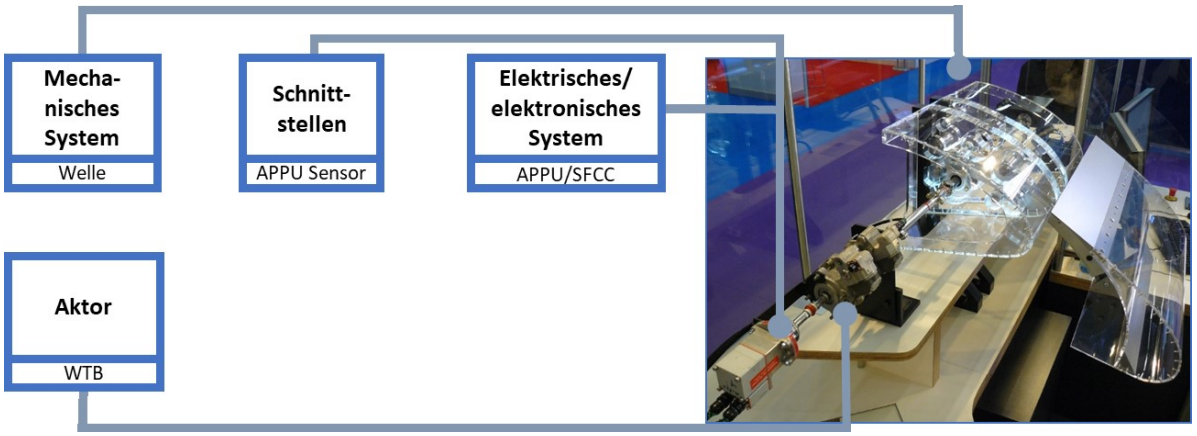

Abbildung 1: Demonstrator eines Hochauftriebssystems (Vorflügel) von einem A320 [Franke und Thoben 2022]

Das Vorflügelsystem ist eine Hochauftriebsvorrichtung, die normalerweise am Flügel angebracht ist. Der Vorflügel (Slat) hat die Aufgabe, den Anstellwinkel des Flügels in Abhängigkeit von der Flugphase zu verändern. Dazu kann die Klappe ein- oder ausgefahren werden. Dafür ist bereits ein Rechner (Informationsverarbeitungseinheit) zuständig, der die Bewegung über ein Kommunikationsmedium steuert. Im Beispiel erfolgt die Kommunikation über den ARINC 429 Bus und der Rechner ist ein Slat Flap Control Computer (SFCC).

Die Zulieferpyramide besteht aus einem OEM und mehreren Lieferanten. Innerhalb einer Zulieferpyramide wird die Entwicklung komplexer mechatronischer Systeme ermöglicht. Hierbei sind Lieferanten zunehmend voneinander abhängig und durch ihre vertikale Zusammenarbeit miteinander vernetzt [Pan et al. 2021]. Die Abhängigkeit entsteht zum einen durch die iterative Spezialisierung der Anforderungen vom OEM zu den Lieferanten. Zum anderen entsteht die Abhängigkeit durch die iterative Integration der Produkte beginnend vom Teil über die Komponenten und

Module zum Produkt beim OEM. Im Rahmen dieser Entwicklungsaktivitäten entsteht in jedem Unternehmen spezifisches Entwicklungswissen und eine eigene Sichtweise auf das Produkt. Dieses Wissen liegt bisher nur in Form von Daten vor und der Austausch würde innerhalb der Zulieferpyramide nur über Datenträger erfolgen. Die Nutzung der Daten erfordert eine nachgelagerte Datenintegration, um auf die Informationen zugreifen zu können. Hierbei entspricht die Anzahl an involvierten Datenquellen laut [Reeve 2013] mehrere tausend bei mittleren und großen Unternehmen.

Der Austausch von Datenträgern innerhalb der Zulieferpyramide hat einen großen Vorteil. So können die unterschiedlichen Perspektiven der Lieferanten und des OEM zu einer Gesamtsicht auf das Produkt aggregiert werden. Dies ermöglicht den Informationsaustausch mit Hilfe eines Informationsträgers für eine konkrete Aufgabenstellung. Dieser Bedarf besteht bereits heute und wird mit zunehmender Komplexität der Produkte weiter steigen. Der Aufbau einer ganzheitlichen Sicht ist nur möglich, wenn die Interoperabilität des Entwicklungswissens in der Zulieferpyramide erreicht wird. IEEE definiert Interoperabilität als die Fähigkeit von zwei oder mehr Systemen sowohl Informationen austauschen zu können als auch die ausgetauschten Informationen zu nutzen [Geraci et al. 1991]. Die Dissertation ermöglicht die Interoperabilität von Informationen aus den Testprozessen.

Die Interoperabilität kann mit der Methode der semantischen Datenintegration den Wandel von der Repräsentation des Entwicklungswissens als Daten zu einer Menge von Informationen einleiten. Sobald diese Herausforderung gemeistert ist, kann die Aggregation der Informationen zu einer ganzheitlichen Sicht erfolgen. Somit kann die Voraussetzung für ein breites Spektrum an datengetriebenen Diensten geschaffen werden, welche beispielsweise die Testprozesse in der Entwicklung unterstützten.

Im Rahmen der Dissertation wird der Austausch von Informationen und von interoperablen Testfällen für eine Fehlersuche in der Zulieferpyramide erforscht. Im Folgenden wird der Mehrwert des Informationsaustausches in der Zulieferpyramide begründet. Anschließend wird auf die Notwendigkeit und die Herausforderung im Austausch von Testwissen eingegangen.

1.1 Motivation

Komplexe mechatronische Systeme bestehen aus einem Verbund von Teilsystemen [Bolton 1997], welche von verschiedenen Lieferanten entwickelt und beim OEM zu einem Gesamtsystem integriert werden. Dabei wird für jedes Teil, Komponente und Modul ein Testprozess durchlaufen. Anschließend wird für ihre Integration wieder ein Testprozess durchgeführt. Das Ziel eines Testprozesses ist die Qualitätssicherung. Hierfür wird der Nachweis erbracht, dass sich ein System under Test (SUT) entsprechend den Anforderungen verhält [Thoben et al. 2011]. Es treten Fälle auf,

in denen die Testprozesse für die Integration fehlschlagen. In solchen Fällen müssen unterschiedliche Lieferanten vertikal in der Zulieferpyramide zusammenarbeiten. Das Ziel der Zusammenarbeit ist es, durch eine kollaborativen Fehlersuche herauszufinden, warum der Testprozess fehlgeschlagen ist. Dazu müssen die Lieferanten Informationen und keine Daten austauschen. Derzeit können sie nur Daten in ihren spezifischen Datenformaten und *Testskriptsprachen* austauschen, was zu Informationsverlusten führt.

Hierbei ist eine *Testskriptsprache* im Kontext der Arbeit eine domänenspezifische Sprache (DSL), mit der Testfälle als Testskripte für die Automatisierung von Tests ausgeführt werden können. Eine DSL ist hierbei eine Programmiersprache oder eine ausführbare Spezifikationssprache, die auf eine Domäne zugeschnitten ist [van Deursen et al. 2000].

Im Kontext der Testprozesse sind die auszutauschenden Informationen in den Testfällen enthalten. Testfälle müssen daher interoperabel innerhalb der Zulieferpyramide ausgetauscht werden. Nur dann ist es möglich, den „fremden“ und den „eigenen“ Testfall gleich zu behandeln und dementsprechend die Gründe für das Scheitern des Testprozesses zu finden. Eine wichtige Voraussetzung hierfür ist, dass der „fremde“ Testfall in den eigenen Testprozess integriert werden kann. Die Fähigkeit, Testfälle in der Zulieferpyramide interoperabel austauschen zu können, ermöglicht auch deren Aggregation. Das Ergebnis wäre, sowohl die verschiedenen Sichten auf das Produkt als SUT zu kennen, als auch die Schnittmengen der verschiedenen Sichten als konsolidierte Informationen zu speichern und darauf aufbauend datengetriebene Dienste anzubieten. Beispielsweise können statische Analysen als Teil der Fehlersuche in Testprozesse integriert werden oder das Sollverhalten für die Konfiguration der prädikativen Wartung genutzt werden [Franke et al. 2013; Thoben et al. 2018]. Langfristig könnten digitale Zwillinge z.B. dazu dienen, Betriebsdaten anhand des in Testfällen enthaltenen Sollverhaltens besser auszuwerten oder die in Testfällen enthaltenen Stimuli für die Produktentwicklung zur Optimierung des Designs zu nutzen.

1.2 Problembeschreibung

Das Produktlebenszyklus-Management (PLM) [Kiritsis et al. 2003] ist mit dem Supply Chain Management verknüpft und definiert die Integration verschiedener Arten von Aktivitäten, aus technischer, organisatorischer und betriebswirtschaftlicher Sicht. Diese Aktivitäten werden ausschließlich durch den Hersteller oder aber mithilfe einer Zulieferpyramide erbracht. Dementsprechend werden Aktivitäten auch von Mitarbeitern aus unterschiedlichen Unternehmen über den gesamten Lebenszyklus von Produkten beigesteuert.

Bei der Entwicklung komplexer mechatronischer Produkte, wie beispielsweise Flugzeuge oder Autos, erfolgt bereits die Produktentwicklung über eine Zulieferpyramide [Schulte 2013]. Die zugrundeliegende Zulieferpyramide fokussiert sich sowohl auf die Entwicklung von Teilen als auch auf die Entwicklung von Komponenten oder Modulen. Dabei sind die Lieferanten innerhalb der Zulieferpyramiden zunehmend voneinander abhängig und durch ihre vertikale Zusammenarbeit auch miteinander vernetzt [Pan et al. 2021]. Innerhalb der vertikalen Zusammenarbeit einer Zulieferpyramide sind mehrere Testprozesse hinsichtlich ihrer Ziele, der verwendeten *Testskriptsprachen* und der eingesetzten Testsysteme heterogen. Diese Heterogenität wird zu einem Problem, sobald Testfälle im Rahmen der Zusammenarbeit zwischen Testprozessen ausgetauscht werden müssen.

Eine Teilmenge der Testfälle aus der Zulieferpyramide weist besondere Eigenschaften auf. Sie werden von unterschiedlichen Lieferanten implementiert und testen die gleiche Funktion unter ähnlichen Anforderungen, verwenden aber unterschiedliche *Testskriptsprachen*. Dementsprechend haben die Testfälle eine ähnliche Semantik, aber eine unterschiedliche Syntax. Die Semantik eines Testfalls enthält Informationen darüber, wie das mechatronische System stimuliert und wie es getestet wird. Diese Informationen sind für die gesamte Zulieferpyramide wertvoll und erleichtern die Zusammenarbeit in den Testprozessen. Lieferanten könnten sie mit dem derzeitigen Stand der Technik als Ressourcen austauschen. Zu diesem Zweck können Methoden des Supply-Chain-Managements genutzt werden, die stärker auf die Digitalisierung setzen. Aktuelle Ansätze rund um digitale Zwillinge [Barykin et al. 2020; Wang et al. 2022] ‚Digital Threads [Deng et al. 2021] und rund um die Basistechnologien für die Interoperabilität wirken sich positiv auf die Zulieferpyramide aus [Zhu et al. 2022].

Trotz der aufgezählten Vermittlungsfähigkeiten ist der ursprüngliche Testfall weder interoperabel noch für die kollaborative Fehlersuche im Kontext einer Zusammenarbeit integrierbar. Dies liegt daran, dass eine Testskriptsprache und ein Testsystem eng verzahnt sind und dieses Tupel zuliefererspezifisch ist. So verwenden die Lieferanten und der OEM unterschiedliche *Testskriptsprachen* und können Testfälle nur in ihrer Testskriptsprache interpretieren und ausführen. Die Herausforderung besteht darin, dass der Austausch und die anschließende Ausführung von Testfällen für die kollaborative Fehlersuche erforderlich ist. Die manuelle Sichtung durch die Extraktion der relevanten Informationen aus dem „fremden“ Testfall ist keine geeignete Lösung, da jeder Testprozess nur mit seiner eigenen Testskriptsprache vertraut ist. Dies führt zu Informationsverlusten in den Testprozessen. Daher ist die Interoperabilität der Testfälle erforderlich, die mit dem derzeitigen Stand der Technik nicht gegeben ist.

IEEE definiert Interoperabilität als die Fähigkeit von zwei oder mehr Systemen sowohl Informationen austauschen zu können als auch die ausgetauschten Informatio-

nen zu nutzen [Geraci et al. 1991]. Die Anwendung dieser Definition auf den Austausch von Testfällen bedeutet, dass der Inhalt der Testfälle bei einem Lieferanten als Informationen extrahiert und als Informationen bei einem anderen Lieferanten integriert werden können. Integration bedeutet hier, dass der Lieferant die Informationen als Testfall in seine Testskriptsprache übersetzen und anschließend ausführen kann. Falls dies nicht möglich ist, sollten die Informationen anderweitig für die Fehlersuche nutzbar sein.

Um den Austausch der Informationen im Rahmen einer testprozessübergreifenden Aktivität zu ermöglichen, muss jedes Element eines Testfalls auf die Ebene *semantic understanding* nach [Ören et al. 2007] gehoben werden, um die Bedeutung des Elements zu kennen und damit eine Vergleichbarkeit mit anderen Elementen aus anderen Datenquellen herstellen zu können. Die Aufbereitung erfordert die Überwindung aller Ebenen der Datenintegrationskonflikte nach [Goh 1997; Wache 2003]. Sobald die Ebene *semantic understanding* erreicht wurde, müssen für die Übertragung der Information die Qualitätskriterien nach [Dong und Naumann 2009] wie Vollständigkeit und Korrektheit berücksichtigt werden. Hierbei ist insbesondere der Kontext der Information zu berücksichtigen, der durch den Testprozess bestimmt wird. Die zu testenden Anforderungen und die Eigenschaften des Testsystems sind Beispiele für Faktoren, welche die Schritte innerhalb des Testprozesses beeinflussen.

Die interoperable Darstellung von Testfällen ist mit den existierenden Informationsmodellen für Informationen (durch Ontologien) oder für Testfälle (durch Testmodelle) nicht sinnvoll. Darüber hinaus gibt es keine Transformationsansätze, um Informationen aus Testfällen zu extrahieren und zu nutzen. Ein Nutzungsszenario ist die Rückübersetzung, welche zu der Gruppe der Transformationen gehört, der Testfälle in die *Testskriptsprachen* der Zielprozesse.

1.3 Untersuchungsrahmen

Die vorliegende Arbeit ist thematisch in die Forschungsprojekte STEVE und AGILE-VT eingebunden. Der Autor ist hier in seiner Funktion als projektverantwortlicher wissenschaftlicher Mitarbeiter des BIBA - Bremer Institut für Produktion und Logistik GmbH tätig.

Im Rahmen der Forschungsprojekte wurde die Problemstellung anhand des Endanwenders Airbus konkretisiert und hierfür spezifische Ansätze und Funktionsmuster entwickelt. Dementsprechend sind die in dieser Arbeit betrachteten Testprozesse einschließlich der *Testskriptsprachen* an Airbus angelehnt und mit Hilfe der Literatur verallgemeinert.

1.4 Forschungsfrage und Zielsetzung

Die Entwicklung komplexer mechatronischer Systeme benötigt testprozessübergreifende Aktivitäten als Bestandteil der involvierten Testprozesse. Im Rahmen der testprozessübergreifenden Aktivitäten erfolgt die Zusammenarbeit und es entstehen signifikante Datenintegrationsaufwände oder Informationsverluste.

Im Rahmen der Dissertation soll die Forschungsfrage beantwortet werden, **ob mithilfe des Austausches von Informationen aus interoperablen Testfällen der Datenintegrationsaufwand reduziert und der Nutzen der ausgetauschten Informationen für die testprozessübergreifenden Aktivitäten erhöht werden kann**.

Die Forschungsfrage soll anhand der kollaborativen Fehlersuche in abhängigen Testprozessen untersucht werden. Dementsprechend ist das Ziel der Arbeit, die informationstechnischen Grundlagen für die angestrebte kollaborative Fehlersuche zu legen. Der gewählte Ansatz für die Erreichung des Ziels ist die Interoperabilität von Testfällen. Der Nutzen des interoperablen Austausches von Testfällen und Informationen soll für die unterstützenden Methoden eines **Transcompilers** und einer **Mustersuche** als Bestandteil der Fehlersuche erforscht werden. Hieraus ergeben sich abgeleitete Forschungsfragen für die Fehlersuche:

1. Generelle Forschungsfragen
    a. Sind die Testaktivitäten der Testprozesse aus der Zulieferpyramide vergleichbar, um testprozessübergreifende Aktivitäten zwischen den Testprozessen zu ermöglichen?
    b. Welche Fehlersuchmethoden sind notwendig, um den Grund für die unterschiedlichen Ergebnisse der Testprozesse zu finden?
2. Informationstechnische Forschungsfragen
    a. Welche Informationen beinhaltet ein Testfall in den Testprozessen der Zulieferpyramide?
    b. Welche *Testskriptsprachen* werden für den „Hardware in the Loop“ (HIL)-Test eingesetzt und inwiefern sind sie zueinander kompatibel?
    c. Welches Schema muss ein Informationsmodell besitzen, das auf die Semantik und nicht auf die Syntax von Testfällen fokussiert?
    d. Welche Datenintegrationsprobleme müssen für die Interoperabilität von Testfällen gelöst werden?

1.5 Vorgehen

Die Vorgehensweise der Arbeit orientiert sich am Prozessmodell der Design Science Research Methodology (DSRM), welches von [Peffers et al. 2007] entwickelt

wurde. DSRM zielt hierbei auf Informationssysteme ab und ermöglicht die Entwicklung von Artefakten, wie beispielsweise Methoden und Modellen. Das in dieser Arbeit fokussierte Artefakt ist die kollaborative Fehlersuche als Bestandteil einer testprozessübergreifenden Aktivität, einschließlich der darin beinhalteten Informationsmodelle. Eine für diese Arbeit stark vereinfachte Version vom DSRM-Prozessmodell ist in der Abbildung 2 zu sehen und wird im Detail vorgestellt.

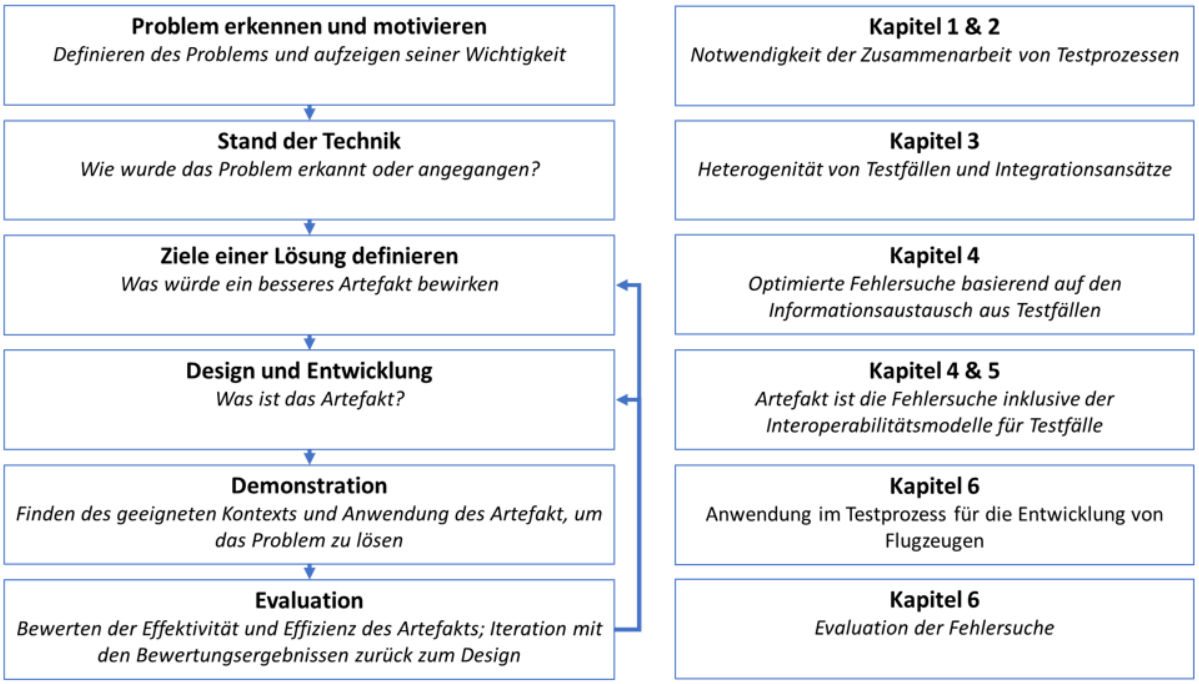

Abbildung 2: Angepasstes DSRM-Prozessmodell für das Vorgehen in der Dissertation

Das **Kapitel 2** leitet die Grundlagen für die Dissertation ein. Hierfür wird das Produktlebenszyklusmanagement im Unterkapitel 2.1 kurz eingeführt und der äußere Rahmen für die Testprozesse gesteckt. Anschließend werden in den Unterkapiteln 2.2 und 2.3 die Testprozesse detailliert vorgestellt. Hierbei wird insbesondere auf deren Bedeutung für die Entwicklung mechatronischer Systeme und ihrer Heterogenität eingegangen. Die sich daraus ergebene Vielfalt wird im Unterkapitel 2.4 präsentiert. Abschließend wird auf die Notwendigkeit und die sich ergebenen Herausforderungen der kollaborativen Fehlersuche als testprozessübergreifende Aktivität zwischen den Testprozessen im Unterkapitel 2.5 eingegangen.

Das **Kapitel 3** fokussiert sich auf den Stand der Technik bezüglich der Informationen, *Testskriptsprachen* und Integrationsansätze, welche für die kollaborative Fehlersuche in Testprozessen relevant sind. Hierfür wird im Unterkapitel 3.1 ein Überblick über verfügbare Informationen in den Testprozessen gegeben. Testfälle werden in diesem Zusammenhang als ein essenzieller Datenträger identifiziert.

Die Heterogenität der eingesetzten *Testskriptsprachen* wird im Unterkapitel 3.2 detailliert vorgestellt. Anschließend werden die Herausforderungen für die Extraktion der Informationen aus *Testskriptsprachen* im Unterkapitel 3.3 zusammengefasst. Unterkapitel 3.4 legt die zu lösende Problemdefinition informationstechnisch dar.

Das **Kapitel 4** stellt das Konzept vor, was das Ziel der Lösung ist. Das Ziel ist die Optimierung der testprozessübergreifenden Aktivitäten von Testprozessen. Hierfür wird im Unterkapitel 4.1 die entwickelte kollaborative Fehlersuche vorgestellt, welche den Austausch und die Anwendung von Testfällen als Informationsträger ermöglicht. Der hierfür benötigte Interoperabilitätsansatz wird im Unterkapitel 4.2 vorgestellt. Anschließend werden im Unterkapitel 4.3 die drei Fehlersuchmethoden der kollaborativen Fehlersuche im Detail dargestellt. Hierbei wird jede Fehlersuchmethode als ein Anwendungsfall beschrieben.

Das **Kapitel 5** fokussiert auf die Spezifikation und Implementierung der kollaborativen Fehlersuche. Hierfür wird zunächst im Unterkapitel 5.1 der Schritt 1 der kollaborativen Fehlersuche vorgestellt. Darauf aufbauend werden im Unterkapitel 5.2 die drei Fehlersuchmethoden des Schrittes 2 der kollaborativen Fehlersuche präsentiert.
In jedem Unterkapitel wird das entwickelte Informationsmodell und die benötigte Transformation von dem Datenträger in den Informationsträger detailliert beschrieben. Des Weiteren werden für jede der drei Fehlersuchmethoden Details zu der Implementierung dargelegt.

Das **Kapitel 6** setzt sich mit der Anwendung der drei Fehlersuchmethoden der Fehlersuche auseinander. Hierfür wird jede Methode einzeln in einem Unterkapitel (6.1 – 6.3) evaluiert. Jede Evaluation beschreibt hierfür zuerst die Ausgangssituation und anschließend das Ziel der Evaluation. Darauffolgend werden der Evaluationsgegenstand, die angewendete Evaluationsmethode und das Evaluationsergebnis vorgestellt.

Die **Kapitel 7 & 8** diskutieren die Evaluationsergebnisse, fassen die Ergebnisse zusammen und schließen mit einem Ausblick ab. Hierbei wird auf potenzielle Verbesserungen für die Methode und die Funktionsweise, sowie auf verschiedene Herausforderungen in komplementären Forschungsbereichen, eingegangen.

2 Testprozesse im Supply Chain Management

In diesem Kapitel werden die Grundlagen des Supply Chain Managements, des Product Lifecycle Managements und die Grundlagen für das Testen von mechatronischen Systemen vorgestellt. In den folgenden Unterkapiteln werden die Einflüsse von der Supply Chain auf die Testprozesse identifiziert und die sich daraus resultierende notwendige Zusammenarbeit in den Testprozessen herausgearbeitet.

Im Unterkapitel 2.1 wird zunächst das Supply Chain Management vorgestellt und der Testprozess darin eingeordnet. Anschließend wird im Unterkapitel 2.2 der Testprozess vorgestellt. Dazu wird seine Aufgabe im Entwicklungsprozess mechatronischer Systeme kurz skizziert. Darauf aufbauend wird im Unterkapitel 2.3 der Einfluss der Supply Chain auf den Testprozess detailliert dargestellt. Dabei wird herausgearbeitet, dass die Testprozesse innerhalb der Supply Chain voneinander abhängig sind. Anschließend werden die Einflüsse auf die Testprozesse und die daraus resultierende Heterogenität der Testprozesse detailliert dargestellt. Im Unterkapitel 2.4 wird der Prozessablauf der betrachteten Testprozesse dargestellt und die Variantenvielfalt in der Ausgestaltung der Testprozessschritte aufgezeigt. Abschließend wird in Unterkapitel 2.5 die Problemstellung herausgearbeitet. Dazu wird die Notwendigkeit einer kollaborativen Fehlersuche als testprozessübergreifende Aktivität beschrieben, bei der die Heterogenität der Testfälle für die Zusammenarbeit der Testingenieure überwunden werden muss. Herausforderungen bei der Fehlersuche werden identifiziert und mit einem **(H)** gekennzeichnet.

2.1 Supply Chain Management

Supply Chain Management (SCM) ist das Management von Supply Chains, und ist definiert nach [Scholz-Reiter und Jakobza 1999, S. 8] durch „*Supply Chain Management, auch Lieferkettenmanagement, ist die unternehmensübergreifende Koordination der Material- und Informationsflüsse über den gesamten Wertschöpfungsprozess von der Rohstoffgewinnung über die einzelnen Veredelungsstufen bis hin zum Endkunden mit dem Ziel, den Gesamtprozess sowohl zeit- als auch kostenoptimal zu gestalten*". Hierbei gilt diese Definition nach [Busch und Dangelmaier 2002, S. 6] als kleinster gemeinsamer Nenner.

[Busch und Dangelmaier 2002, S. 4] definiert die Supply Chain als ein Netzwerk, indem mehrere Organisationen zusammenarbeiten. Das Ziel des Netzwerks ist die Erstellung eines Produktes und die anschließende Lieferung zu den Endkunden. Netzwerke können in Supply Chains verschiedene Topologien annehmen, welche die Zustände 0-NoStructure, 1-Centralized, 2-Linear, 3-Flat, 4-Hierachical, 5-Federated und 6-Starburst annehmen können [Mari et al. 2015, S. 664]. Im Rahmen dieser Arbeit wird die Topologie 4-Hierachical betrachtet, welche bei einem Hersteller

und vielen verschiedenen Lieferanten angewendet wird. Die Struktur entspricht der eines Baumes und als ein Spezialfall der einer Zulieferpyramide. Die Automobilindustrie wird explizit als Beispiel für die Anwendung der 4-Hierachical Topologie genannt [Mari et al. 2015, S. 664].

Die Berücksichtigung der Produktentwicklung und Entsorgung ist Bestandteil des SCM [Thaler 1999]. Hierdurch würde der Testprozess als Bestandteil der Produktentwicklung mit seinen Informationsflüssen im SCM betrachtet werden, was die Voraussetzung für die Entwicklung der kollaborativen Fehlersuche ist. In dieser Arbeit wird die Annahme von Thaler geteilt und sogar auf das Produktlebenszyklusmanagement (PLM) ausgeweitet. Im Rahmen der Produktentwicklung sind die in der Dissertation betrachteten Testprozesse im PLM verortet. Die betrachteten Informationsflüsse der Testprozesse sind im SCM verortet und sollen optimiert werden.

Das Ziel von SCM ist die Optimierung der Supply Chain und das hierfür geeignete SCM-Konzepte benötigt werden [Busch und Dangelmaier 2002, S. 8]. Das einzusetzende SCM-Konzept entspricht dem Konzept der kollaborativen Fehlersuche, welches im Kapitel 4 vorgestellt wird. Die Optimierung der Supply Chain kann durch Kostenvorteile, Zeitvorteile oder Qualitätsvorteile erreicht werden [Busch und Dangelmaier 2002, S. 8]. Die kollaborative Fehlersuche strebt hierbei mithilfe der Interoperabilität und automatisierten Transformationen eine Optimierung der Zeit- und Qualitätsvorteile in den Testprozessen an. Im Folgenden wird das Produktlebenszyklusmanagement und die darin verorteten Testprozesse im Detail vorgestellt.

Das Produktlebenszyklusmanagement [Kiritsis et al. 2003] definiert die Integration verschiedener Arten von Aktivitäten, aus technischer, organisatorischer und betriebswirtschaftlicher Sicht.

Die Vision von [Sudarsan et al. 2005, S. 1] ist die nahtlose Integration von Informationen in das PLM. Die von Sudarsan vertretene Vision konzentriert sich auf die Integration von Informationen in eine einheitliche Darstellung, die aus den verschiedenen Phasen des Produktlebenszyklus gespeist wird. Für die Integration der Informationen muss identifiziert werden, welche Datenquelle aus welcher Phase welche Informationen für den Informationsaustausch beisteuern kann. Im Rahmen der Arbeit sollen Informationen aus der Phase Beginning-of-Life extrahiert und reintegriert werden. Die zu berücksichtigenden Datenquellen und Informationen sind in der Regel nicht auf ein Unternehmen be-

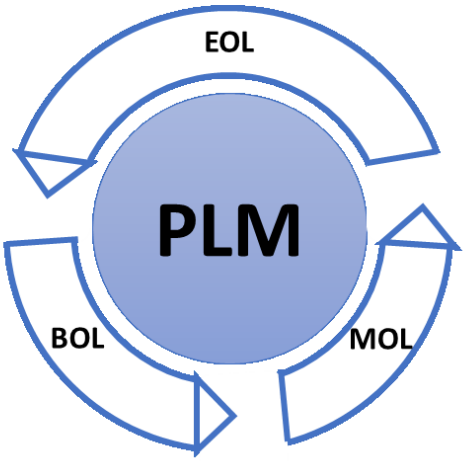

Abbildung 3: Die Phasen im Produktlebenszyklusmanagement

schränkt, sondern verteilen sich über die gesamte Zulieferpyramide, die an der Entwicklung eines Produktes beteiligt ist. Ziel des PLM ist es, die produktbezogenen Informationen über den gesamten Prozess hinweg effizient zu verwalten. Die Aktivitäten innerhalb des Produktlebenszyklusmanagements werden dabei in verschiedenen Phasen zusammengefasst. In der Abbildung 3 sind die Phasen des geschlossenen Produktlebenszyklus nach [Kiritsis et al. 2003] aufgelistet, welche im Folgenden kurz dargestellt werden.

Beginning-of-Life (BOL)

Diese Phase umfasst alle Aktivitäten, die mit der Entwicklung, der Produktion und dem Vertrieb des Produktes zusammenhängen. Dementsprechend werden in dieser Phase Dokumente erstellt, die sowohl die Anforderungen an das Produkt, seine Funktionsweise als auch die Spezifikation und den Nachweis der korrekten Funktionsweise beschreiben.

Middle-of-Life (MOL)

Diese Phase umfasst alle Aktivitäten im Zusammenhang mit der Nutzung, den begleitenden Dienstleistungen und der Wartung. Dementsprechend werden in dieser Phase Informationen aus der Art der Nutzung, den integrierten Fehler- und Diagnosesystemen und der Produktspezifikation verknüpft.

End-of-Life (EOL)

Diese Phase fokussiert auf die Widerverwendung, das Recyceln und die Entsorgung des Produktes und entspricht dem Ende der Produktlebenszeit. Der Zustand des Produktes bestimmt die Aktivitäten in dieser Phase. Hierfür können Informationen aus dem BOL und MOL benutzt werden, um eine Entscheidungshilfe für die Aktivitäten dieser Phase zu ermöglichen.

Die oben aufgeführten Phasen umfassen den geschlossenen Lebenszyklus eines Produktes und generieren unterschiedliche Dokumente, welche sich sowohl im Formalisierungsgrad als auch im Inhalt unterscheiden. Im Folgenden werden der Testprozess und insbesondere die Rolle der Testfälle als Teil des BOL im Detail betrachtet.

2.2 Grundlagen für die Durchführung von Testprozessen

Das Testen ist die primäre Technik zur Überprüfung von Hardware und Software, welche von der Industrie heutzutage verwendet wird [Broy 2005]. Die folgenden Grundlagen betrachten Funktionstests für mechatronische Systeme.

Alle Aktivitäten des Testens sind als Testprozess in die Entwicklung mechatronischer Systeme integriert und erfolgen heutzutage nach dem V-Modell, welches in

der Abbildung 4 dargestellt ist. Das V-Modell fokussiert zum einen auf die Entwicklung des Systems, welches auf der linken Seite abgebildet ist und zum anderen auf die Qualitätssicherung, welches auf der rechten Seite zu sehen ist. Die Qualitätssicherung eines mechatronischen Systems wird heutzutage mithilfe von Testprozessen garantiert, welches das System in unterschiedlichen Entwicklungsstadien testet.

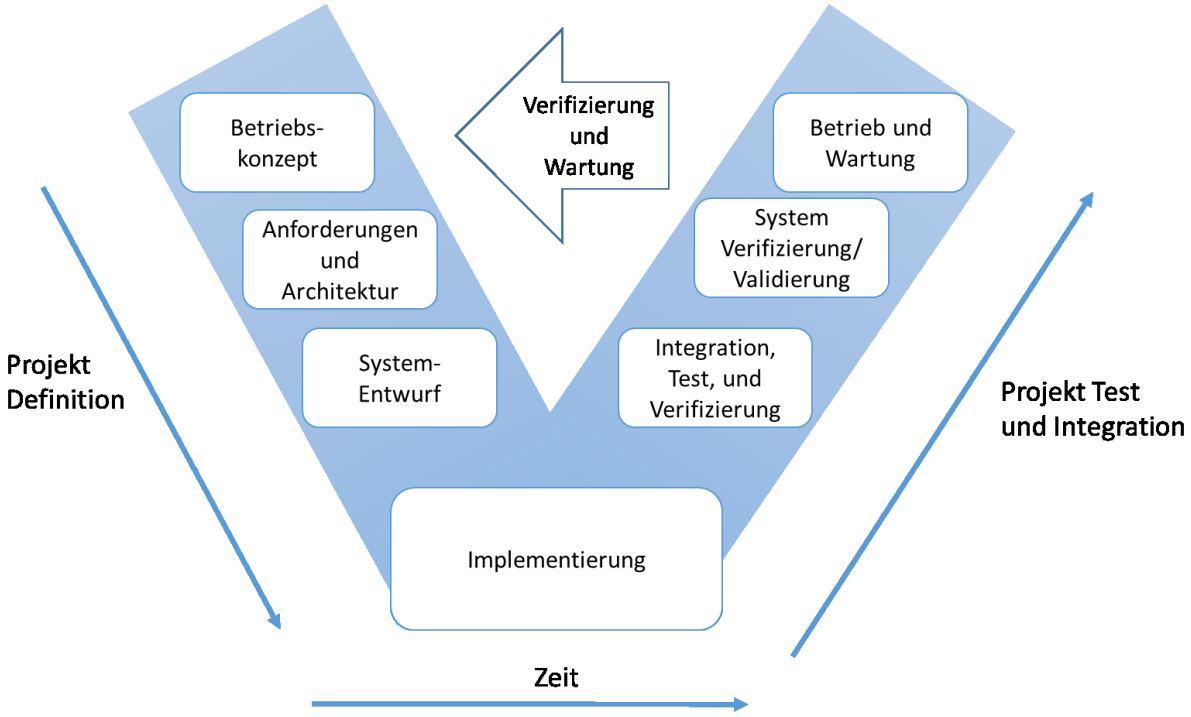

Abbildung 4: V-Modell für die Entwicklung von Systemen, in Anlehnung an [Krueger et al. 2009] und [Jastram 2016]

Die unterschiedlichen Entwicklungsstadien ergeben sich, weil komplexe mechatronische Systeme aus einem Verbund von Teilsystemen bestehen, welche von unterschiedlichen Lieferanten entwickelt und vom OEM zu einem Gesamtsystem verschmolzen werden. Sowohl bei den Lieferanten als auch beim OEM werden Testprozesse durchgeführt, um im Rahmen der Qualitätssicherung die korrekte Funktionsweise nachzuweisen.

Das Hauptziel eines jeden Testprozesses ist die Überprüfung der Funktionalität, der Zuverlässigkeit und der Betriebssicherheit des zu testenden Produktes [Franke et al. 2012]. Die vorliegende Arbeit betrachtet Funktionstests. Hierfür werden im Rahmen des Testprozesses von sicherheitskritischen mechatronischen Systemen sowohl statische Analysemethoden eingesetzt als auch eine Menge von Testfällen ausgeführt [Lübbert 2010; Thoben et al. 2011], um die korrekte Funktionsweise des Systems nachzuweisen. Jeder konkrete Testfall ist ein Testfall mit konkreten Werten [ISTQB Glossary 2021]. Er überprüft eine spezifische Funktionalität gegenüber den spezifizierten Anforderungen [Franke et al. 2019]. Der Testprozess ist abgeschlossen, sobald die korrekte Funktionsweise für alle Anforderungen gezeigt, protokolliert und

der entsprechende Bericht von der Zertifizierungsbehörde akzeptiert wurde. Anschließend kann das Produkt operativ genutzt werden.

Im Rahmen der Entwicklungsaufgaben im V-Modell erfolgt die Integration der Teile zu Komponenten, der Komponenten zu Modulen und schließlich der Module zum Endprodukt. Das Endprodukt ist z.B. das Auto oder das Flugzeug und besteht aus verschiedenen mechatronischen Systemen.
Handelt es sich bei der Entwicklungsaufgabe um eine Komponente, wird das Ergebnis von einem Modul-Lieferanten integriert. Wenn die Entwicklungsaufgabe bereits ein Modul ist, wird das Ergebnis vom OEM integriert. In beiden Fällen wird das Produkt integriert und erneut getestet. Bei diesem Integrationstest wird die Black-Box-Methode angewendet und dementsprechend werden auch die Schnittstellen getestet. Der beschriebene Prozess wiederholt sich rekursiv in der Zulieferpyramide nach oben, bis das Produkt einschließlich aller integrierten Teile, Komponenten und Module der Lieferanten vom OEM getestet wurde. Die Tests des OEM prüfen die Funktion des Produktes und testen dabei implizit alle verbauten Module und Komponenten als Black Box. Die Testprozesse sind innerhalb der Zulieferpyramide nicht unabhängig voneinander, sondern *lose gekoppelt*.

Der Begriff *lose gekoppelt* wird im Rahmen der Dissertation genutzt, um einen geringen Grad der Abhängigkeit zwischen Testprozessen in der Zulieferpyramide auszudrücken. Hierbei hat die Ausgestaltung des Testprozesses lokale Auswirkungen, wobei es externe Einflüsse gibt. Diese externen Einflüsse ergeben sich durch die Zulieferpyramide und der Testprozess kann sich denen nicht entziehen.

Im Folgenden werden die Hintergründe und Auswirkungen der losen Kopplung auf die Testprozesse innerhalb einer Zulieferpyramide detailliert dargestellt.

2.3 Testprozesse in der Zulieferpyramide

Die Testprozesse sind Bestandteil der Entwicklungsaktivitäten im PLM des OEM und sind somit der Phase BOL zuzuordnen. Die vorliegende Arbeit beschäftigt sich mit der Optimierung der Testprozesse durch eine kollaborative Fehlersuche innerhalb der Zulieferpyramide. Im Verlauf der Arbeit werden die zu lösenden Herausforderungen mit dem Buchstaben H und einem Index gekennzeichnet und im Rahmen des Konzepts wieder aufgegriffen.

Im Folgenden werden die Einflüsse der Testprozesse und die Auswirkungen auf eine kollaborative Fehlersuche dargestellt und hierüber die Herausforderung **H1** abgeleitet. Die Hierarchie der Testprozesse entspricht der Hierarchie der Lieferanten innerhalb der Zulieferpyramide. Benachbarte Testprozesse in der Zulieferpyramide sind

lose gekoppelt. Hierdurch ergeben sich Einflüsse auf den Testprozess, welche beispielsweise die spezifischen Eingaben für den Testprozess und die erwartete Ausgabe beeinflussen. Im Folgenden wird dies im Detail beschrieben.

Komplexe mechatronische Systeme vereinen in sich sowohl mechanische, elektronische als auch software-technischen Komponenten und bestehen heutzutage aus mehr als 10 000 Einzelteilen und entsprechend vielen Komponenten. Während ein durchschnittlicher PKW beispielsweise aus nur 10000 Einzelteilen [Focus Online 2014] besteht, werden für die Boeing 747-8 F bereits sechs Millionen Einzelteile benötigt. Die Entwicklung solcher komplexen mechatronischen Produkte wird heutzutage nicht mehr von einer einzelnen Firma, sondern von einem Konsortium durchgeführt.

Das Konsortium wird durch den OEM geleitet. Im Rahmen der Leitung wird auch die Art und Weise der Zusammenarbeit zwischen dem OEM und den Lieferanten definiert. In der Automobilindustrie und in der Luftfahrtindustrie wird das Konzept des Modular Sourcing für den Aufbau des Konsortiums angewendet. Hierbei beinhaltet Modular Sourcing nicht nur den Bezug der Einzelteile, sondern auch Montage- Komplettier Leistungen und Leistungen im Bereich der Qualitätssicherung [Schulte 2013]. Ein darauf ausgelegtes Konsortium ergibt die in Abbildung 5 abgebildete Zulieferpyramide.

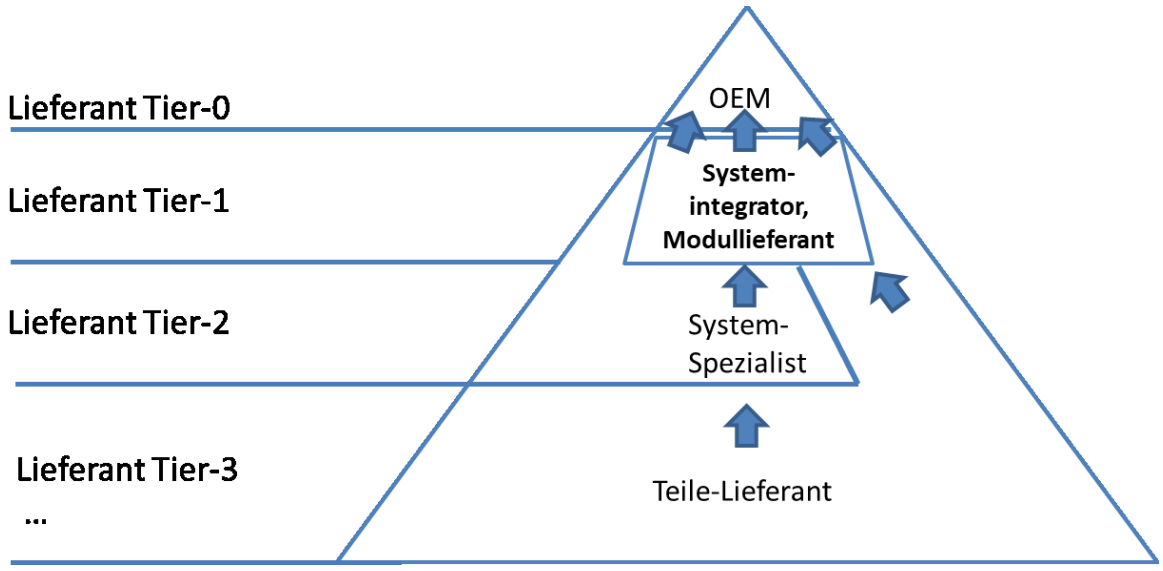

Abbildung 5: Zulieferpyramide, in Anlehnung an [Schulte 2013]

Die Spitze der Zulieferpyramide bildet der OEM, welcher seine eigenen Entwicklungsleistungen mit den entwickelten Produkten der Lieferanten zu Systemen und schlussendlich zu einem Endprodukt integriert. Hierbei greift der OEM primär auf Produkte zurück, welche eigenständige Systeme bzw. Module sind. Diese werden vom Lieferanten (Tier-1) entwickelt, welcher wiederum auf Komponenten eines

Lieferanten auf einem höheren Rang (Tier-2) zurückgreift. Die aufgezeigte Hierarchie ist nicht strikt zwischen den Stufen zu sehen, so nutzen beispielsweise Lieferanten aus unterschiedlichem Tier die gleichen Teile wie Kabel oder Schrauben, um ihre Komponenten oder Module herstellen zu können. An dieser Stelle sei erwähnt, dass Flugzeuge und Autos bereits ähnliche elektronische Systeme für die Anzeige von Instrumenten, für die Steuerung der Aktuatoren und für die Kommunikation besitzen [Munoz-Castaner et al. 2011]. Die folgenden zwei Beispiele in Abbildung 6 zeigen Ausschnitte aus der Zulieferpyramide für die Entwicklung eines Autos und eines Flugzeugs.

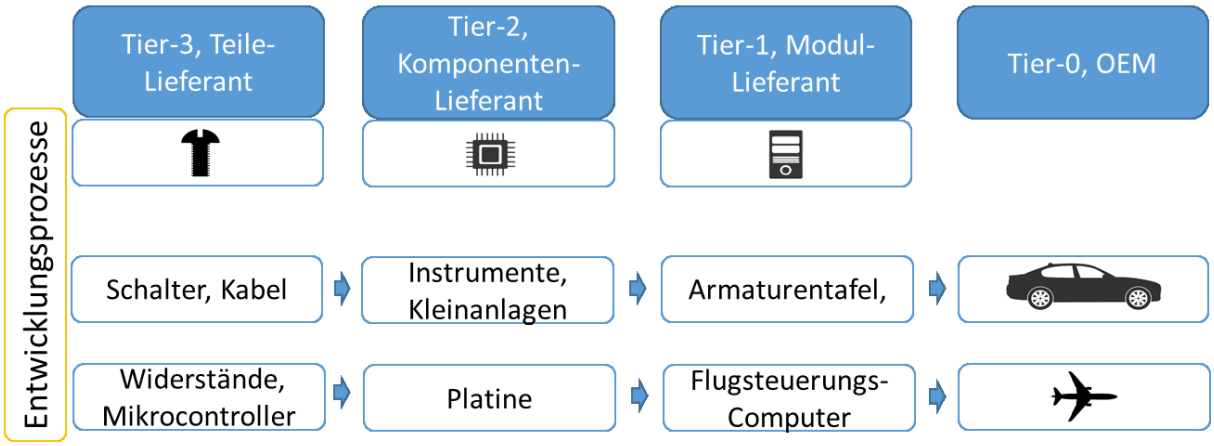

Abbildung 6: Ausschnitte aus der Zulieferpyramide für die Produkte Auto und Flugzeug

Beide Ausschnitte der Zulieferpyramiden lokalisieren die Bereitstellung von Teilen, wie Schrauben, Widerstände oder Kabel auf dem dritten Tier (Tier-3). Diese Teile werden dann in den übergeordneten bzw. niedrigeren Tier eingebaut und entsprechend ihrer Spezifikation getestet. Im Rahmen der Tests werden Belastungstests durchgeführt, welche primär die Materialeigenschaften überprüfen. Das Ziel der Belastungstests für die Teile ist bereits in den Anforderungen des Produktes von Seiten des OEM definiert und über die Hierarchie der Zulieferpyramide zu dem Teil-Lieferanten durchgereicht worden.

In den niedrigeren Stufen (Tier-2 – Tier-1) entstehen komplexere Strukturen, wie Baugruppen oder informationstechnische Systeme. Der wesentliche Unterschied zwischen einem Teil und einer Baugruppe/Komponente liegt in der Bereitstellung einer produktspezifischen Funktion, wie beispielsweise der Sensor für die Klappe eines Flugzeugs (Tier-2) oder das Steuergerät (Tier-1) für den Blinker eines Autos. Auf diesem Komplexitätslevel reichen einfache Belastungstest, wie Druck, Temperatur oder Verzug nicht mehr aus, sondern es müssen komplexe Muster aus Stimulus und Reaktion getestet werden. Hierfür werden von den Lieferanten Testfälle, basierend auf den spezifizierten Anforderungen, implementiert und für den Nachweis erfolgreich ausgeführt.

Die Position eines Lieferanten in der Zulieferpyramide hat einen direkten Einfluss auf die Entwicklung seines Produktes. Die Position beeinflusst die Anforderungsanalyse und die Zusammenarbeit mit anderen Lieferanten. Im Folgenden werden beide Einflussfaktoren im Detail vorgestellt.

2.3.1 Einfluss der Anforderungsanalyse auf den Testprozess

Einer der ersten Schritte in der Produktentwicklung ist die Anforderungsanalyse, welche auf den Normen ISO/IEC 12207:2017 [ISO/IEC/IEEE 12207:2017 2020] und ISO/IEC 15288:2008 [ISO/IEC 15288:2008 2020] basiert.

Die Anforderungsanalyse definiert die Anforderungen an das Produkt und bildet die Grundlage für den Entwicklungsprozess. Die Anforderungen werden sowohl im Spezifikations-, Entwicklungs- als auch im Testprozess genutzt und beschreiben sowohl funktionale als auch nicht-funktionale Aspekte des Produktes. Im Folgenden wird die Anforderungsanalyse aus der Sicht einer Zulieferpyramide beschrieben.

Die Anforderungsanalyse startet bei dem OEM und dort werden in einem ersten Schritt Stakeholder Anforderungen definiert. Diese werden genutzt, um Systemanforderungen und System Element Anforderungen abzuleiten. Der generische Ableitungsprozess ist in der ISO/IEC 12207:2017 beschrieben und ist in der Abbildung 7 abgebildet. Die konkreten Methoden zur Erhebung und Modellierung der Anforderungen variieren zwischen den Domänen. Eine Übersicht und Gruppierung ist gegeben in [Matyokurehwa et al. 2017].

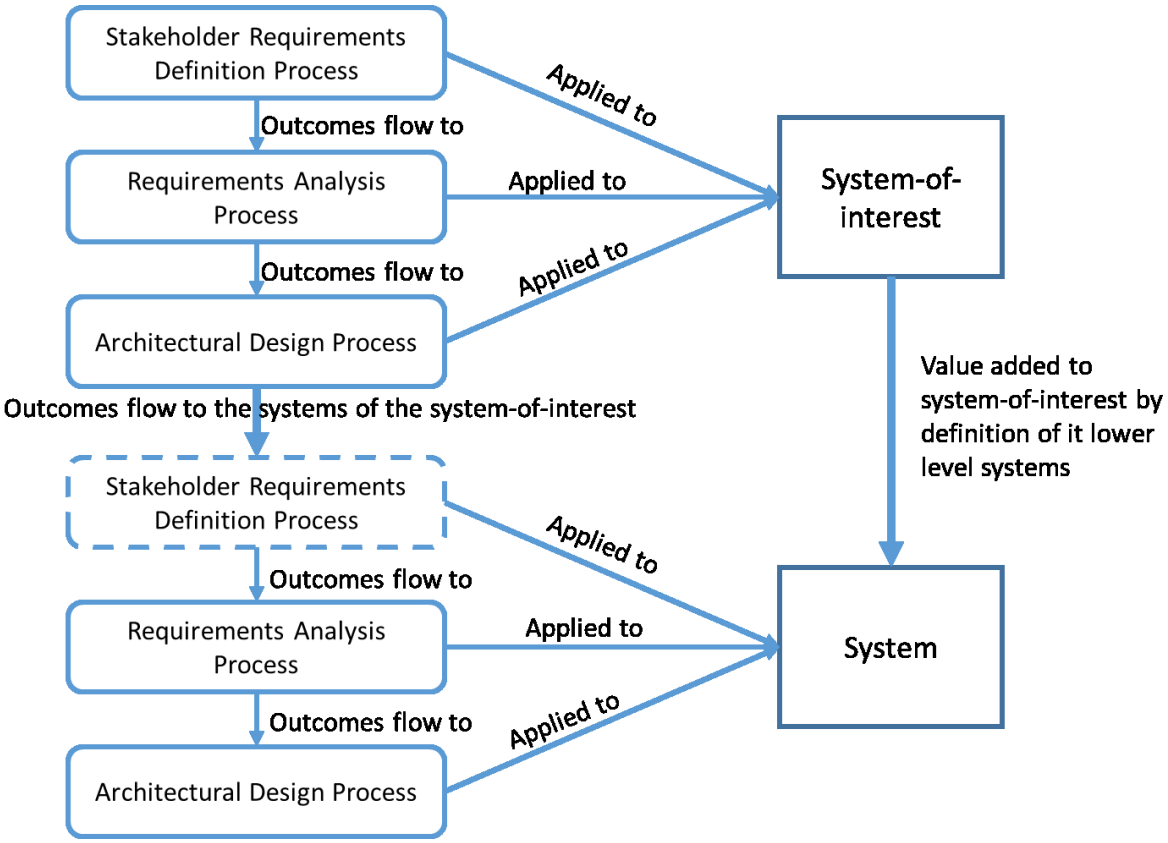

Abbildung 7: Anforderungsanalyse nach [ISO/IEC 15288:2008 2020]

Das *System-of-interest*, welches in der Abbildung 7 gezeigt ist, wird im Kontext der Arbeit als das mechatronische Produkt betrachtet, was der OEM mithilfe einer Zulieferpyramide entwickeln möchte. Die Ergebnisse der Anforderungsanalyse sind sowohl Stakeholder Anforderungen, als auch abgeleitete Systemanforderungen und System Element Anforderungen. Im Bereich der Luftfahrt werden die Kundenanforderungen mit der Bezeichnung *User Requirements* und die Top Level Anforderungen mit der Bezeichnung *Higher Level Requirements* als Stakeholder Anforderungen zusammengefasst und die System- und System Elementanforderungen zum System Requirement Document (SRD) aggregiert.

Sobald der OEM die zu entwickelnden Systeme/Module mit ihren Systemanforderungen definiert hat, können diese als Stakeholder Anforderungen innerhalb der Zulieferpyramide weitergereicht werden. Die Weiterleitung der Anforderungen ist ein rekursiver Vorgang, welcher die Anforderungsanalyse bei jedem Lieferanten beinhaltet. Dieser Vorgang ist in Abbildung 8 gezeigt.

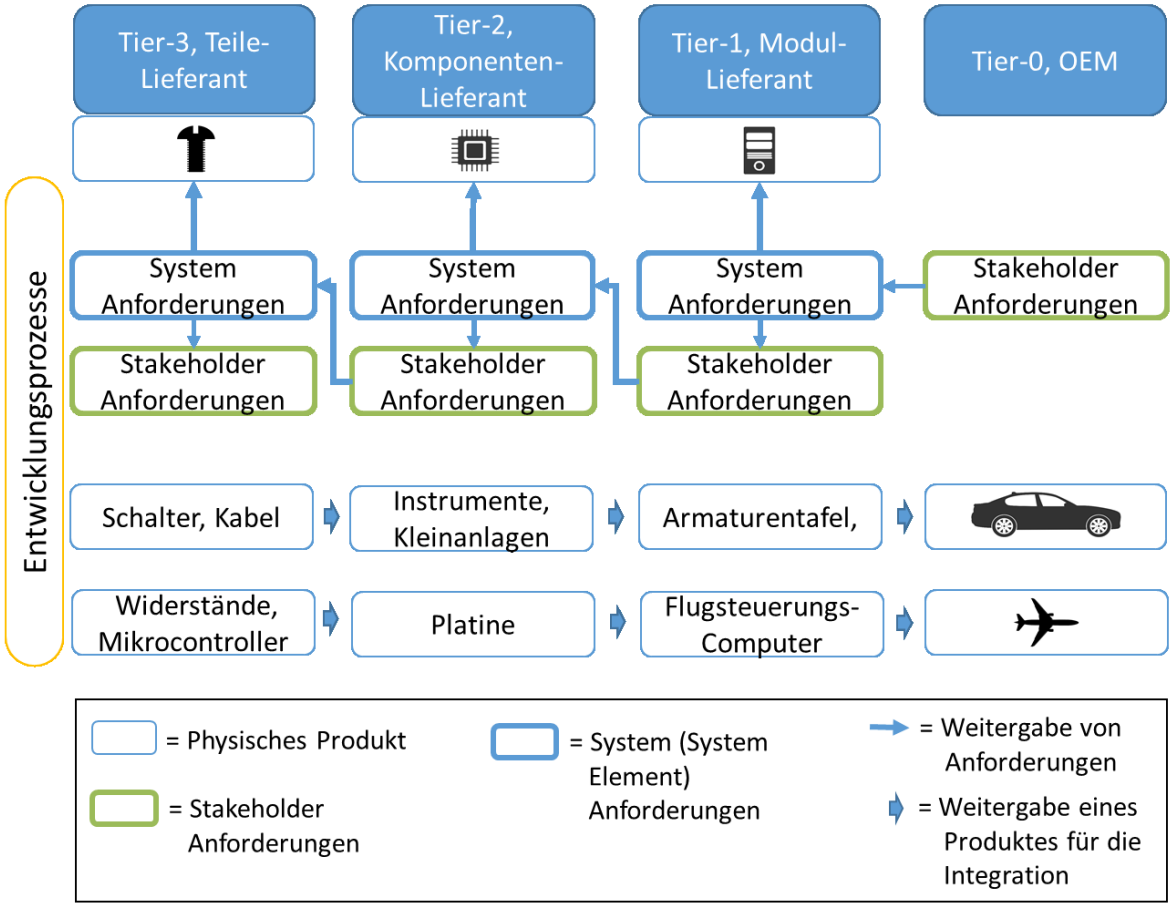

Abbildung 8: Spezialisierung der Anforderungen in der Zulieferpyramide

Bei jeder Rekursionsstufe werden die abgeleiteten System- und System Element Anforderungen an den „nächsten“ Lieferanten weitergereicht. Befindet sich der „nächste“ Lieferant auf demselben Tier werden die Systemanforderungen weitergereicht. Befindet sich der „nächste“ Lieferant auf einem höheren Tier (i+1), werden die System Element Anforderungen weitergereicht. In beiden Fällen erfolgt die Betrachtung der Anforderungen beim höheren Tier Lieferanten als Stakeholder Anforderungen, was in der Abbildung 9 gezeigt ist. Somit erfolgt eine rekursive Transformation der Anforderungen des OEM über die gesamte Zulieferpyramide, was in Abbildung 9 zusammengefasst ist.

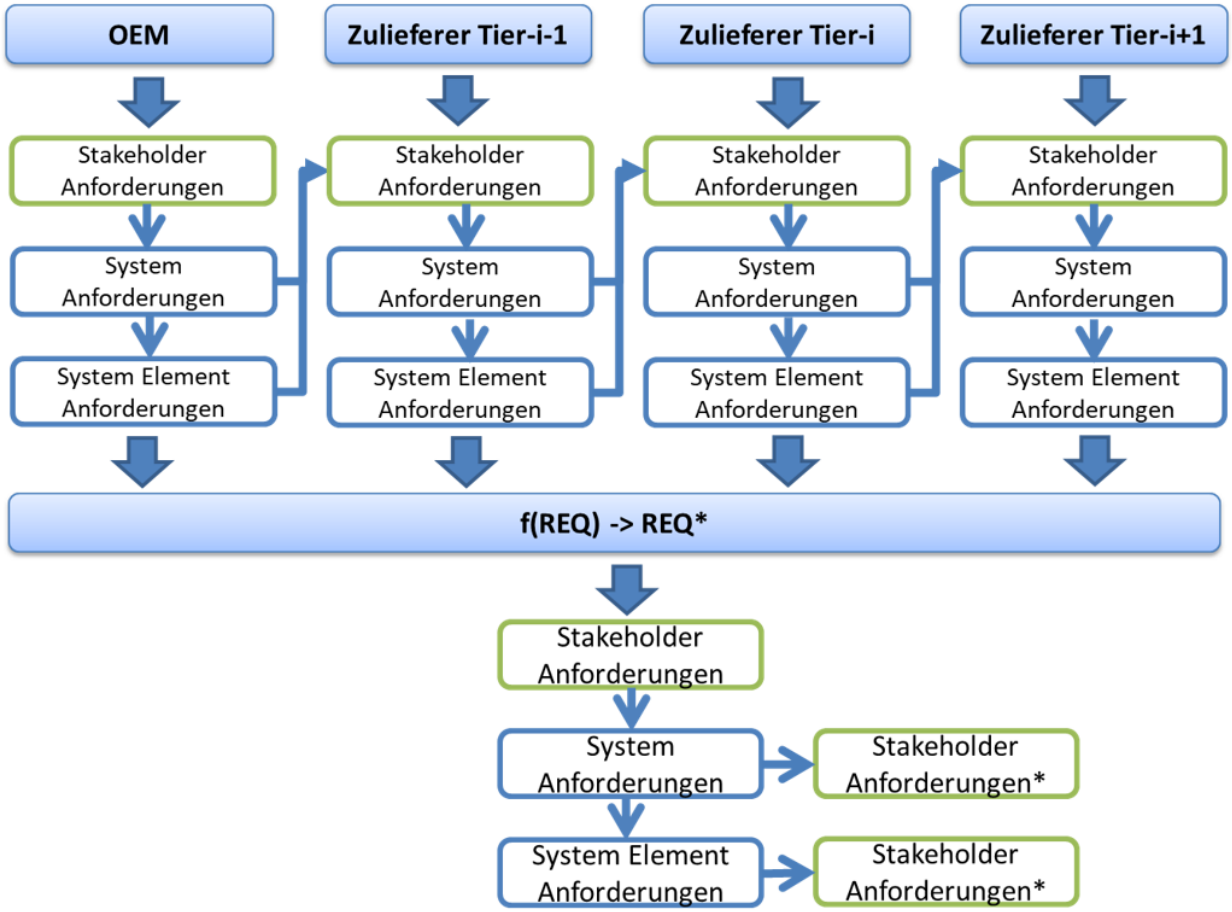

Abbildung 9: Transformation der Anforderungen in der Zulieferpyramide nach [Franke und Thoben 2022]

Die rekursive Transformation von Stakeholder Anforderungen zu System Anforderungen und wieder zu Stakeholder Anforderungen führt zu einer Baumstruktur der Anforderungen in der Gesamtheit aller durchgeführten Anforderungsanalysen innerhalb der Zulieferpyramide. Dabei nimmt der Elternknoten immer die Rolle des Hyperonyms und der Kindsknoten die Rolle des Hyponyms eins, worüber sich eine semantische Ähnlichkeit definieren lässt. Die Baumstruktur spiegelt hierbei auch die lose Kopplung der Testprozesse untereinander wider. Im Folgenden wird die Berücksichtigung der verschiedenen Anforderungen in den Testprozessen beschrieben.

Das Ziel jeder Anforderung, unabhängig von ihrem Typ und ihrer Position in der Zulieferpyramide, ist eine Definition, die gemäß ISO/IEC 15288:2017 überprüfbar ist. Überprüfbar bedeutet, dass die Anforderung messbare Bedingungen und Einschränkungen definiert. Das bedeutet, dass sowohl die Stakeholder Anforderung mit den Mitteln des Urhebers (OEM oder Lieferant (Tier-i)) als auch die abgeleiteten Systemanforderungen mit den Mitteln des Lieferanten (Tier-i+1) überprüfbar sein müssen. Das Mittel zur Überprüfung der Anforderungen ist jeweils der Testprozess, welcher auf Testfällen basiert.

Zusammenfassend lässt sich sagen, dass die Testprozesse die korrekte Funktion des Produktes anhand der Anforderungen überprüfen, die durch Testfälle verifiziert werden. Die zu prüfenden Anforderungen ergeben sich aus der Zulieferpyramide und begründen die lose Kopplung (*lose gekoppelt*) der Testprozesse.

2.3.2 Einfluss der Eigenschaften eines mechatronischen Systems auf den Testprozess

Ein komplexes mechatronisches Produkt ergibt sich aus der Integration von Elektronik, Regelungstechnik und Mechanik. [Bolton 1997]. Das Ergebnis ist ein Verbund aus verteilten, ereignisdiskreten Systemen [Campetelli und Broy 2018]. Ein Beispiel ist das Hochauftriebssystem eines Flugzeugs und ist in der Abbildung 1 zu sehen. Hierbei sind der SFCC und die WTB ereignisdiskrete Systeme, welche über den Bus ARINC 429 miteinander kommunizieren.

Die Kommunikation zwischen den ereignisdiskreten Systemen im mechatronischen Produkt ist notwendig, um die Produktfunktionen zu realisieren. Die technologische Umsetzung der Kommunikation wird im Entwicklungsprozess durch die angrenzenden Lieferanten in der Zulieferpyramide vorgegeben. Dabei definiert der OEM über Anforderungen, wie die Module und Komponenten untereinander kommunizieren sollen. Beispielsweise kommunizieren der SFCC (Modul) und das WTB (Modul) des Hochauftriebssystems bereits über Protokolle. Jedes Modul besteht wiederum aus Komponenten und der Modul-Lieferant bestimmt intern die Art der Kommunikation.

Die Kommunikation zwischen Teilen, Komponenten und Modulen erfolgt über verschiedene Arten von Hardware-Schnittstellen. Dabei können die Hardware-Schnittstellen nach der Verarbeitung von Eingangssignalen zu Ausgangssignalen klassifiziert werden. Eine entsprechende Einteilung erfolgt nach [Sima und Zapciu 2022]:

- Das Signal ändert die physikalischen Eigenschaften (Zum Beispiel: mechanisches Signal -> elektrisch)
- Das Signal ändert seine Signalcodierung (Zum Beispiel: analog -> digital)
- Das Signal ändert den Signalübertragungsmodus (Zum Beispiel parallele Übertragung -> serielle Übertragung)

Generell können die Hardware-Schnittstellen auch über die Art der Signalweitergabe klassifiziert werden. Eine entsprechende Einteilung erfolgt nach [Sima und Zapciu 2022]:

- Keine Schnittstelle
- Eine passive Schnittstelle, welche keine Stromversorgung benötigt

- Eine aktive Schnittstelle, welche eine Stromversorgung für die Konvertierung des Signals benötigt
- Eine intelligente Schnittstelle, welche eine programmierbare Signalverarbeitung beinhaltet

Die für diese Arbeit relevanten Hardware-Schnittstellen sind Schnittstellen, welche über Testfälle getestet werden können. Dementsprechend werden Hardware-Schnittstellen betrachtet, welche die Übertragung von Signalen mithilfe einer Signalcodierung und einem Signalübertragungsmodus ermöglichen. Zusätzlich verfügen sie über aktive und intelligente Schnittstellen. Hierbei erfolgt das Testen der Schnittstellen redundant in den *lose gekoppelten* Testprozessen sowohl zwischen dem Komponenten-Lieferanten und dem Modul-Lieferanten als auch zwischen dem Modul-Lieferanten und dem OEM.

Die technische Umsetzung der für die Kommunikation relevanten Hardware-Schnittstellen erfolgt hard- und softwareseitig durch Protokolle/APIs. Damit können die Schnittstellen und die darüber laufende Kommunikation durch HIL-Testverfahren verifiziert werden. Die hierfür notwendigen Anforderungen werden vom OEM an seine Modul-Lieferanten weitergegeben. Der Modul-Lieferant wiederum gibt seine Anforderungen an den Komponenten-Lieferanten weiter. Daraus ergeben sich für die *lose gekoppelten* Testprozesse ähnliche Anforderungen und Testbedingungen, die für eine kollaborative Fehlersuche zu berücksichtigen sind. Im Folgenden wird die Wechselwirkung zwischen Anforderungen und Testbedingungen verallgemeinert.

Das Produkt des Lieferanten (Tier-i) wird basierend auf Stakeholder Anforderungen entwickelt. Hierbei sind die Stakeholder Anforderungen die System Anforderungen von dem niedrigeren Lieferanten (Tier-i-1). Diese Wechselwirkung ist für das Testen von Schnittstellen in Abbildung 10 zusammengefasst.

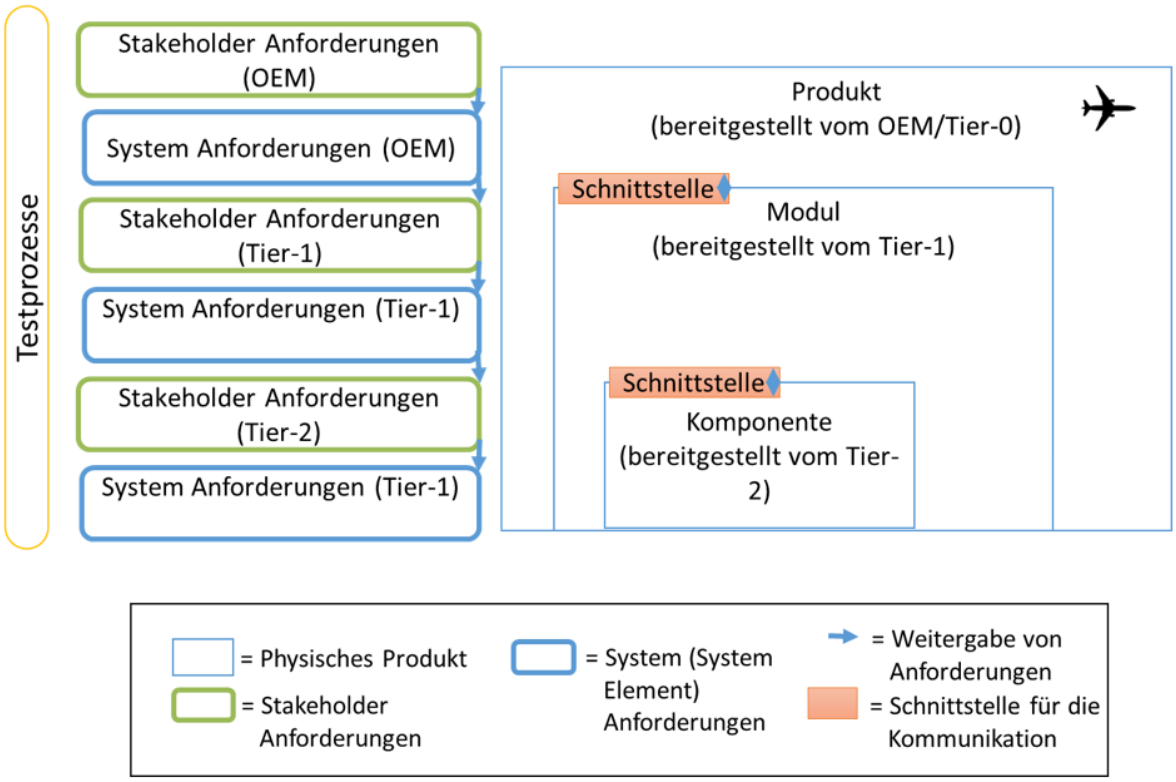

Abbildung 10: Wechselwirkung zwischen Anforderungen und Hardware-Schnittstellen aus der Perspektive des Testens

Die Stakeholder Anforderungen werden von dem Lieferanten (Tier-i) zu System Anforderungen spezialisiert. Diese System Anforderungen definieren, wie die festgelegten Schnittstellen für die Kommunikation und die erwartete Funktionalität aus den Stakeholder Anforderungen von dem Lieferanten (Tier-i-1) technisch im Rahmen der Entwicklung umgesetzt werden sollen. Anschließend dienen die System Anforderungen vom Lieferanten (Tier-i) und die Stakeholder Anforderungen vom Lieferanten (Tier-i-1) als Eingabe für den Testprozess des Lieferanten (Tier-i). Es ergeben sich dementsprechend bei der Gestaltung des Testprozesses des Lieferanten (Tier-i) drei Abhängigkeiten:

- Informationstechnische Schnittstellen vom Produkt des Lieferanten (Tier-i-1)
- Informationstechnische Schnittstellen vom Produkt des Lieferanten (Tier-i+1)
- Technische Realisierung vom Produkt des Lieferanten (Tier-i)

Die Abhängigkeiten wirken sich auf die Spezifikation und Implementierung der Testfälle für den Testprozess des Lieferanten (Tier-i) aus. Um die Abhängigkeiten zu berücksichtigen, werden ähnliche Testfälle für die Überprüfung der Schnittstellen in den *lose gekoppelten* Testprozessen implementiert.

Das Ergebnis ist eine Untermenge an Testfällen von unterschiedlichen Lieferanten, welche semantisch ähnlich sind. Semantisch ähnlich bedeutet hierbei, dass die Stimulierung des Prüflings, die Stimulierung des Zielzustands oder die Überwachung der Reaktion des Prüflings sich ähnlich für das Testen der Kommunikation verhalten, indem beispielsweise die gleichen Signale oder Bedingungen verwendet wurden. Dieser Zusammenhang wird in der Abbildung 11 verdeutlicht.

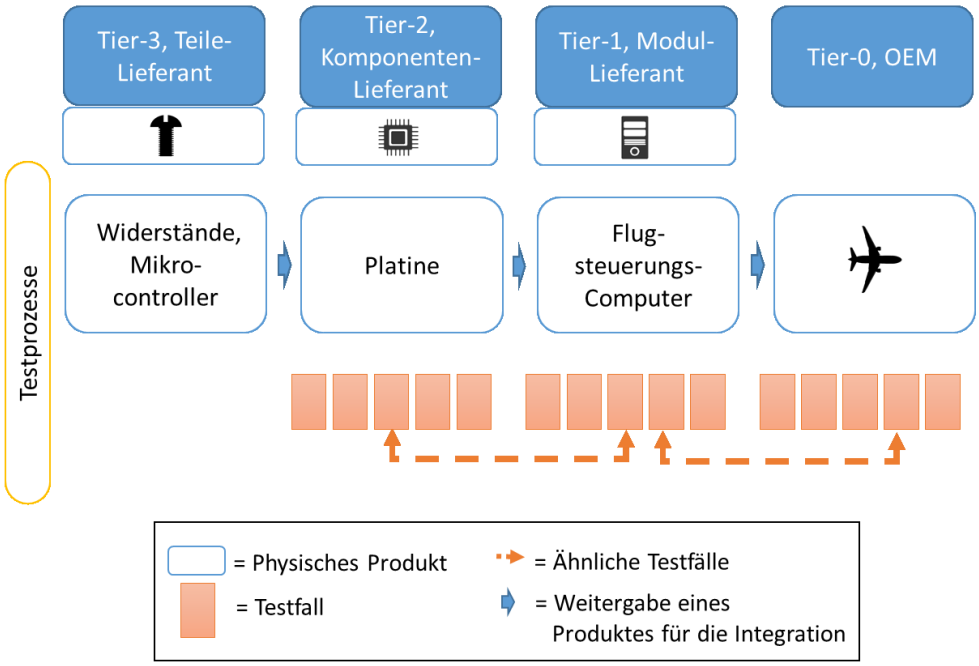

Abbildung 11: Ähnlichkeit von Testfällen in den Testprozessen der Zulieferpyramide

Abgesehen von der Realisierung der Kommunikation innerhalb des mechatronischen Produktes bestimmt die Art des Produktes, nämlich ob es ein Teil, eine Komponente oder ein Modul ist, wie die Testfälle aufgebaut und welche Funktionen getestet werden müssen. Dieser Zusammenhang wird hierauf kurz angerissen, weil dies einen Einfluss auf die *Testskriptsprachen* und Testsysteme im Testprozess besitzt. Zu diesem Zweck wird der Einfluss exemplarisch durch die Produkteigenschaften beschrieben.

Die Teile, Komponenten und Module eines mechatronischen Systems besitzen einen unterschiedlichen Aufbau und implementieren über ihren Aufbau eine spezifische Funktion. Die korrekte Funktionsweise wird mithilfe von Testprozessen überprüft, welche die spezifischen Funktionen mithilfe von Testfällen überprüfen können. Die Einordnung dieser Testprozesse in die rechte Seite des V-Models ist in Abbildung 12 gezeigt. Im Folgenden wird die Eignung der *Testskriptsprachen* für das Testen der spezifischen Funktionen bezogen auf Testaufgabe, welche im V-Modell abgebildet ist, und verantwortlichen Lieferanten skizziert.

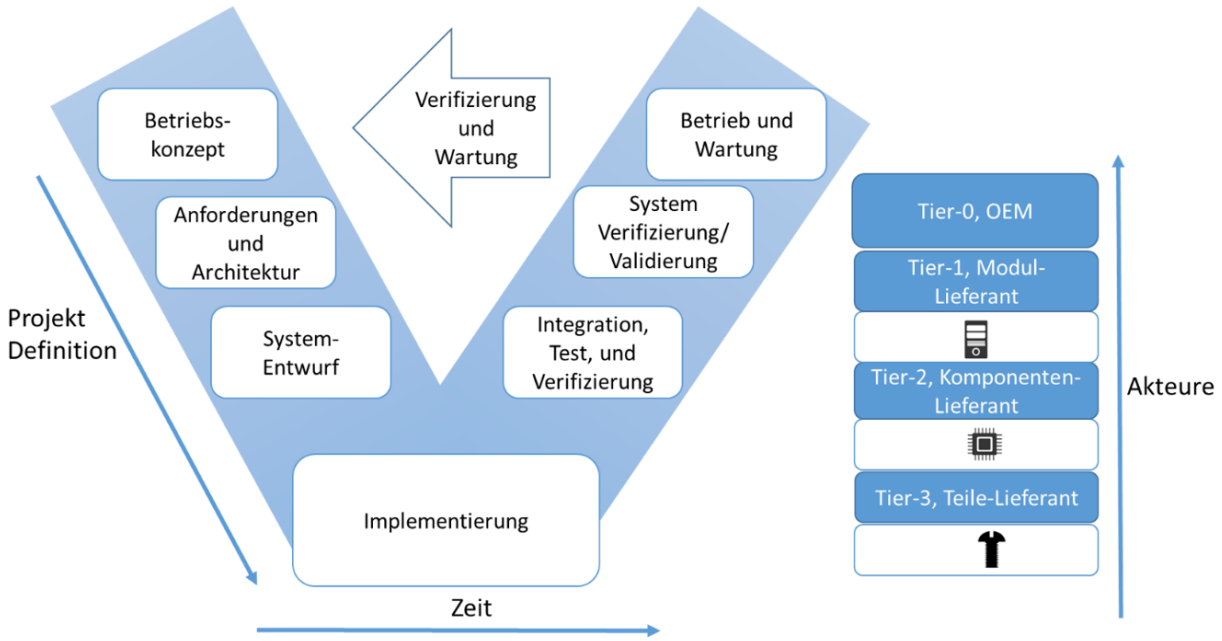

Abbildung 12: Position der Lieferanten in dem V-Modell, angelehnt an [Krueger et al. 2009] und [Jastram 2016]

Die Eignung von *Testskriptsprachen* wird exemplarisch für Teile, Komponenten und Module angerissen. Hierbei ist eine *Testskriptsprache* im Kontext der Arbeit eine domänenspezifische Sprache (DSL), mit dessen Hilfe Testfälle als Testskripte ausgeführt werden können. Eine DSL ist hierbei eine Programmiersprache oder eine ausführbare Spezifikationssprache, die auf eine Domäne zugeschnitten ist [van Deursen et al. 2000]. Ein Testskript ist hierbei eine Abfolge von Anweisungen, die ausgeführt werden können [ISTQB Glossary 2021].

Ein Lieferant auf Tier-3 entwickelt Teile, welche überwiegend mechanische Bestandteile beinhalten und keine informationstechnischen Elemente. Dementsprechend kann kein HIL-Testprozess angewendet werden und wird nicht näher betrachtet. Die darauf aufbauenden Komponenten und Module integrieren zunehmend Elektronik und Software. Dieser Trend wird sich durch die digitale Revolution in Autos [Bourns 2023] und Flugzeugen [Fette und Herrmann 2017] verstetigen. Der Fokus der betrachteten Testprozesse fokussiert auf Komponenten und Module, welche elektronische oder softwaregestützte Funktionen bereitstellen. Diese Funktionen können mithilfe von Testfällen in einer HIL-Umgebung getestet werden.

Die testrelevanten Produkteigenschaften der Komponenten und Module werden durch die System- und Systemelement-Anforderungen spezifiziert. Sie beeinflussen das Design des mechatronischen Systems und in einem nachgelagerten Schritt auch die benötigte Funktionalität der *Testskriptsprachen*. Hierbei führen unterschiedliche funktionale Anforderungen und daraus abgeleitete Hardwarearchitekturen des me-

chatronischen Systems auch zu unterschiedlichen Funktionen einer *Testskriptsprache*, um die Grundfunktionen für die Stimulierung des Prüflings, die Überwachung des Prüflings und die Auswertung der Reaktion des Prüflings zu ermöglichen. Das folgende Beispiel soll die Wechselwirkung der gewählten Hardwarearchitektur für Komponenten und Module zu den *Testskriptsprachen* verdeutlichen:

Es gibt smarte Sensoren als Komponenten, welche ihre Messwerte beispielsweise über Serial Peripheral Interface (SPI) oder auch über Wireless Local Area Network (WLAN) übermitteln können. Die Testfälle wären für beide Sensoren aus funktionaler Sicht ähnlich. Erst die Ausführung der Testfälle müsste zwingend die Hardware-Schnittstellen der Sensoren berücksichtigen. Für das Auslesen der Werte über SPI und das Auslesen der Werte über WLAN werden unterschiedliche Protokolle benötigt. Somit kann eine *Testskriptsprache*, welche ausschließlich SPI unterstützt, keine smarten Sensoren über WLAN testen. Dementsprechend beeinflussen die Anforderungen und die daraus abgeleiteten Produkteigenschaften die Wahl der *Testskriptsprache*.

Abgesehen von den Eigenschaften der *Testskriptsprache*, beeinflussen die nicht funktionalen Anforderungen an die Komponenten/Module auch die Architektur des Testsystems. Die folgenden Beispiele sollen diese Wechselwirkung verdeutlichen:

- Der Test, ob Statuslampen in der Kabine eines Flugzeugs leuchten, kann automatisch oder manuell durch einen Testingenieur erfolgen. Beide Optionen sind möglich, weil das Registrieren der Statusänderung einer Statuslampe in der Kabine mehrere Sekunden dauern darf. Ein geeignetes Testsystem muss entsprechend in der Lage sein, die Statuslampe über die IO anzubinden und den Status regelmäßig zu erfassen.
- Der Test ob ein Wellenbruch bevorsteht, muss innerhalb von wenigen Millisekunden garantiert vom Testsystem erkannt werden. Hierfür ist eine manuelle Interaktion mit dem Testingenieur nicht mehr möglich. Dementsprechend muss das Testsystem beim Testen der Welle technisch in die Lage versetzt werden, in harter Echtzeit und im Millisekunden Bereich reagieren zu können. Hierfür werden andere Architekturen für Testsysteme benötigt, als wenn eine manuelle Interaktion mit dem Testingenieur vorgesehen ist.

In beiden beschriebenen Szenarien, welche aus unterschiedlichen Systemen eines Flugzeugs stammen, bestimmen die nicht funktionalen Anforderungen maßgeblich die Funktionen der *Testskriptsprache* und die Architektur des Testsystems. Dementsprechend werden unterschiedliche Testskriptsprachen eingesetzt. In den Unterkapitel 2.3 wurden die Einflüsse der Testprozesse herausgearbeitet und ihre Auswirkungen auf die Testskriptsprachen angedeutet. Im Folgenden wird der Ablauf eines Testprozesses und die Berücksichtigung der externen Einflüsse dargestellt.

2.4 Gestaltung und mögliche Varianten eines Testprozesses

Ein Testprozess für die Softwareentwicklung besteht nach [Witte 2020, 60-67] aus fünf Schritten, welche in Abbildung 13 dargestellt sind. Hierbei ist der gezeigte Prozess nicht auf das Testen von Software beschränkt, sondern kann auch eingeschränkt auf das Testen von mechatronischen Systemen für die Test Level Integration und System angewendet werden, welche von den Komponenten-Lieferanten, Modul-Lieferanten und dem OEM benötigt werden. Der Grund dafür ist, dass mechatronische Systeme Informationssysteme sind und sie im Verbund getestet werden. Dementsprechend verfügen sie über APIs und ihre Kommunikation ist über Protokolle realisiert. Zusätzlich erfolgt sowohl die Steuerung der Systeme durch implementierten Quellcode [Sima und Zapciu 2022] als auch die interne Funktionsweise wird durch Software realisiert [Sima 2022].

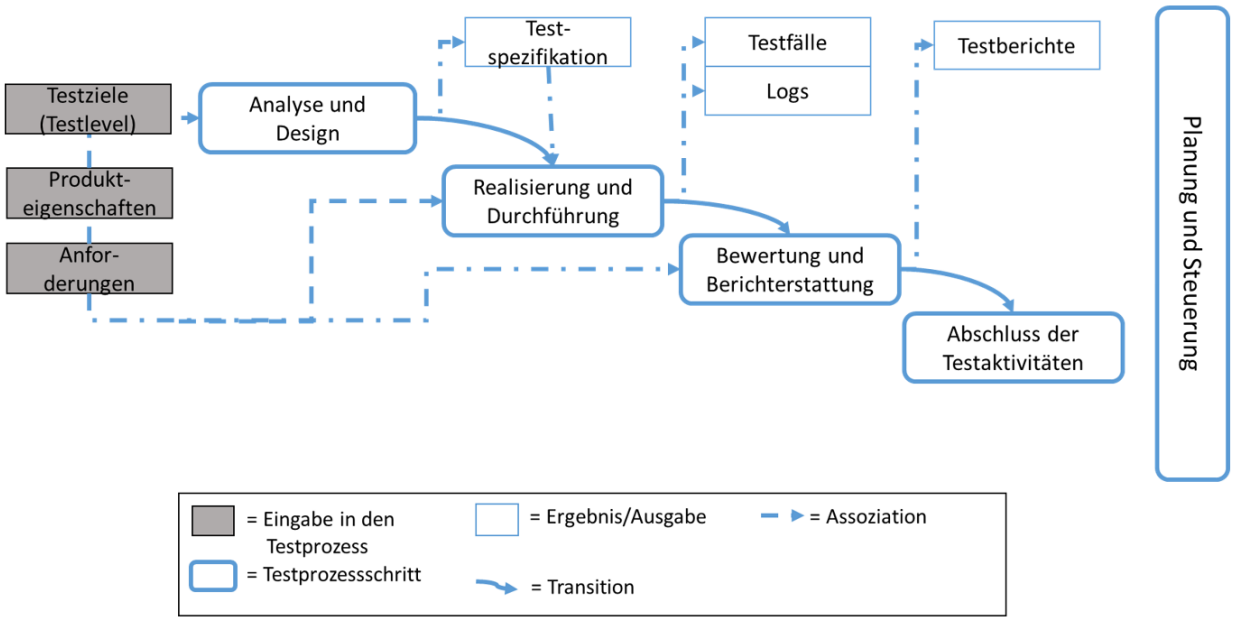

Abbildung 13: Gleicher Ablauf der Testprozesses in der Zulieferpyramide

Die fünf Schritte eines Testprozesses sind *Planung und Steuerung*, *Analyse und Design*, *Realisierung und Durchführung*, *Bewertung und Berichtserstattung* und *Abschluss der Testaktivitäten*. Der erste Schritt *Planung und Steuerung* läuft parallel zu den übrigen vier Schritten und plant die Testressourcen und den Zeitplan. Die inhaltliche Durchführung des Testprozesses wird durch die vier übrigen Schritte sequenziell abgearbeitet, wobei es zu Überlappungen und Rückkopplungen kommen kann.

Im zweiten Schritt *Analyse und Design* wird analysiert, ob das System überhaupt testbar ist. Zusätzlich wird analysiert, ob die Testziele und die Testabdeckung pas-

send zu dem geplanten Testumfang und bestehenden Ressourcen sind. Hierbei beschreiben die Ressourcen sowohl die zur Verfügung stehenden Testingenieure als auch die benötigte Testinfrastruktur. Die Definition der Testvoraussetzungen, Testdaten, Konzeption der Testautomatisierung und der Testdefinitionen stehen im Fokus des Schrittes. Hierfür werden als Eingaben, die Testziele, die Testabdeckung, die Produkteigenschaften und die Anforderungen benötigt. Diese Eingaben sind, wie im Kapitel 3.1.2 detailliert beschrieben, spezifisch für jeden Testprozess in der Zulieferpyramide. Das Ergebnis ist die Testspezifikation, welche maßgeblich die Spezifikation und Implementierung der Testfälle vorgibt.

Die Anforderungen beeinflussen hierbei nicht nur den Schritt Analyse und Design, sondern beeinflussen auch die Schritte *Realisierung und Durchführung* und *Bewertung und Berichtserstattung.* Der Grund hierfür liegt in dem Ziel, dass Testfälle das korrekte Verhalten des Prüflings basierend auf den Anforderungen definieren und entsprechend überprüfen. Die Herausforderung ist, dass sich die Anforderungen durch die Anforderungsanalysen der Zulieferpyramide wandeln (siehe 2.3.2).

Somit wird ein Testprozess innerhalb der Zulieferpyramide in seiner Ausgestaltung durch die Zulieferpyramide beeinflusst. Das Ergebnis der Beeinflussung der Testprozesse durch die Zulieferpyramide ist, dass alle Testprozesse den gleichen Prozess durchlaufen, aber bereits im ersten Schritt *Design und Analyse* spezifische Eingaben erhalten und sich entsprechend spezialisieren. Die Spezialisierung führt beispielsweise dazu, dass unterschiedliche Methoden für die Definition des Testraums oder unterschiedliche Testmethoden eingesetzt werden können. Die Auswahlmöglichkeiten für die Spezifizierung und Ausführung der Testfälle sind in Abbildung 14, welche auf [Lübbert 2010; ISO/IEC/IEEE 29119-1:2013 2013; Jorgensen 2018; Franke et al. 2020a] basieren, abgebildet.

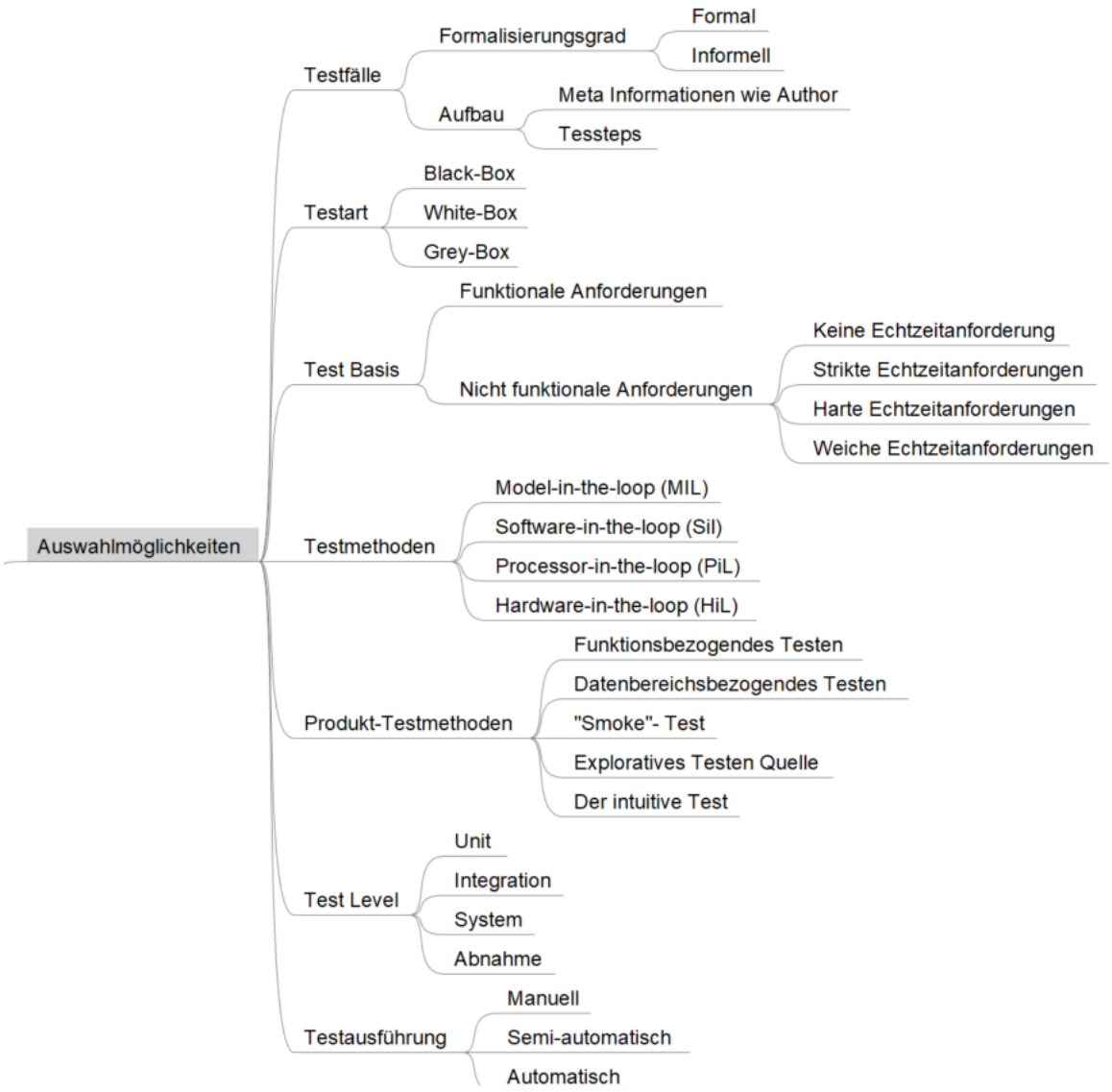

Abbildung 14: Variantenvielfalt in der Ausgestaltung der Testprozessschritte

Jede Kombination der Blätter aus Abbildung 14 führt zu einem einzigarten Testprozess, welcher spezifische Anforderungen sowohl an die Testumgebung als auch an die *Testskriptsprachen* stellt. Hierdurch wird die Heterogenität zwischen den Testprozessen kreiert. Aufgrund dieser Heterogenität kann nicht jeder Testprozess mit derselben *Testskriptsprache* implementiert werden. Das Ergebnis ist die Etablierung verschiedener *Testskriptsprachen* in den Testprozessen, welche in 3.2 beschrieben sind.

2.5 Notwendigkeit einer kollaborativen Fehlersuche

Die Testprozesse für mechatronische Systeme folgen in einer Zulieferpyramide auf jeder Stufe den gleichen Prozessschritten, wobei die Inhalte der Testschritte hinsichtlich der verwendeten Methoden, *Testskriptsprachen* etc. heterogen sind. Diese

Heterogenität ist für die Testaktivitäten innerhalb der Zulieferpyramide irrelevant, solange die Testprozesse sequenziell erfolgreich durchgeführt werden. Erfolgreich heißt hierbei, dass die Testprozesse mit den Testfällen gezeigt haben, dass ihr Produkt entsprechend den Anforderungen funktioniert. Somit sind testprozessübergreifende Aktivitäten zwischen benachbarten Testprozessen nicht notwendig. Diese Aussage trifft so lange zu, bis Integrationstests bei einem Lieferanten fehlschlagen. **In einer solchen Situation muss die Ursache ergründet werden und die Expertise der Testingenieure eines Testprozesses reicht nicht mehr aus. Hierfür müssen die vor- und nachgelagerten Lieferanten eine testprozessübergreifende Aktivität starten.** Das Ziel im Kontext der Arbeit ist die Identifizierung der Fehlerursache in den Testprozessen mithilfe einer Fehlersuche. Hierbei wird primär versucht, Fehler innerhalb der Testprozesse und nicht Fehler im Produkt zu finden. Zum Beispiel könnte der Fall auftreten, dass die Testprozesse für das Testen der gleichen Anforderung den Prüfling unterschiedlich stimuliert haben und zu unterschiedlichen Ergebnissen gelangt sind.

Die Eingabe der betrachteten Fehlersuche sollte der fehlgeschlagene Testfall sein, welcher mindestens ein fehlgeschlagenes Verdikt enthält. Startpunkte ohne konkrete Indizien für eine Fehlersuche wären generisch, beziehen sich auf die Konfigurationen und stehen nicht im Fokus der Arbeit. Die Heterogenität der Inhalte in den Testschritten *Realisierung und Durchführung* und *Bewertung und Berichtserstattung* führt zwischen den benachbarten Testprozessen dazu, dass der fehlgeschlagene Testfall inklusive seines Testberichtes nicht unmittelbar von den Testingenieuren in den benachbarten Testprozessen uneingeschränkt nutzbar ist. Gründe hierfür sind u.a. die Verwendung von unterschiedlichen *Testskriptsprachen* und Testsystemen.

In einem solchen Szenario wird im Rahmen der Fehlersuche beispielsweise der fehlgeschlagene Testfall vom Modul-Lieferanten und der bestandene Testfall vom Komponenten-Lieferanten, welcher den ähnlichen Sachverhalt beschreibt, verglichen. Hierbei wird es sich in der Regel um Testfälle handeln, welche auch die Schnittstellen mittesten. Der Hintergrund ist, dass Schnittstellen immer bei beiden Kommunikationspartnern (Sender, Empfänger) im Kontext von Komponenten- und Integrationstests getestet werden müssen.

Die Identifizierung des Testfalls, welcher ähnlich zu dem fehlgeschlagenen Testfall des Modul-Lieferanten ist, erfolgt aus einer Menge von Testfällen vom Komponenten-Lieferanten. Die Identifizierung kann über die getesteten Anforderungen der Testfälle erfolgen. Hierfür kann der Testfall gesucht werden, welche die abgedeckten System Anforderungen vom fehlgeschlagenen Testfall überprüft. Dieser Zusammenhang ist in Abbildung 15 skizziert. Der gefundene Testfall (Mögliche Suchergebnisse sind auch Regionen innerhalb eines Testfalls oder mehrere Testfälle) und der fehlgeschlagene Testfall sind semantisch ähnlich und ermöglichen eine Fehlersuche. Hierbei ist die Voraussetzung, dass es sich um funktionale Tests handelt und

das die getestete Systemfunktion über eine API zugänglich ist, um eine Integration durch ein Produkt des niedrigeren Lieferanten zu ermöglichen.

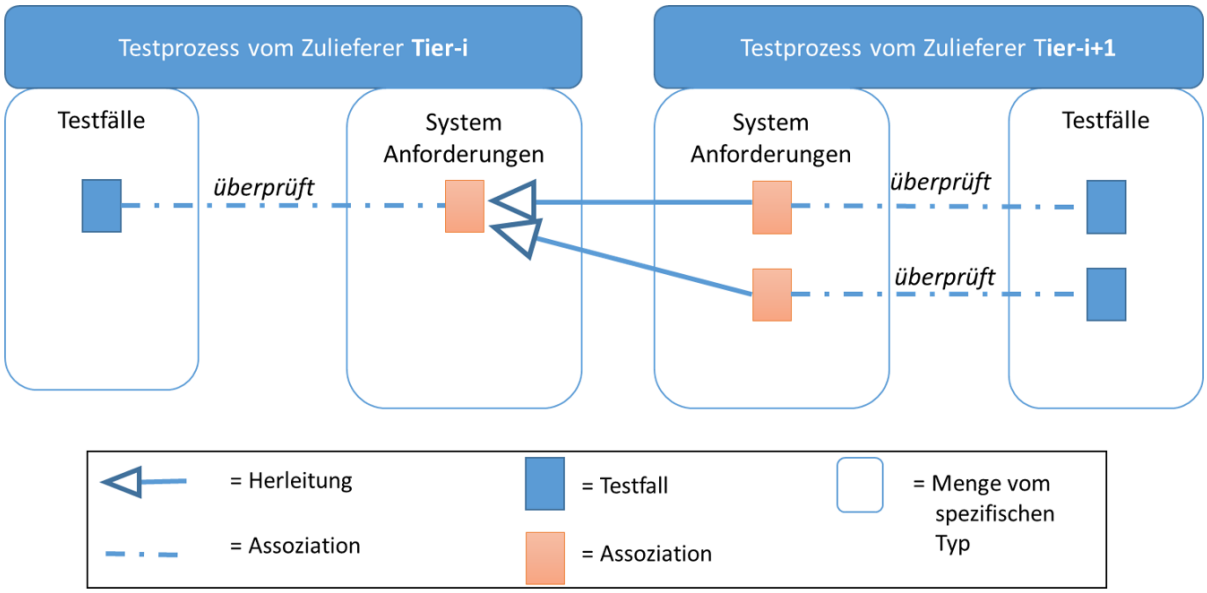

Abbildung 15: Suche nach ähnlichen Testfällen in *lose gekoppelten* Testprozessen

Ein übliches Szenario für die Fehlersuche beschreibt die Ausführung des fehlgeschlagenen und des identifizierten Testfalls beim Modul-Lieferanten oder beim Komponenten-Lieferanten und eine daran anschließende Analyse der Logs und der Testberichte. In diesem Szenario können folgende Fehler festgestellt werden:

- Unterschiedliche Konfigurationen für Konstanten
- Unterschiedliche Stimulierung für das Erzielen desselben Systemverhaltens beim Prüfling
- Unterschiedliche Überprüfung der Anforderung spezifischen Reaktion des Prüflings

Die erste Herausforderung (H1) für die zu entwickelnde kollaborative Fehlersuche konzentriert sich auf die Vergleichbarkeit der Testfälle und lautet: Die aufgeführten Fehlerquellen können nur in den Testfällen erkannt werden, wenn die beiden Testfälle von den *lose gekoppelten* Testprozessen miteinander verglichen werden können. Vergleichen heißt hierbei, dass sowohl die Durchführung einer statischen Analyse als auch das Vergleichen des Laufzeitverhaltens der beiden Testfälle durchgeführt wird. Letzteres wird erzielt, indem die Signalverläufe der ausgeführten Testfälle mithilfe der Logs ausgetauscht werden. Ein Beispiel für den Vergleich von Signalkurven ist in Abbildung 16 gegeben.

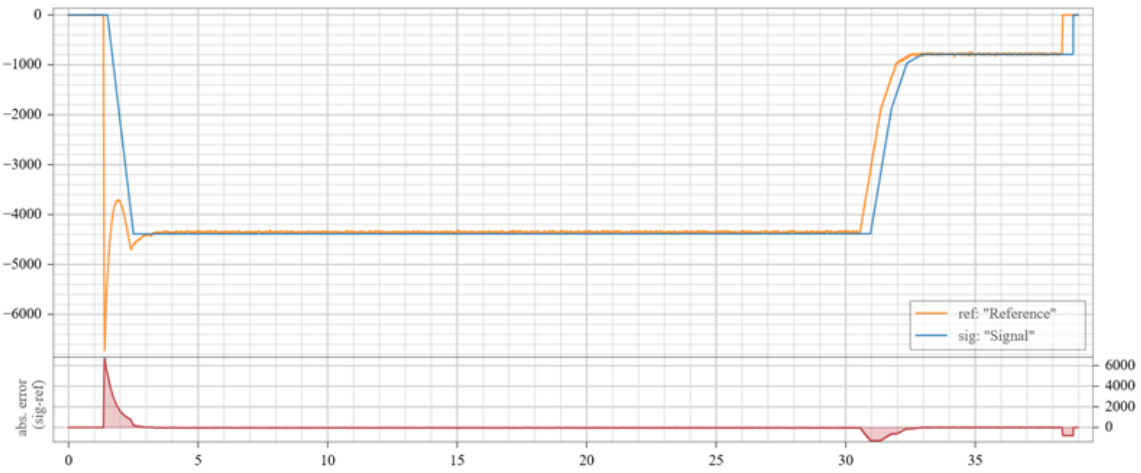

Abbildung 16: Manuelle Auswertung von Signalverläufen

Mithilfe dieses Austausches kann manuell bestimmt werden, ob die Stimulierung des Prüflings bezogen auf die Signale ähnlich auf den unterschiedlichen Testsystemen erfolgt ist. Ein ähnliches Vorgehen lässt sich für die erwartete Systemreaktion durchführen. Wird hier eine Diskrepanz in den Signalverläufen erkannt, ist die Fehlersuche eingegrenzt, aber ohne die statische Analyse der Testfälle nicht auflösbar. Aus den Signalverläufen kann nur die Auswirkung der Stimulierung auf den Prüfling erkannt werden. Daraus kann nicht unmittelbar hergeleitet werden, wie das System stimuliert wurde. Um dies zu ermitteln, wird eine statische Analyse von den Testfällen benötigt.

Die statische Analyse kann nur erfolgen, indem die Testfälle ausgetauscht werden und als Vorverarbeitung syntaktisch angenähert werden. Zum Beispiel müssen Testfälle so angepasst werden, dass Signalnamen gleich heißen und dass die Stimulierung vergleichbar implementiert ist. Die statische Analyse lässt keine eindeutigen Rückschlüsse auf die Signalverläufe des Prüflings zu. Hintergrund ist, dass der Testfall nicht die einzige Quelle in dem Testsystem ist, welche stimulieren kann. Beispiele für zusätzliche Stimulatoren sind Simulationsmodelle, Sensoren oder vordefinierte Startwerte in Konfigurationen. Dementsprechend können die realen Signalverläufe des Prüflings nicht mithilfe der statischen Analyse bestimmt werden, sondern kann nur während der Ausführung des Testfalls beobachtet werden.

Der pragmatische Ansatz der angestrebten kollaborativen Fehlersuche ist die Ausführung des fehlgeschlagenen Testfalls in derselben Testumgebung, in welcher der bestandene Testfall ausgeführt wurde. Das bedeutet, dass der fehlgeschlagene Testfall auf einem anderen Testsystem und mit einer anderen Konfiguration ausgeführt werden muss. Dies ist zurzeit nicht ohne die manuelle Übersetzung als Transformation des Testfalls möglich. Gründe hierfür liegen in der Nutzung unterschiedlicher *Testskriptsprachen* und spezifischer Anpassungen der Testumgebungen.

Zusammengefasst führt die Notwendigkeit des Austausches eines Testfalls innerhalb der Zulieferpyramide zu einem hohen Integrationsaufwand, um den Testfall in die Zielsprache zu überführen. Die Transformation ist ohne manuelle Aufwände und unternehmensübergreifendes Expertenwissen nicht möglich. Hierbei ist Transformation nach [DUDEN 2018]definiert als ein Vorgang der „umwandeln, umformen, umgestalten…“ soll. Im Rahmen der vorliegenden Dissertation wird der Inhalt der Testfälle so umgewandelt werden, dass der Inhalt in die Zielsprache integriert werden kann.

Die Anwendbarkeit des Stands der Technik für eine automatische und lesbare Transkompilierung von Testfällen und der statischen Analyse von Testfällen wird im Nachfolgenden untersucht. Hierfür werden die *Testskriptsprachen* untersucht, inwiefern sich ihre Semantik für eine Transkompilierung und statische Analyse eignen.

3 Herausforderungen für die Umsetzung einer kollaborativen Fehlersuche

Eine kollaborative Fehlersuche wird als testprozessübergreifende Aktivität in der Zulieferpyramide notwendig, sobald lose gekoppelte Testprozesse zu unterschiedlichen Testergebnissen gelangen. In diesem Fall sind die durch die lose Kopplung der Testprozesse entstehenden Abhängigkeiten zu berücksichtigen (siehe 2.3). Insbesondere sind die Anforderungen und die verwendeten *Testskriptsprachen* bei der kollaborativen Fehlersuche zu berücksichtigen.

Ziel der kollaborativen Fehlersuche ist es, Indikatoren für die unterschiedlichen Testergebnisse zu ermitteln. Hierfür ist der fehlgeschlagene Testfall eine wichtige Informationsquelle. Der fehlgeschlagene Testfall enthält alle Informationen darüber, wie eine Anforderung im Prüfling als System under Test (SUT) überprüft werden sollte. Er enthält implizit auch den Hinweis, warum die Testausführung im SUT fehlgeschlagen ist. Dementsprechend ist der Testfall der primäre Input für die Fehlersuche. Dabei ist der Testfall nicht unabhängig von seinem Testprozess, sondern der Testfall ist das Ergebnis des Testprozessschrittes *Realisierung und Durchführung* und referenziert auf andere Elemente innerhalb des Testprozesses. Beispielsweise referenziert der Testfall auf die Anforderung, die der Testfall prüfen soll oder wie die Kommunikation mit dem SUT erfolgen soll.

Im Unterkapitel 3.1 wird zunächst definiert, was eine Information ist und warum die Fehlersuche auf den Austausch von testrelevanten Informationen und nicht auf den Austausch von Testfällen als Daten abzielt. Danach wird detailliert dargestellt, welche Informationen im Rahmen eines Testprozesses entstehen und somit zusätzlich zu den Testfällen für eine Fehlersuche zur Verfügung stehen können. Anschließend werden im Unterkapitel 3.2 die Informationen vorgestellt, welche bereits in einem Testfall enthalten sind und wie heterogen ihre Repräsentation in den *Testskriptsprachen* ist. Im Unterkapitel 3.3 wird die Heterogenität in den *Testskriptsprachen* zusammengefasst. Basierend auf der Heterogenität wird untersucht, ob der Zugriff auf die Informationen für eine kollaborative Fehlersuche durch die Transformation eines Testfalls von einem Datenträger in einen Informationsträger mit dem Stand der Technik möglich ist. Das Ergebnis benennt die Herausforderung für die Datenintegration von Testfällen und konkretisiert hiermit die Problemstellung. Hierbei wird im Folgenden die zweite Herausforderungen **(H2)** an die zu entwickelnde kollaborative Fehlersuche hergeleitet.

3.1 Verfügbare Information im Testprozess

Der Inhalt eines Testfalls beschreibt, wie der Prüfling als SUT getestet werden soll. Diese Informationen sind für den Informationsaustausch zwischen *lose gekoppelten*

Testprozessen relevant, um die Ergebnisse eines Testprozesses nachvollziehen zu können. Zurzeit sind diese Informationen kodiert in verschiedenen *Testskriptsprachen*, welche den Informationsaustausch durch die unterschiedliche Syntax der Sprachen erschwert. In diesem Unterkapitel werden die testrelevanten Informationen identifiziert. Dazu wird in einem ersten Schritt definiert, was eine Information ist und warum der Austausch von Informationen einen Mehrwert gegenüber dem Austausch von Testfällen als Daten hat. Anschließend werden die Informationsquellen vorgestellt, die für die kollaborative Fehlersuche zusätzlich zur Verfügung stehen.

3.1.1 Information

Eine Information sind Daten, welche über die Beziehung zu einem Objekt eine Bedeutung erhalten [Bellinger et al. 2004]. Hierdurch kann die Information für einen gezielten Kontext, nämlich der kollaborativen Fehlersuche, ausgetauscht und angewendet werden. Diese Annahme wird gestützt durch [Endres 2003], indem er eine Information als Daten bezeichnet, die man interpretieren kann. Hierbei wären die Daten der Testfall in einem Dateiformat und die Information wäre, was der Inhalt des Testfalls beim SUT bewirkt. Zum Beispiel kann aus dem Testfall die Information extrahiert werden, welches Signal über die Zeit beim SUT stimuliert wurde. Der Testfall ist in einer *Testskriptsprache* implementiert und somit eindeutig für einen Compiler und Interpreter zu interpretieren. Die Lesbarkeit des Testfalls durch Testingenieure ist allerdings nur gegeben, solange der Testingenieur die *Testskriptsprache* beherrscht. Diese Voraussetzung ist bei der kollaborativen Fehlersuche für *lose gekoppelte* Testprozesse nicht für alle Testingenieure gegeben. Aus diesem Grund stufen wir Testfälle im Rahmen der vorliegenden Dissertation als Daten ein.

Die Anwendung von Informationen führt zur Generierung von Wissen. „Wissen ist die in einem bestimmten Kontext eingebettete Information, die im Individuum wirksam ist und potenziellen Einfluss auf zukünftige Entscheidungen hat.“ [Schindler 2002, S. 34]. In Kontext der Fehlersuche ist das generierte Wissen das Verständnis, wieso der Testfall fehlgeschlagen ist. Die Transformation von Daten zu Wissen ist in der DIKW-Pyramide [Rowley 2007] beschrieben, welche in der Abbildung 17 abgebildet ist.

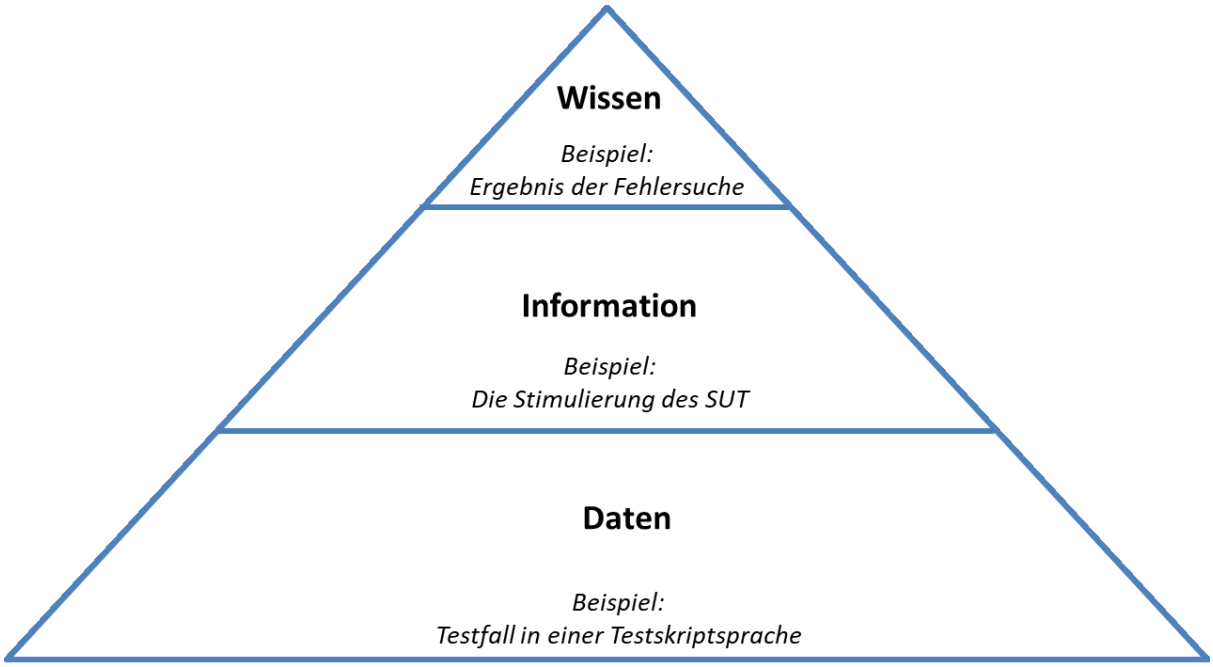

Abbildung 17: DIKW-Pyramide für einen Testprozess

Der Inhalt eines Testfalls muss als Informationen für die Fehlersuche austauschbar sein, wofür ihre Persistenz eine Voraussetzung ist. Dazu sind ein geeignetes Dateiformat und eine geeignete Terminologie erforderlich. Die gewählte Syntax und die unterstützte Semantik müssen für eine Fehlersuche berücksichtigt werden, weil sich hierdurch die anwendbaren Fehlersuchmethoden ergeben. Um die Verwendung der Informationen für mehr als einen Testprozess zu ermöglichen, muss die Transformation von Daten in Informationen erfolgen. Hierfür muss das Level *semantic understanding* erreicht werden, was eine Interpretation von Daten als Informationen ermöglicht [Ören et al. 2007]. Dieses Level umfasst die in der Tabelle 1 aufgeführten Interpretationslevel und beschreibt die extrahierbare Semantik eines Testfalls. Dies fasst die notwendigen Interpretationsfähigkeiten der Testfälle für die Transformation von Daten in Informationen zusammen.

Tabelle 1: Ausschnitt der Interpretationslevel in Anlehnung an [Ören et al. 2007]. Die mit einem * markierten Level sind eigene Erweiterungen.

ID	Level der Interpretation	Beschreibung
1	Lexical understanding	Die lexikalischen Eigenschaften einer Entität verstehen. (Lexikalisch: Dies ist die niedrigste Stufe des Verstehens und ermöglicht unter anderem die Definition von Schlüsselwörtern wie *if* in einer Testskriptsprache.)

2	Syntactic understanding	Das grundlegende Verständnis der syntaktischen Merkmale einer Entität. (Syntaktisch: Damit können u.a. einzelne Anweisungen erkannt werden, wie z.B. eine If-Anweisung, die mit einem *if* beginnt und auf die eine Bedingung folgt.)
3	Morphological understanding	Das grundlegende Verständnis der Struktur (morphologische Merkmale) einer Entität. Dies ermöglicht es, den Aufbau eines Testfalls und seine hierarchische Struktur zu erkennen, z.B. dass eine If-Anweisung in einem Block von Anweisungen enthalten ist.
4	Semantic understanding	Das grundlegende Verständnis der Bedeutung (semantische Eigenschaften) einer Entität. Damit kann die Bedeutung einzelner Anweisungen oder hierarchischer Strukturen definiert werden. Ein Beispiel ist, dass Anweisungen, die mit einem *if* beginnen, den Ablauf der Anweisungen eines Testfalls beeinflussen können.
4.1	Semantic understanding – Structure*	Das grundlegende Verständnis der Bedeutung von hierarchischen Strukturen eines Testfalls, wie z.B. Testschritte.
4.2	Semantic understanding – Anweisungen*	Das grundlegende Verständnis der Bedeutung von Anweisungen.

3.1.2 Informationsquellen für die Fehlersuche

Zentraler Input für die Fehlersuche sind Testfälle, die im Testprozessschritt *Realisierung und Durchführung* entwickelt werden. Anzahl und Inhalt der Testfälle ergeben sich aus den vorgelagerten Testschritten *Planung und Steuerung* sowie *Analyse und Entwurf*. Dementsprechend liefern diese Testschritte Hintergrundinformationen, die bei der Fehlersuche optional genutzt werden sollen. Des Weiteren erfolgt die Bewertung der Testausführung im Testschritt *Auswertung und Reporting* und ermöglicht eine Klassifizierung der Symptome eines fehlgeschlagenen Testfalls. Dabei ist zu beachten, dass die Symptome nicht die Ursache für den fehlgeschlagenen Testfall sein müssen. Die Symptome sind ein zusätzlicher Input für die Fehlersuche. Eine Übersicht über alle weiteren Informationen, die ein Testprozess generiert und die als Dokumente bzw. Daten in eine Fehlersuche eingespeist werden können, ist in Tabelle 2 gegeben, welche auf [Witte 2020; ISTQB Glossary 2021] basiert.

Tabelle 2: Verfügbare Informationen im Testprozess [Witte 2020]

Testphase	Eingabe	Ausgabe
Planung und Steuerung	• Projektziele • Teststrategie • Testrichtlinie • Anforderungen • Systemdokumentation • Infrastruktur	• Testplan
Analyse und Design	• Anforderungen • Teststrategie • Systemdokumentation	• Testspezifikation
Realisierung und Durchführung	• Testspezifikation • Anforderungen • Systemdokumentation • Konfigurationen • Testdaten	• Testfälle • Logs
Bewertung und Berichterstattung	• Anforderungen • Testfälle • Logs	• Testberichte
Abschluss der Testaktivitäten	• Alle Daten aus vorherigen Testphasen	• Archivierung aller Daten vorheriger Testphasen • Dokumente für nachgelagerte Prozesse wie Zertifizierung

3.2 Heterogenität in den Testskriptsprachen

Die Unterkapitel 3.2.1 und 3.2.3 sind aus der Veröffentlichung [Franke und Thoben 2022] direkt übernommen und leicht angepasst worden.

Die Testprozesse in der Zulieferpyramide verwenden unterschiedliche *Testskriptsprachen*. Hierbei ist eine *Testskriptsprache* im Kontext der Arbeit eine domänenspezifische Sprache (DSL), mit der Testfälle als Testskripte für die Automatisierung von Tests ausgeführt werden können. Eine DSL ist hierbei eine Programmiersprache oder eine ausführbare Spezifikationssprache, die auf eine Domäne zugeschnitten ist [van Deursen et al. 2000]. Ein Testskript ist hierbei eine Abfolge von Anweisungen, die durch einen Compiler oder Interpreter ausgeführt werden können [ISTQB Glossary 2021].

Das Ziel der Fehlersuche ist der Austausch von testrelevanten Informationen. Hierfür wird die Transformation eines Testfalls von einem Datenträger in eine Menge von Informationen benötigt. Dabei müssen die Eigenschaften der Testskriptsprache berücksichtigt werden, weil die Eigenschaften wie die Typsicherheit oder Programmierparadigma die Darstellung von Anweisungen definieren. Dementsprechend ist das Ziel der Transformation die Interpretation der Testfälle auf der Ebene des *semantic understanding* zu ermöglichen, was auch die lexikalischen, syntaktischen und morphologischen Merkmale der *Testskriptsprachen* einschließt. Die Herausforderung hierbei ist, dass sich insbesondere die morphologischen Merkmale in den *Testskriptsprachen* unterscheiden. Darüber hinaus sind die *Testskriptsprachen* in Bezug auf Syntax, Semantik und Funktionalität heterogen. Bevor die Heterogenität der *Testskriptsprachen* im Detail beschrieben wird, werden zunächst die Struktur und der Inhalt eines minimalistischen Testfalls für HIL-Tests vorgestellt. Hierbei spiegelt der Inhalt eines minimalistischen Testfalls auch die minimale Informationsmenge wider, die der kollaborativen Fehlersuche als Eingabe zur Verfügung gestellt werden müssen.

3.2.1 Der Aufbau eines Testfalls

Der Aufbau und der Inhalt eines Testfalls ist nicht willkürlich, sondern ergibt sich aus seiner Aufgabe im Testprozess und dem zu testenden Prüfling. Als Nächstes wird der Aufbau eines Testfalls für das Testen von mechatronischen Systemen für HIL-Tests herausgearbeitet, was auch die darin enthaltenen Informationen für die Eingabe der kollaborativen Fehlersuche definieren wird.

Ein Testfall besteht aus einer Sequenz von Anweisungen. Das Ziel der Sequenz ist es, im Rahmen einer Testausführung den Nachweis zu erbringen, dass das SUT entsprechend den zugehörigen Anforderungen funktioniert. Zu diesem Zweck werden beim Testen von Informationssystemen die spezifizierten Testdaten, die erwarteten Ergebnisse und eine Beschreibung der zu testenden Schritte mit den erwarteten Ergebnissen erfasst und als Sequenz von Anweisungen dargestellt. [Pilorget 2012]. Eine deckungsgleiche Definition des Testfalls ergibt sich aus dem ISTQB und dem IEEE 829 für Software. Hiernach umfasst ein Testfall die Angaben „*… für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts), die Menge der vorausgesagten Ergebnisse, sowie die erwarteten Nachbedingungen. Testfälle werden entwickelt im Hinblick auf ein bestimmtes Ziel bzw. auf eine Testbedingung, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen…*“ [ISTQB Glossary 2021].

Beide Definitionen führen die definierten Testdaten und die erwarteten Ergebnisse als Bestandteile eines Testfalls ein. Die Testdaten werden auf verschiedene Weise eingegeben und führen zur Stimulierung des SUT. Im Kontext der Arbeit sind bei-

spielsweise die Komponenten und die Module der Prüfling und werden als SUT getestet. Basierend auf der Stimulation werden Ergebnisse erwartet, wie sich das SUT entsprechend der Anforderung verhalten soll. Die erwarteten Ergebnisse werden als Bedingung definiert und die Überprüfung der Bedingung wird mit Hilfe von Anweisungen vom Typ Verdikt realisiert. Beispiele für Verdikte sind z.B. *expect* in CCDL [Wittner 2014] oder *result* in RTT [dSPACE 2023].

Ein mechatronisches System ist ein Verbund verteilter, ereignisdiskreter Systeme [Bolton 1997], das ein Informationssystem ist und seine Steuerungsfunktion durch Software realisiert [Campetelli und Broy 2018]. Dementsprechend enthalten Testfälle für das Testen mechatronischer Systeme sowohl Anweisungen zur Stimulierung des SUT als auch Verdikte. Die Stimulierung eines mechatronischen Systems kann nur über Signale erfolgen, welche über Schnittstellen zugänglich sind. Ebenso können die Bedingungen für die Verdikte nur über Signale definiert werden, die über die Schnittstelle erreichbar sind. Die Integration von Software und Aktorik führt dazu, dass die Stimulation eines Signals nicht sofort umgesetzt wird, sondern systemabhängig Zeit benötigt. Dieses Zeitverhalten ist in den Testfällen zu berücksichtigen. Dazu werden Anweisungen zur Steuerung des Zeitverhaltens bereitgestellt.

Der Aufbau eines minimalistischen Testfalls wurde für das Testen von mechatronischen Systemen durch [Rasche et al. 2018] [Franke und Thoben 2022] definiert und ist in der Abbildung 18 abgebildet.

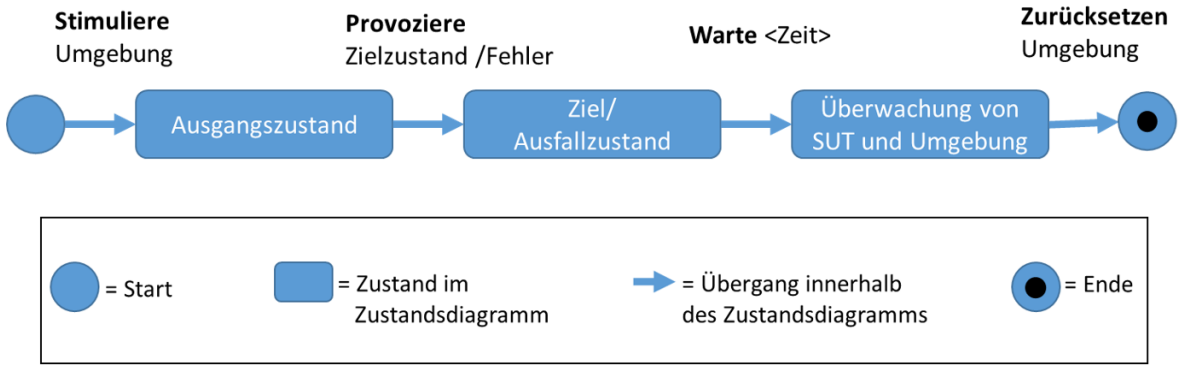

Abbildung 18: Der gleiche Grundaufbau eines Testfalls, angelehnt an [Franke und Thoben 2022]

Wie in der Abbildung 18 dargestellt, stimuliert ein Testfall drei aufeinander folgende Zustände des SUT. Zunächst stimuliert der Testfall die Umgebung. Dann wird der Fehler- oder Zielzustand stimuliert und schließlich wird das SUT beobachtet. Diese Abfolge von Zuständen wird erreicht, indem der Testfall als Sequenz von Anweisungen von einem Testsystem ausgeführt wird. Die Anweisungen, die den Zustand ändern, sind in der Abbildung durch Pfeile gekennzeichnet. Um zu überprüfen, ob

das SUT einen Zielzustand erreicht hat oder ob die Testausführung lange genug pausiert hat, werden die Anweisungen innerhalb des Zustands ausgeführt.

Ein realistischer Testfall umfasst neben Zuständen und Übergängen auch Testschritte und testsystemspezifischen Funktionen [Franke et al. 2019]. Der minimalistische Testfall beinhaltet bereits unterschiedliche Typen von Anweisungen, welche unterschiedliche Aufgaben erfüllen. Die Abbildung des Grundaufbaus eines minimalistischen Testfalls auf die Anweisungstypen der *Testskriptsprachen* ist in [Franke und Thoben 2022] beschrieben und ist in Abbildung 19 zu sehen.

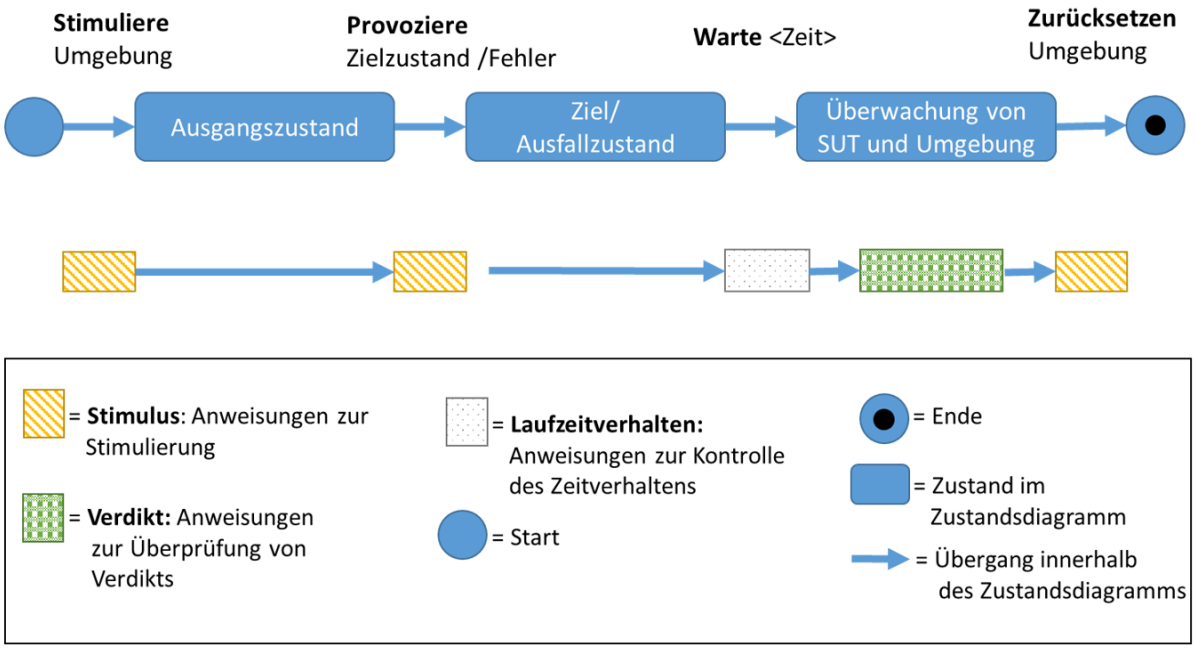

Abbildung 19: Abbildung des Grundaufbaus von Testfällen auf die Anweisungstypen [Franke und Thoben 2022]

Hierbei ist zu erkennen, dass für die Erreichung des initialen Zustands, des Zielzustands und der Zurücksetzung der Testumgebung Anweisungen des Typs *Stimulus* verwendet werden. Hierbei kann es sich beispielsweise um die Wertzuweisung eines Signals oder auch um eine Funktion handeln, welche die Signalwerte über die Zeit ändert. Das Warten des Testfalls auf die zu prüfende Reaktion des SUT wird mithilfe von Anweisungen des Typs *Laufzeitverhalten* erzielt. Schlussendlich wird die Reaktion des Systems mithilfe von Anweisungen des Typs *Verdikt* ausgewertet.

Die oben aufgeführten Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* sind für die Bereitstellung der Grundfunktionalität eines Testfalls notwendig und sind in allen *Testskriptsprachen* verfügbar. Die gängigsten *Testskriptsprachen* für das Testen von mechatronischen Systemen, wie Autos und Flugzeuge, sind ASAM XIL [ASAM 2023], TTCN-3, CCDL [Wittner 2014], C [ISO/IEC 9899:2018 2018], RTT [dSPACE 2023] und Python [Python.org 2023].

Die Anweisungstypen ermöglichen nicht nur die Gruppierung der Anweisungen eines Testfalls, sondern beschreiben direkt zu welchem Zweck die Anweisungen verwendet werden. [Bellinger et al. 2004] definiert eine Information als Daten, die durch ihre Beziehung zu einem Objekt eine bestimmte Bedeutung besitzen. Die Verbindung einer Anweisung mit ihrem Verwendungszweck erfüllt die Bedingung einer Information. Dementsprechend können die Anweisungen der *Testskriptsprachen* zu Anweisungstypen zugeordnet werden. Diese Zuordnung bestimmt die relevanten Informationen für die Fehlersuche. Die Darstellung der Anweisungen unterscheidet sich zwischen den *Testskriptsprachen*. Ein Grund hierfür liegt in der Struktur der *Testskriptsprachen*. Im Folgenden wird die Heterogenität der *Testskriptsprachen* bezogen auf die lexikalischen, syntaktischen und morphologischen Merkmale untersucht. Dies ist notwendig, um die Anforderungen für eine Transformation einer Anweisung aus einer *Testskriptsprache* in eine Information ableiten zu können.

3.2.2 Struktur der Testskriptsprachen

Testskriptsprachen können unterschiedlich klassifiziert werden, woraus sich die Heterogenität ableiten lässt. Dazu wird die Struktur der Sprachen anhand ihrer Typsicherheit und ihrem zu Grunde liegenden Programmierparadigma beschrieben. Andere Eigenschaften, wie beispielsweise Programmierkonzepte, werden in der Arbeit nicht explizit herausgearbeitet. Als Nächstes wird zuerst die Typsicherheit und anschließend das Programmierparadigma vorgestellt.

Eine sichere Sprache verhindert, dass die Sprache missbraucht werden kann. Dazu werden zusätzliche Mechanismen, wie z.B. Typsicherheit von Variablen oder spezielle Zugriffsmethoden für Datenstrukturen hinzugefügt, um einen inkonsistenten Zustand in der internen Repräsentation der Programmausführung zu vermeiden. Ein Beispiel für eine solche Inkonsistenz wäre, wenn die Anfänge zweier Arrays im Arbeitsspeicher überlappende Speicheradressen hätten und sich gegenseitig beeinflussen könnten. Im Gegensatz dazu haben unsichere Sprachen diese Einschränkungen nicht. Die Überprüfung der Programme auf Korrektheit kann sowohl statisch als auch dynamisch zur Laufzeit erfolgen. Tabelle 3 gibt einen Überblick über gängige Programmiersprachen und die oben genannten *Testskriptsprachen*.

Tabelle 3: Einordnung der Programmiersprachen, basierend auf [Pierce 2002] und übernommen von [Franke und Thoben 2022]

	Statisch geprüft	**Dynamisch geprüft**
Sicher	ML, Haskell, Java, CCDL, FLATSCRIPT, XIL	Lisp, Scheme, Perl, Postscript

Unsicher	C, C++	Python, RTT

3.2.2.1 Einfluss der Typisierung auf die Syntax

Die Einordnung einer *Testskriptsprache* in eine der beiden Gruppen hat direkten Einfluss darauf, welche Metainformationen über eine Variable vor ihrer ersten Verwendung bekannt sein müssen und welche Funktionen für Datenstrukturen möglich sind. So muss in sicheren Sprachen zu einer Variablen der Datentyp, die Sichtbarkeit und ein Initialwert bekannt sein. Im Gegensatz dazu sind diese Informationen in einer unsicheren Sprache nicht zwingend erforderlich. Ein entsprechendes Beispiel ist in der Abbildung 20 gegeben.

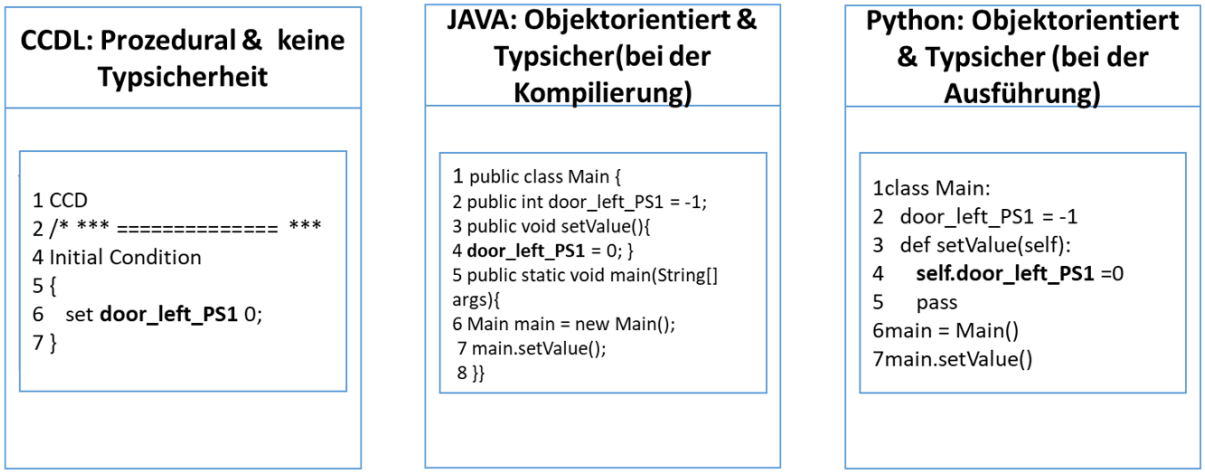

Abbildung 20: Beispiel für Unterschiede zwischen sicheren und unsicheren Sprachen

Ähnliches gilt für die Parameterübergabe in Methoden, bei der Definition von Klassen und Interfaces. Die genannten Unterschiede sind Beispiele für die Heterogenität bezogen auf der Definition und Nutzung von Variablen, Klassen und Interfaces (Datensicht). Aus der Perspektive des Testingenieurs enthalten die Variablen die Information, wie die Signale des SUT stimuliert werden soll (Informationsmodell).

3.2.2.2 Einfluss des Programmierparadigma auf die Syntax

Abgesehen von der Typisierung beeinflussen die grundlegenden Eigenschaften der Sprache, die von den verwendeten Programmierparadigmen und der Datenkapselung abhängen, die grundlegende Struktur jedes Testfalls in der Sprache. Gängige Programmierparadigmen im Testumfeld sind prozedural und objektorientiert. Beispiele für die Grundstruktur eines Testfalls in einer prozeduralen und einer objektorientierten Sprache sind in Abbildung 21 dargestellt. Diese Unterschiede sind auch bei der Heterogenität eines Testfalls zu berücksichtigen.

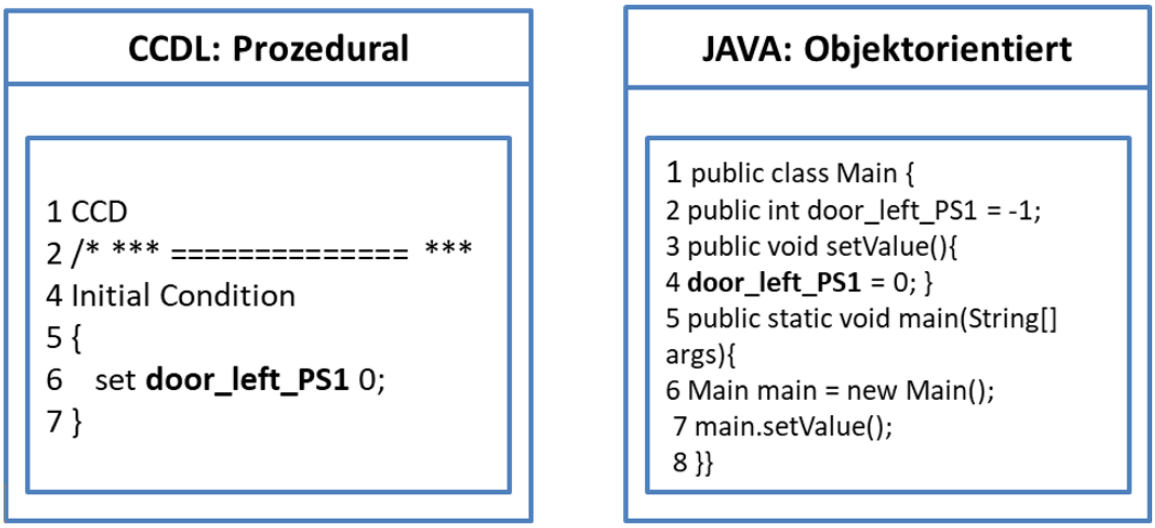

Abbildung 21: Beispiel für den Grundaufbau von Testfällen für die Stimulierung eines Signals

3.2.2.3 Einfluss des Laufzeitverhaltens auf die Struktur von Testfällen

Testskriptsprachen werden zur Ausführung entweder mit einem Compiler in Maschinencode übersetzt oder mit einem Interpreter interpretiert. Ein Beispiel für eine kompilierte Sprache ist C, ein Beispiel für eine interpretierte Sprache ist Python. Die unterschiedliche Art der Testfallausführung hat einen Einfluss auf die Echtzeitfähigkeit der Testfälle, wobei kompilierte Sprachen näher an der Hardware ausgeführt werden und ihre Anweisungen entsprechend optimiert auf der Hardware ausgelöst werden können.

Echtzeit ist definiert als *„den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“* [Scholz 2006, S. 39].

Eine *Testskriptsprache* ist dementsprechend echtzeitfähig, sobald die Ausführung einer Anweisung garantiert in einer vorgegebenen Zeitspanne verarbeitet worden ist. Diese Eigenschaft ist wichtig, sobald der Testingenieur mechatronische Systeme testen muss, in denen Monitore die Sicherheit des Systems überwachen und diese innerhalb von Millisekunden reagieren müssen. Ein Beispiel hierfür ist ein möglicher Wellenbruch im Hochauftriebssystem eines Flugzeugs (siehe Abbildung 1).

Das daraus resultierende unterschiedliche Laufzeitverhalten der *Testskriptsprachen* bestimmt, welche Anweisungen der Sprache in Echtzeit ausgeführt werden können. Zusätzlich wird definiert, wie viele Anweisungen in einem Zeitfenster parallel un-

terstützt werden können. Diese beiden Kriterien beeinflussen den Aufbau eines Testfalls, der folgend exemplarisch anhand der Sprachen CCDL und Python erläutert wird.

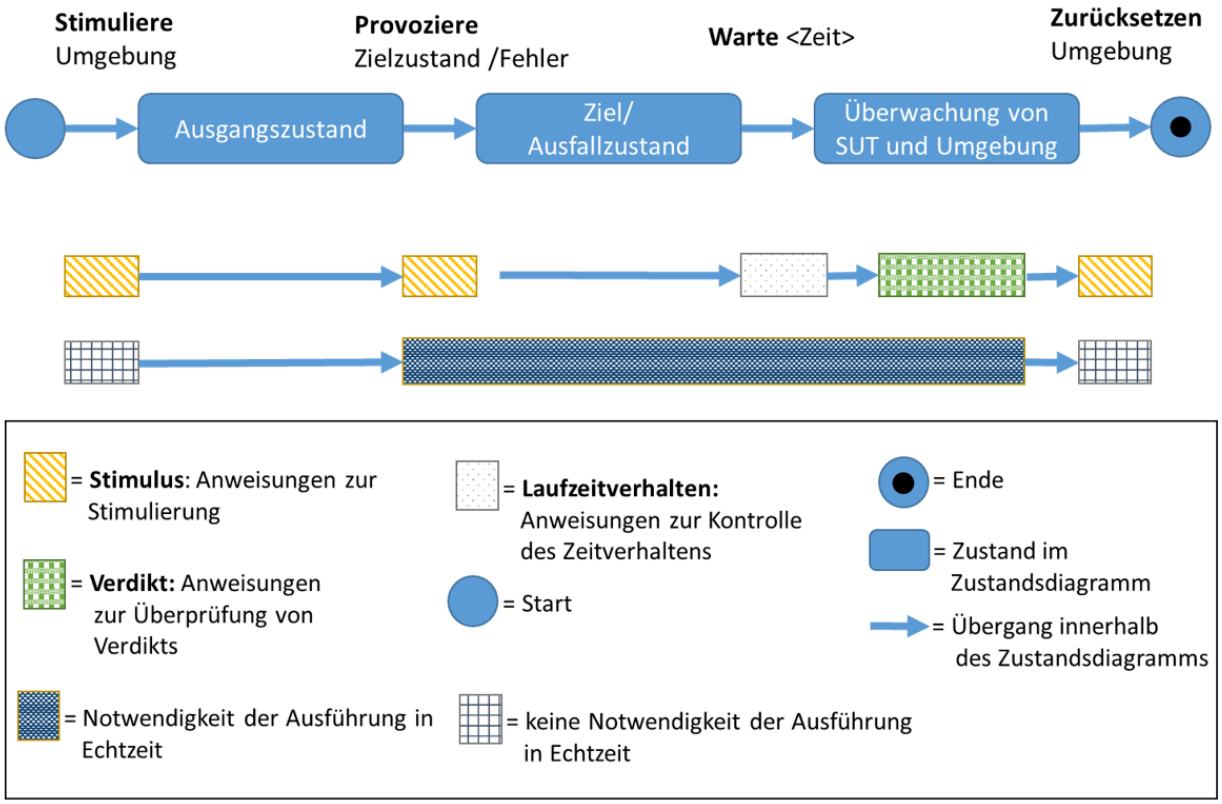

Abbildung 22: Zeitkritikalität innerhalb eines Testfalls

Jeder Testfall, wie in Abbildung 22 dargestellt, besteht aus vier Phasen. In der ersten Phase werden alle für den Test relevanten Systeme, wie z.B. die Datenaufzeichnung, initialisiert und das SUT in den Zustand gebracht, von dem aus der Test gestartet werden soll. Zum Beispiel muss für das Testen eines Blinkers am Auto die Stromversorgung eingeschaltet und der Blinker in Ausgangsstellung sein. Anschließend wird das SUT mit eigenen Bordmitteln in den für den Test relevanten Zustand gebracht. Dazu müssen entsprechende Signale in einem definierten zeitlichen Kontext stimuliert werden. Das Auslösen der Signale in einem definierten Zeitfenster, das bei jeder erneuten Ausführung des Testfalls gleich sein soll, erfordert die Ausführung der Anweisungen in harter Echtzeit. Diese Anforderungen sind z.B. für das Testen von Wellenbrüchen in sicherheitskritischen Systemen, bei denen wenige Millisekunden über den Totalausfall eines Systems entscheiden, sehr wichtig. Die gleichen harten Echtzeitanforderungen gelten dann auch für die Überwachung des SUT im Falle eines stimulierten Fehlers. Hierbei wird u.a. überprüft, ob die Sicherheitsfunktionen des mechatronischen Systems rechtzeitig und im richtigen Zeitfenster ansprechen. An dieser Stelle ist zu erwähnen, dass nicht alle Module eines Autos oder eines Flugzeugs in Echtzeit getestet werden müssen.

Die Stimulus- und Überwachungsmodellierung basiert auf Anweisungen zur Erzeugung spezifischer Signalverläufe und anschließender Überwachung der Signalverläufe. In der Sprache CCDL können diese direkt an der logischen Stelle des Testfalls integriert werden und die Auswertung der Signalverläufe erfolgt direkt beim Auslösen der Anweisungen. Dadurch wird die Auswertung im Bereich der zeitkritischen Teile des Testfalls durchgeführt. Die Möglichkeit der direkten Verwendung der Anweisungen ist in Python für Anweisungen mit harter Echtzeit nicht möglich. Dafür ist der Rechenaufwand zu groß. Die Auswirkungen der unterschiedlichen Fähigkeiten zur Unterstützung echtzeitkritischer Anweisungen führen zu unterschiedlichen Strukturen der Testfälle. Ein Beispiel für die Stimulierung eines Signalverlaufs mithilfe einer Rampe ist in Abbildung 23 gegeben.

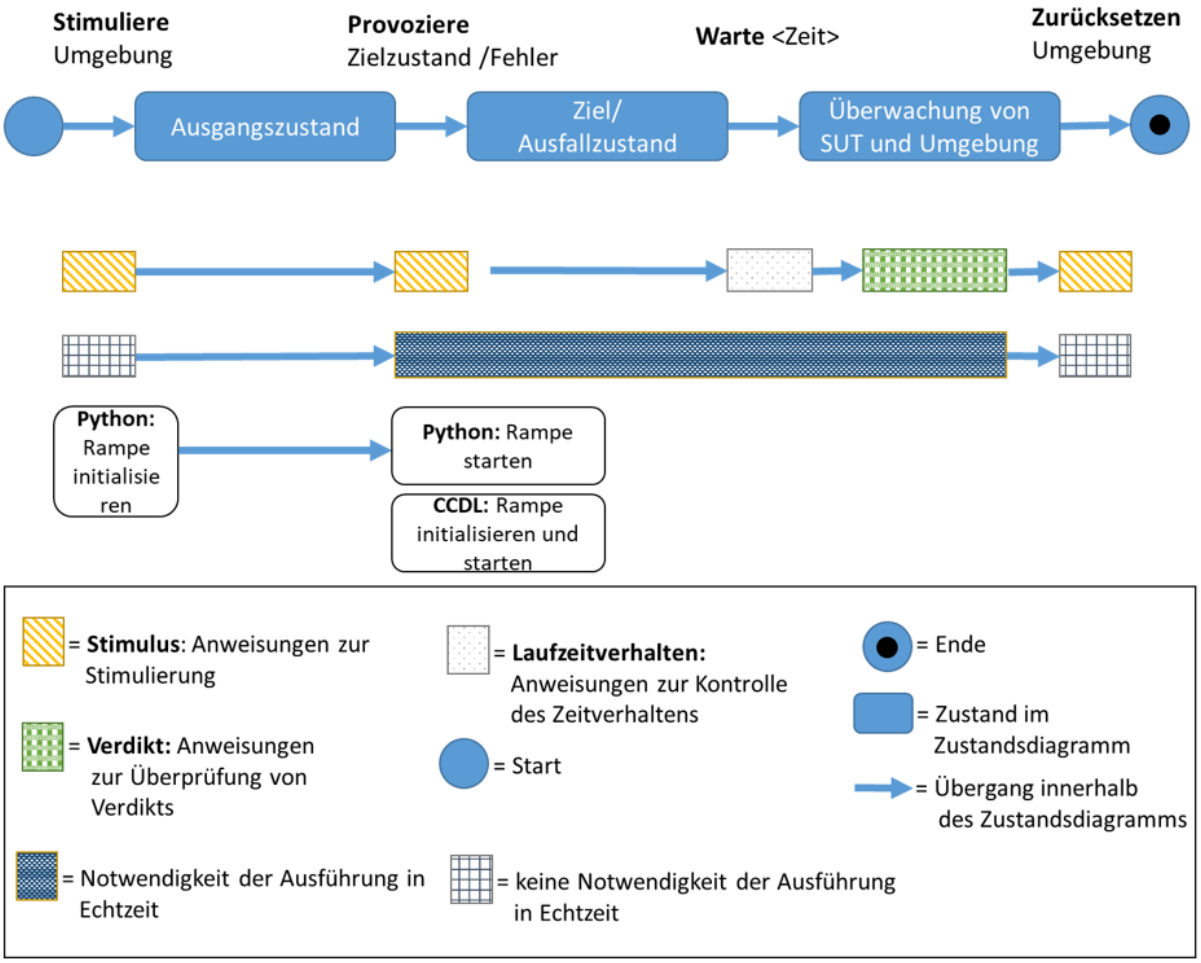

Abbildung 23: Zeitkritikalität von Anweisungen in Testskriptsprachen

Die Modellierung von echtzeitkritischen Anweisungen in Testfällen unterscheidet sich nach Anweisungstypen (Informationen) und Sprachen. Eindeutige Modellierungsregeln für Testskriptsprachtypen gibt es nicht.

3.2.3 Sprachumfang der Testskriptsprachen

Ein minimalistischer Testfall ist in Abbildung 21 dargestellt. Dieser definiert die wiederkehrenden Schritte und die darin enthaltenden Anweisungstypen. Die *Testskriptsprachen* bieten für die identifizierten Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* verschiedene Methoden und Funktionen an. Die Heterogenität der *Testskriptsprachen* bezogen auf ihre Struktur erschwert die Analyse des Sprachumfangs für diese Anweisungstypen, was für das Ziel von interoperablen Testfällen notwendig ist. Dementsprechend wird eine gemeinsame Perspektive auf die *Testskriptsprachen* benötigt, um eine Evaluation zu ermöglichen. Aus diesem Grund wird der Sprachumfang der *Testskriptsprachen* aus der Perspektive eines Zustandsübergangsdiagramms erfolgen.

Die Darstellung eines Testfalls kann immer auf ein Zustandsübergangsdiagramm zurückgeführt werden. Diese Annahme wird durch Methoden des modellbasierten Testens gestützt, die Zustandsübergangsdiagramme als Aggregation von Testfällen verwenden [Utting et al. 2012].

Als Nächstes wird der Sprachumfang aus der Sicht eines Zustandsübergangsdiagramms, das aus verbundenen Zuständen besteht, beschrieben. Hierfür wird die Definition der Zustände und Transitionen eingeführt. Anschließend werden die Anforderungen an die für den operationellen Testprozess in der Luftfahrt benötigten Funktionen für die Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* dargestellt. Abschließend wird auf Basis der Anforderungen ein Überblick über den Sprachumfang der ausgewählten Sprachen ASAM (genauer XIL-Signalbeschreibungsdefinitionen), ATML, TTCN-3, CCDL und FlatScript II gegeben.

3.2.3.1 Definitionen

Zustand

Ein Zustand wird durch die Informationen definiert, die zu einem bestimmten Zeitpunkt gültig sind. Bei der Ausführung eines Testfalls ergibt sich der Zustand aus dem SUT und dem Testsystem.

In Testfällen werden Zustände durch den Satz von Parameterwerten und Signalen des SUT sowie die aufgerufenen Testsystemfunktionen definiert. Die konkrete Zuweisung von Werten zu Parametern oder Signalen ist sprachspezifisch und unterscheidet sich zwischen den *Testskriptsprachen*. Während es syntaktische Unterschiede gibt, hat die Wertzuweisung für die Stimulierung des SUT eine ähnliche Semantik. Beispielsweise wird die Zuweisung in gängigen *Testskriptsprachen* mittels eines Schlüsselworts (z.B. *set* oder *put*) oder durch Operatoren (z.B. = oder :) durchgeführt. Zusätzlich zu diesen Informationen kann ein Zustand auch Aussagen

enthalten, die seine Gültigkeit definieren. Beispielsweise können Einschränkungen für Zeitüberschreitungen und Wertebereiche von Parametern definiert werden.

Testfälle können auch zusammengesetzte Zustände verwenden, um Einschränkungen, die für mehr als einen Zustand gelten, zusammenzufassen. Zusammengesetzte Zustände werden verwendet, um Zustände in logische Blöcke zu kapseln.

Transition

Transitionen werden verwendet, um Zustandsänderungen zu modellieren. In einem Testfall muss jede Zustandsänderung einen Grund haben. Zu den häufigsten Gründen gehören u.a. ausgelöste Ereignisse, nicht mehr gültige Bedingungen und verstrichene Zeit. Beispielsweise wird der Zeitablauf in *Testskriptsprachen* durch sprachspezifische Wertzuweisungen wie *wait 5* oder *Thread.sleep* (5) und benutzerspezifische Funktionen implementiert.

3.2.3.2 Grundlegende Anforderungen an Testskriptsprachen

In diesem Abschnitt werden die detaillierten Anforderungen an einen *Zustand* und einer *Transition* für einen Testfall aufgeführt. Die detaillierten Anforderungen wurden aus den Funktionen der *Testskriptsprachen* ASAM (genauer XIL-Signalbeschreibungsdefinitionen), ATML, TTCN-3, CCDL und FlatScript II für die Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* extrahiert. Die ausgewählten Sprachen decken hierbei sowohl die Anwendungsdomänen Automotive (ASAM XIL API) und Luftfahrt (CCDL, FlatScript II) als auch Kommunikationsprotokolle (TTCN-3) ab. Zusätzlich wurde in die Auswahl der *Testskriptsprachen* Kandidaten aufgenommen, die harte Echtzeitanforderungen erfüllen (wie *CCDL*) und die keine Echtzeitanforderungen besitzen (wie FlatScript II).

Die Anforderungen wurden direkt aus der Veröffentlichung [Franke et al. 2019, 2019] übernommen und sind das Ergebnis des Forschungsprojektes STEVE [BIBA - Bremer Institut für Produktion und Logistik 2018].

Tabelle 4: Mindestanforderungen an eine *Testskriptsprache*

IDs	Requirement
State-specific requirements	
ST1	Assigning of parameters, including values and their units, to states
ST2	Assigning of active test system functions to a state

ST3	Assigning of specific conditions like timeouts, events, and conditions
ST4	Creating of compound states to summarize states and enable cross-state conditions
ST5	Enabling parallel states to support concurrency
Transition-specific requirements	
TR1	Transition without restriction must be available
TR2	Transition with a timing restriction must be available
TR3	Transition with an event restriction must be available
TR4	Transition with conditions defined by a parameter value must be available
Test-specific functions	
SF1a	Stimulation or failure injection of a parameter/signal via predefined shapes (ramp, sine, pulse, etc.)
SF1b	Stimulation or failure injection by adding an offset to a parameter variable/signal
SF2	Failure injection via manipulation of hardware pins
SF3	Logging parameter values/signals during a specific interval
SF4	Enabling requirement traceability

Die aufgeführten Anforderungen definieren die Mindestanforderungen an eine *Testskriptsprache*, welche für das Testen von Schnittstellen im Rahmen von Black-Box Tests im Bereich HIL für Informationssysteme benötigt werden. Zusätzlich wurden für die Anforderungen SF1a, SF1b und SF3 in dem Forschungsprojekt AGILE-VT [dSPACE GmbH 2021] weitere Anforderungen spezifiziert und einen Befehlssatz erarbeitet, die für HIL-Tests in Bereich der Luftfahrt erforderlich sind.

Tabelle 5: Testrelevanter Befehlssatz für HIL-Tests in der Luftfahrt

ID	TASCXML-Befehlssatz	Beschreibung
1	<tascxml:set>	Setzt ein Signal auf einen bestimmten Wert
2	<tascxml:get>	Liest den aktuellen Wert eines Signals
3	<tascxml:result>	Protokolliert ein Ergebnis (Urteil) des Testfalls

4	<tascxml:ramp>	Löst eine Rampe aus, die ein Signal für eine bestimmte Dauer ändert
5	<tascxml:sine>	Erzeugt eine Kurvenform einer Sinuskurve
6	<tascxml:sawtooth>	Erzeugt eine Wellenform eines Sägezahns
7	<tascxml:pulse>	Erzeugt eine Wellenform eines Pulses
8	<tascxml:verifytolerance>	Überprüft, ob der Wert eines Signals innerhalb eines bestimmten Toleranzbereichs liegt,

Eine Analyse der oben gelisteten *Testskriptsprachen* ergab, dass sie die Anforderungen mit Ausnahme der Anforderungen SF2 und SF4 erfüllt werden können.

3.2.3.3 Heterogenität von Testskriptsprachen

Die Tabelle 4 fasst die Mindestanforderungen, mit Ausnahme der Anforderungen SF2 und SF4, an *Testskriptsprachen* für HIL-Tests in der Luftfahrt und in Automotive zusammen. Hierdurch könnte der Eindruck entstehen, dass die Sprachen untereinander in Bezug auf den gemeinsamen Funktionskatalog homogen sind und entsprechend direkt in Informationen transformiert werden können. Diese Annahme trifft nicht zu, weil die Spezifikation und Implementierung der jeweiligen Funktionen sich zwischen den Sprachen unterscheiden. Hierdurch entsteht im gemeinsamen Funktionskatalog eine Heterogenität, die eine eindeutige Abbildung der Sprachen untereinander nicht ermöglicht.

Als Beispiel wird die Heterogenität an der Funktion *ramp,* welche den Anweisungstyp *Stimulus* besitzt, für die *Testskriptsprachen* CCDL [Wittner 2014] und ASAM XIL API [ASAM 2023] in Abbildung 24 gezeigt. Diese beiden *Testskriptsprachen* wurden ausgewählt, weil CCDL primär in der Luftfahrt und XIL API primär im Automotive eingesetzt werden.

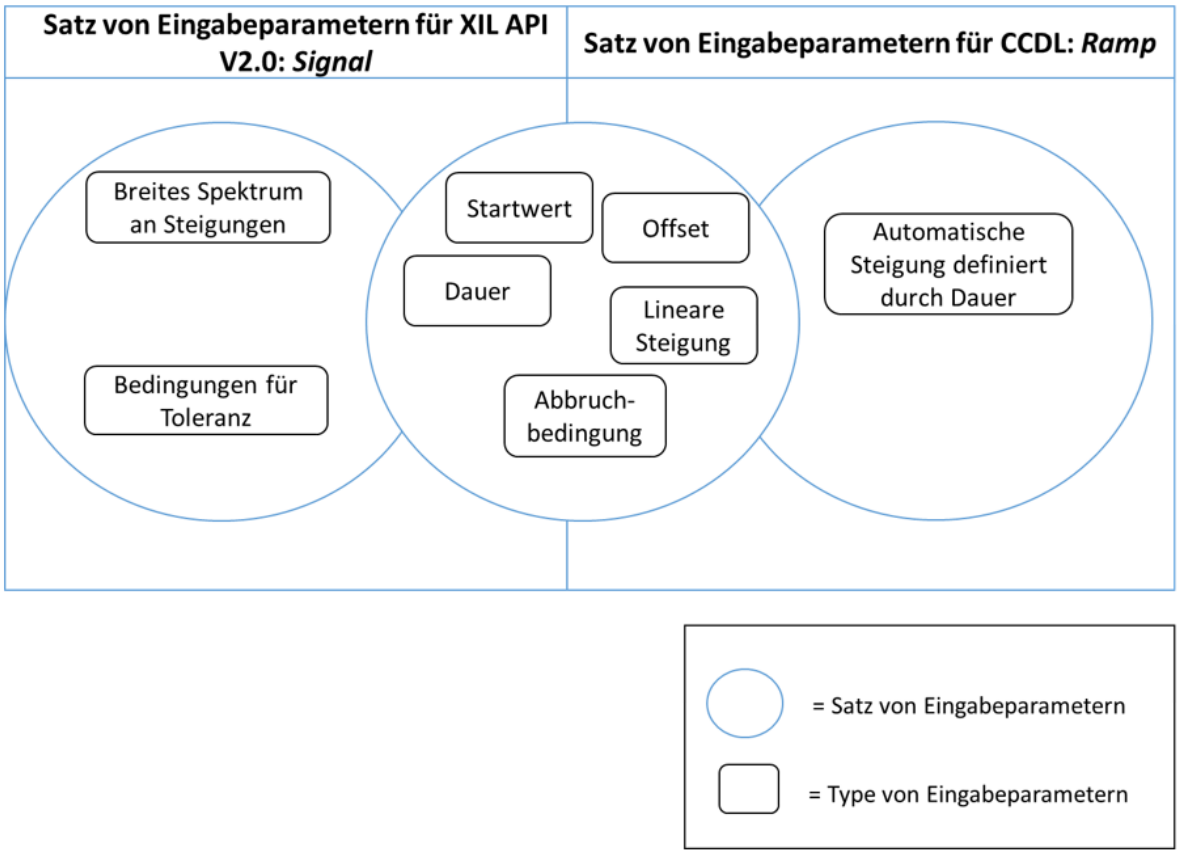

Abbildung 24: Heterogenität der Rampenfunktion in ASAM XIL API und CCDL

Diese Heterogenität lässt sich auch für die anderen aufgeführten Funktionen und Strukturmerkmale aus den Tabellen Tabelle 4 und Tabelle 5 zeigen. Die Überwindung der Heterogenität ist für den Informationsaustausch innerhalb der Zulieferpyramide notwendig. Hierfür wird eine Übersicht benötigt, welche Eigenschaften die jeweiligen Funktionen und Strukturmerkmale besitzen. Eine Übersicht der aus den Anforderungen ableitbaren Eigenschaften ist in Abbildung 25 zu finden.

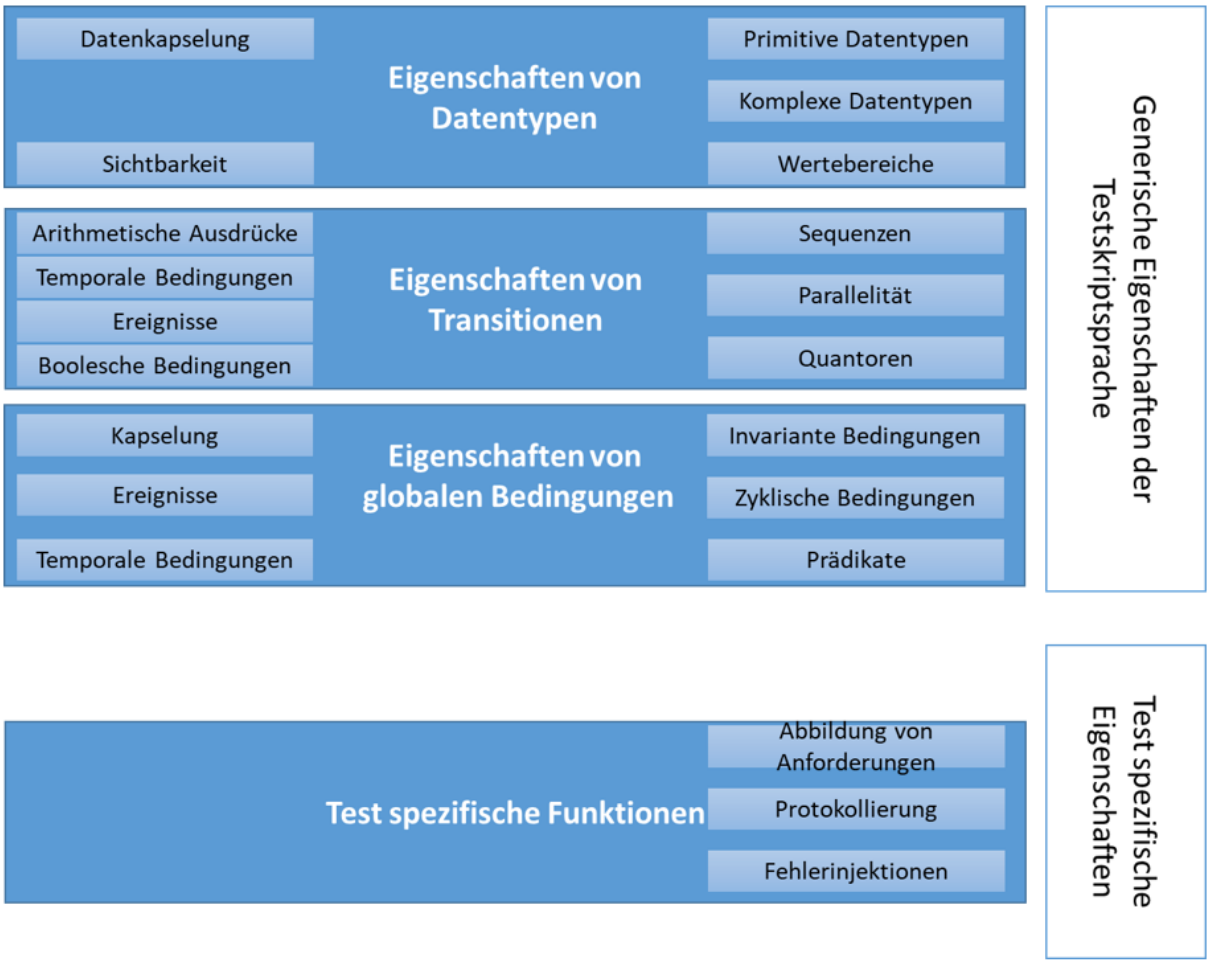

Abbildung 25: Eigenschaften von *Testskriptsprachen*

Anhand der in Abbildung 25 aufgeführten Eigenschaften aus den zwei Gruppen kann der Sprachumfang einer *Testskriptsprache* beschrieben werden, welche die oben aufgeführten Anforderungen berücksichtigt. Im Folgenden werden für die oben ausgewählten *Testskriptsprachen* und ausgewählten Modellierungssprachen, welche die Abbildung von Zustandsübergangsdiagrammen ermöglichen, der Sprachumfang präsentiert.

Tabelle 6: Übersicht über Datentypen in *Testskriptsprachen*

Testskriptsprache	Komplexe Datentypen	Primitive Datentypen	Wertebereiche: Float/Double/Long	Wertebereich: Integer	Unterstützte Zeichensätze für Strings/Größe	Datenkapselung	Sichtbarkeit
State Chart XML(SCXML)	J	N, Erweiterbar.	N, Erweiterbar	N, Erweiterbar	N, Erweiterbar	J	Global, lokal
UML, fokussiert auf State Machine inklusive OCL	J	J	N, Erweiterbar	N, Erweiterbar	N, Erweiterbar	J	Global, lokal
TTCN-3	J	J	J, definiert von IEEE754	64 Bit	8 Bit bis UTF-16	J	J
Automatic Test Markup Language (ATML)	J	J	64 Bit f	32 Bit	String	J	J
XIL-API (ASAM)	N	J	64 BIT	J	J	N	N
CCDL	N	J, aber kein String	J, definiert von IEEE754	J, definiert von IEEE754	N	N	N
FlatScript II	N	J, aber kein String	Float: 32 Bit, Double 64 Bit	32 Bit	N	N	N

Tabelle 7: Übersicht über Transitionen in *Testskriptsprachen*

Testskriptsprache	Sequenzen	Parallelität	Maximale Anzahl an Transitionen	Transition	Transition boolesche Bedingung	Arithmetische Ausdrücke	Trinomische Funktionen	Quantoren	Temporale Bedingungen	Ereignisse	Aktionen
State Chart XML(SCXML)	J	J	?	J	(J)	(J)	(J)	(J)	J	J	J
UML State Machine inclusive OCL	J	J	?	J	J	J	N	N	J	J	J
TTCN-3 (Graphical Format, GFT)	J	J	?	J	J	J	N, Erweiterbar	J	J	J	J
Automatic Test Markup Language (ATML)	J	J	?	J	J	N, Erweiterbar	N, Erweiterbar	N, Erweiterbar	J	J	J
XIL API (ASAM)	J	J	?	J	J	J	J	?	J	J	J
CCDL	J	J	1	J	J	J	J	(J)	J	J	J
Flatscript II	J	J	?	J	J	?	?	N	J	N	?

Tabelle 8: Übersicht über globale Bedingungen in *Testskriptsprachen*

Testskriptsprache	Invariant Bedingungen Beispiel: „Winkel ist nie größer als 23 °“	Zyklische Bedingungen Beispiel: „Alle 10 ms muss der Sensor aktualisiert werden“	Bedingte Bedingungen Beispiel: „Wenn ein Fehler auftritt, muss die zweite Konfiguration geladen werden“	Temporale Bedingungen Beispiel: „Nach 50s sollte das System gestartet und betriebsbereit sein“
State Chart XML(SCXML)	J	J	J	?
UML State Machine inclusive OCL	J	N	?	N
TTCN-3 (grafische Repräsentation (Graphical Format, GFT)	J	J	J	J
Automatic Test Markup Language (ATML)	J	J	J	J
XIL API (ASAM)	J	J	J	J

Tabelle 9: Test spezifische Funktionen in *Testskriptsprachen*

Testskriptsprache	Anforderungen	Protokollierung	Abbildung von physikalischen Störgrößen
State Chart XML(SCXML)	N, indirekt über einen Kommentar	N	N
UML State Machine inclusive OCL	N, indirekt über das Comment Field	N	N
TTCN-3 (grafische Repräsentation (Graphical Format, GFT)	N, indirekt über einen Kommentar	Die Anweisung log ist Teil des Sprachumfangs	?
Automatic Test Markup Language (ATML)	IEEE1641, benötigt „Carrier Language“. In dieser könnte es modelliert sein	?	?
XIL API (ASAM)	N	J	N
CCDL	J	J	J
Flatscript II	?	J	J

3.2.3.4 Heterogenität innerhalb derselben Testskriptsprache

Die *Testskriptsprachen* ermöglichen sowohl die Erweiterung der Sprache für ein spezifisches Testsystem als auch die Erstellung von Komfortfunktionen für die Erstellung von Testfällen. Als Nächstes, werden die Möglichkeiten für die Erweiterbarkeit vorgestellt.

Bei der Anpassung der *Testskriptsprache* werden von Testsystemherstellern Funktionen im Testsystem implementiert und diese über eine API an die *Testskriptsprache* weitergegeben. Die Einbindung von testsystemspezifischen Funktionen erfolgt beispielsweise in CCDL über das Konzept Nutzerfunktionen, in denen in C implementierte Funktionen in CCDL importiert werden können. Andere Sprachen, wie SCXML erlauben die Integration von Skriptelementen oder den Aufruf von externen Prozessen. Das Spektrum an Erweiterungsmöglichkeiten führt dazu, dass Testfälle ohne einen zusätzlichen Integrationsaufwand durch den Testsystemhersteller nicht

zwischen den Testsystemen, welche die gleiche *Testskriptsprache* in der gleichen Version nutzen, ausgetauscht werden können.

Die Erstellung von Komfortfunktionen dient im Testprozess für die Kapselung von häufig auftretende Sequenzblöcken. Hierdurch soll verhindert werden, dass redundante Teile von Testfällen immer wieder durch die Testingenieure implementiert werden müssen. Um dies zu ermöglichen, sehen die *Testskriptsprachen* die Erstellung von Makros, wie in CCDL, Methoden und Klassen in Python oder das Einbetten von Bibliotheken, wie in SCXML, vor. Hierdurch entstehen wiederum Testfälle, welche nicht direkt auf einem anderen Testsystem ausgeführt werden können. In diesem Szenario ermöglicht die Weitergabe der Makros und Bibliotheken die Ausführung der Testfälle auf einem anderen Testsystem ohne zusätzlichen Integrationsaufwand.

Abgesehen von den beiden Arten zur Erweiterung des Funktionsumfangs von *Testskriptsprachen* erlauben alle Sprachen das Einbinden von externen Konfigurationsdateien. So werden beispielsweise produktspezifische Konstanten über diesen Ansatz integriert. Eine Zusammenfassung der Anpassbarkeit einer *Testskriptsprache* für einen Testprozess ist in Tabelle 10 gegeben.

Tabelle 10: Erweiterungen innerhalb einer *Testskriptsprache*, angepasst von [Franke et al. 2018]

ID	Inhalt	Beschreibung
D1	Makros	Gängige Testfälle verwenden Makros, um redundante Testausschnitte zu vereinfachen. Diese Makros werden außerhalb des Testfalls definiert, sind jedoch erforderlich, um seine Semantik aufzulösen.
D2	Benutzerfunktionen und bibliotheksspezifische Funktionen	Benutzerfunktionen und bibliotheksspezifische Funktionen werden verwendet, aber außerhalb des Testfalls definiert. Die Benutzerfunktion könnte auch in einer anderen *Testskriptsprache* definiert werden.
D3	Systemspezifische Funktion testen	Es gibt Funktionen, die direkt auf dem Testsystem implementiert sind und über eine API aufgerufen werden. Diese Funktionen sind nur eine Blackbox für den Testablauf sowie für die verwendete *Testskriptsprache*.
D4	Konstanten	Alle Testverfahrenssprachen beinhalten die Möglichkeit, Konstanten zu verwenden. Dabei sind die Konstanten im Testfall oder in anderen Dateien enthalten.
D5	Abbildung einer Variablen auf das Testsystem	Eine Variable kann für bestimmte IO oder andere Teile des SUT definiert werden. Während der Ausführung müssen die Variablennamen realen Testsystemressourcen zugeordnet werden

3.3 Herausforderungen für den Austausch von Testfällen

Das Unterkapitel fasst die Herausforderungen zusammen, um Anweisungen aus den Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* in Informationen zu transformieren und den Austausch in der Zulieferpyramide. Hierfür werden zuerst die Herausforderung beschrieben, um das Interpretationslevel *semantic understanding* zu erreichen. Anschließend werden Datenintegrationsansätze geprüft, ob sie die Herausforderungen überwinden können.

3.3.1 Interpretation der Testskriptsprachen

Die Transformation von einem Testfall als einen Datenträger in eine Menge von Informationen benötigt das Interpretationslevel *semantic understanding*, welches die niedrigeren Integrationslevel beinhaltet. In der Tabelle 11 ist die Heterogenität der *Testskriptsprachen* bezogen auf die Sprachmerkmale zusammengefasst. Hierbei wird nicht auf den unterschiedlichen Funktionsumfängen der *Testskriptsprachen* eingegangen, weil dies bereits detailliert in 3.2.3 vorgestellt wurde.

Tabelle 11: Heterogenität der *Testskriptsprachen* bezogen auf Sprachmerkmale

ID	Merkmale	Hetero-genität	Beschreibung
1	Lexikalische Merkmale	X	Die vorgestellten *Testskriptsprachen* benutzen unterschiedliche Signalwörter für die Definition der Anweisungen. Dabei werden auch unterschiedliche Signalwörter für ähnliche Anweisungen verwendet. Zum Beispiel wird eine Rampe in CCDL als *Rampe* bezeichnet und in der ASAM XIL API als *Signal*.
2	Syntaktische Merkmale	X	Der Sprachumfang der vorgestellten *Testskriptsprachen* unterscheidet sich voneinander, weshalb auch eine unterschiedliche Menge an Elementen für die Syntax benötigt wird. Außerdem wird für die gleiche Semantik eine unterschiedliche Syntax verwendet. Schließlich werden aufgrund von Unterschieden in der Typsicherheit und im Programmierparadigma unterschiedliche Syntaxelemente benötigt.

3	Morphologische Merkmale	X	Die vorgestellten *Testskriptsprachen* verwenden unterschiedliche Programmierparadigmen. So ist CCDL prozedural und Python objektorientiert. Daraus ergeben sich z.B. unterschiedliche Strukturen für die Deklaration, Initialisierung und den Aufruf von Stimuli.
4	Semantische Merkmale	X	Die vorgestellten *Testskriptsprachen* verwenden eine unterschiedliche Semantik für die Anweisungen. So hat z.B. die Rampe in CCDL und ASAM XIL API unterschiedliche Parameter und bietet damit unterschiedliche Funktionen.

Die Heterogenität der *Testskriptsprachen* ist auf jedem Interpretationslevel vorhanden. **Die Herausforderung ist:** Zwischen den *Testskriptsprachen* muss die Heterogenität für lexikalische, syntaktische, morphologische und semantische Merkmale überwunden werden muss **(H2)**. Dementsprechend muss die Transformation eines Testfalls die Heterogenität für lexikalische, syntaktische, morphologische und semantische Merkmale berücksichtigen. Hierdurch ergeben sich die benötigten Anforderungen für die Transformationsansätze. Im Nachfolgenden wird die Eignung der Transformationsansätze vorgestellt.

3.3.2 Transformationsansätze von Testfällen

Geeignete Transformationsansätze für Testfälle müssen die Heterogenität in den Sprachmerkmalen der betrachteten *Testskriptsprachen* überwinden. Hiermit wäre der Inhalt eines Testfalls als Informationen übersetzbar.

Der Inhalt eines Testfalls ist in einem Dokument gespeichert, wobei das Dateiformat und Schema spezifisch für eine *Testskriptsprache* ist. Beispielsweise sind Testfälle in XIL API in dem Dateiformat XML definiert und Testfälle in CCDL oder Python in dem Format Text. Die Heterogenität in den Dateiformaten und Schemas muss bei der Transformation auch berücksichtigt werden. Hierbei treten Datenintegrationsprobleme auf, welche den Zugriff auf den Inhalt der Testfälle aus verschiedenen *Testskriptsprachen* erschweren.

Die Datenintegrationsprobleme umfassen generell sowohl die Syntax als auch die Semantik, welche als Dateien, Datenbanken oder temporäre Datenstrukturen von Services vorliegen können. An dieser Stelle werden allgemeingültige Datenintegrationsprobleme benannt, welche für alle semi-strukturierten Datenquellen auftreten können. Die in der Abbildung 28 aufgelisteten Datenintegrationskonflikte sind Prob-

leme, die bei einer Datenintegration auftreten können. Der Umfang an Datenintegrationskonflikten wurde von [Goh 1997] definiert und wurden u.a. durch [Wache 2003] bestätigt.

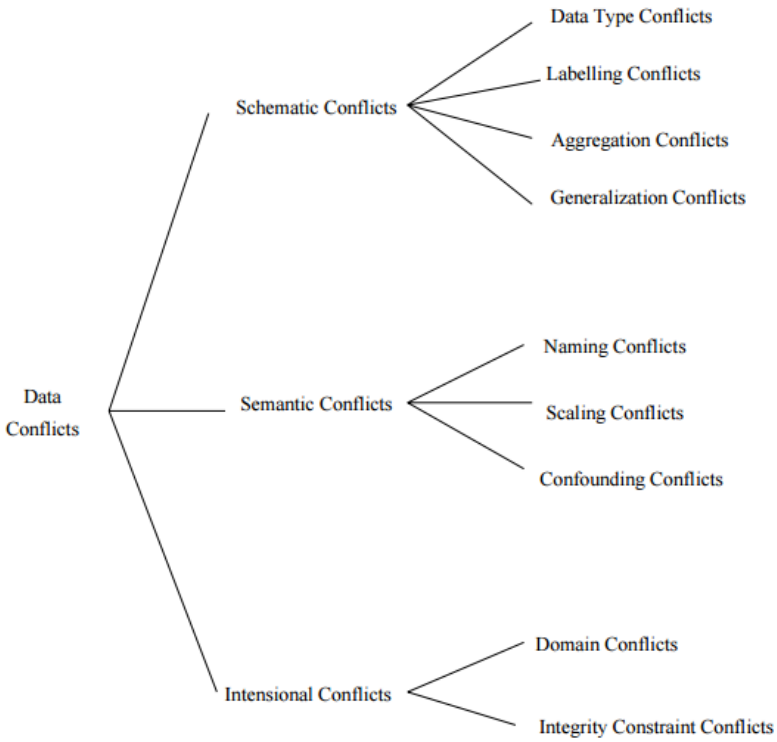

Abbildung 26: Datenintegrationskonflikte nach [Goh 1997]

Die von [Goh 1997] aufgezählten Datenintegrationskonflikte basieren auf der Analyse von Datenbanken. Mit Datenbanken kann eine Untermenge an Strukturen der in Modellen, Beschreibungslogiken und *Testskriptsprachen* enthaltenden Strukturen abgebildet werden. Dementsprechend gelten die Datenintegrationskonflikte von [Goh 1997] auch für die Syntax und Semantik von den betrachteten *Testskriptsprachen*. Es treten zusätzliche Typen von Datenintegrationskonflikten durch die Integration von Taxonomien, Annotation und Axiomen auf. Diese werden hier durch die Sprachmerkmale erfasst und können entsprechend behandelt werden. Die Datenintegrationskonflikte können mithilfe von Datenintegrationsansätzen behandelt werden. Im Folgenden werden die Grundlagen von Datenintegrationsansätzen vorgestellt und anschließend die Anwendbarkeit auf Testfälle beschrieben.

Datenintegrationsansätze

Es gibt eine Vielzahl an Ansätzen für die Datenintegration. Das gemeinsame Ziel dieser Ansätze ist die Bereitstellung einer ganzheitlichen Sicht auf die Daten, welche ihren Ursprung in verschiedenen Datenquellen besitzen. [Lenzerini 2002]. Die

Grundlage dieser einheitlichen Sicht ist ein Informationsmodell, welches den Inhalt der auszutauschenden Informationen beschreibt. Der Inhalt des Modells sind Informationen, welche im Kontext der Arbeit in Testfällen enthalten sind. Die Abbildung der Informationen aus einer Datenquelle auf ein Informationsmodell kann hierbei über die Ansätze *Global as View* (GAV), *Local as View* (LAV) oder Hybriden (GLAV) erfolgen [Lenzerini 2002]. Zu den aufgeführten Ansätzen gibt es eine Vielzahl an technischen Lösungen. Hierbei setzen die GAV-Ansätze der Datenintegration das Informationsmodell als Referenz und definieren hierüber die zu integrierenden Informationen. Entgegengesetzt hierzu setzen die Lösungen der LAV-Ansätze das Schema der lokalen Datenquellen als Referenz und definieren hierüber die zu integrierenden Informationen. Bei allen drei Ansätzen kann die Vollständigkeit der Datenintegration einer Datenquelle bezogen auf das Schema als auch die Tupels (auch Instanzen genannt) variieren. Die Stufen sind in Tabelle 12 gelistet und lehnen sich an [Lenzerini 2002].

Tabelle 12: Vollständigkeitskriterien der Datenintegration

Vollständigkeitskriterien	**Beschreibung**	**Abdeckung des Schemas der Datenquelle**	**Abdeckung des Schemas des Informationsmodells**
Sound Views	Die Informationen sind eine Untermenge der Daten im Informationsmodell	Vollständig	Kann eine Obermenge sein
Complete Views	Die Informationen der Datenquelle sind eine Obermenge der Daten im Informationsmodell	Untermenge	Vollständig
Exact Views	Die Informationen der Datenquelle sind gleich mit den Daten im Informationsmodell	Vollständig	Vollständig

Die Spalte „Abdeckung des Schemas von der Datenquelle“ und „Abdeckung des Schemas vom Informationsmodells“ reflektieren hierbei, welche Elemente der Syntax zwischen den *Testskriptsprachen* und einem Informationsmodell transformiert werden können. Die Stufe *Sound Views* und *Exact Views* sind für die kollaborative Fehlersuche notwendig, um vergleichbare Informationen aus Testfällen unterschiedlicher *Testskriptsprachen* zu erhalten. Diese Stufen können sowohl von LAV als

auch von GAV-Ansätzen erreicht werden. Allerdings orientiert sich die Modellierung der Informationsmodelle in der Praxis an dem GAV-Ansatz, was auch für die austauschbaren Informationen für die kollaborative Suche gilt. Aus diesem Grund werden im Folgenden GAV basierte Datenintegrationslösungen vorgestellt.

Im Rahmen dieser Arbeit wird ein Ansatz vorgestellt, welcher erfolgreich für die Transformation von Daten nach Informationen zwischen semi-strukturierten Dateiformaten funktioniert und die Datenintegrationskonflikte nach [Goh 1997] auflösen kann. Für die Überwindung werden Methoden der Datenintegration benötigt, welche nicht nur auf der syntaktischen Ebene, sondern auch auf der semantischen Ebene eine Integration anstreben. Hierbei befasst sich die Datenintegration mit dem Problem der Kombination von Daten aus unterschiedlichen Quellen und bietet dem Benutzer eine einheitliche Sicht dieser Daten an [Lenzerini 2002]. Im Folgenden werden zwei Lösungen vorgestellt, wobei der Mediator auf die allgemeingültige Datenintegration abzielt und der Quer-Übersetzer gezielt auf Testfälle zugeschnitten ist.

Datenintegration mithilfe von Mediatoren

Die Gruppe der Mediatoren ermöglicht die virtuelle Datenintegration über heterogene Datenquellen. Der grundlegende Aufbau eines Mediators ist in [Franke et al. 2021] [Wiederhold 1992] [Hribernik et al. 2011] beschrieben, kann die aufgeführten Datenintegrationsprobleme handhaben und ist in Abbildung 27 gezeigt.

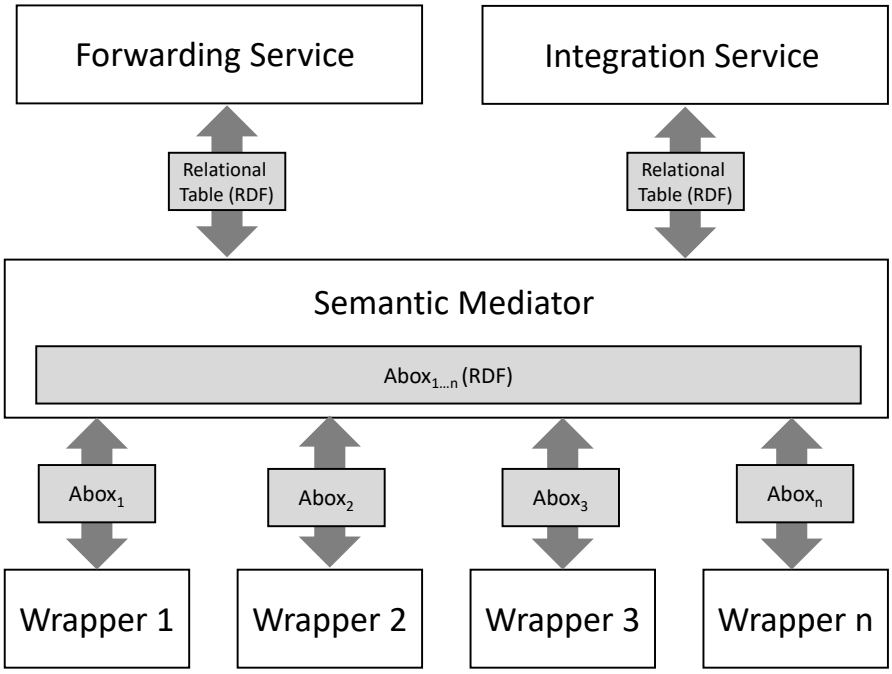

Abbildung 27: Aufbau eines Mediators für die Datenintegration [Franke et al. 2021]

Der Mediator besteht aus zwei Komponenten, nämlich den Wrappern und dem Semantischen Mediator. Der Semantische Mediator verfolgt den GAV-Ansatz. Das hierfür benötigte Informationsmodell kann in verschiedenen Sprachen vorliegen. Dennoch setzen die meisten Ansätze auf Sprachen oder Modelle, welche ausdrucksstärker als die Datenquellen sind. Ontologien haben sich u.a. im Rahmen der Entwicklung zu Semantik Web etabliert [Smart et al. 2010]. Dementsprechend werden für die Formulierung von Suchanfragen SPARQL oder GraphQL [Tubbs 2018] eingesetzt. Die Suchanfrage wird mithilfe der Konzepte und Eigenschaften des Informationsmodells modelliert und an die Wrapper gesendet. Die Aufgabe der Wrapper ist es, die Daten der Datenquelle in lokale Ontologien umzuwandeln und dem Semantic Mediator zur Verfügung zu stellen. Dazu werden für die Transformation prozedurale Regeln definiert und angewendet. Das Ziel einer prozeduralen Regel ist die direkte Transformation eines Elements eines Datenschemas in ein Konzept oder Eigenschaft einer Ontologie. Anschließend aggregiert der semantische Mediator die lokalen Ontologien zu einem Gesamtergebnis. Die Tragfähigkeit des Ansatzes für die semantische Datenintegration wurde für verschiedene heterogene Datenquellen evaluiert. Hierbei hat der Semantische Mediator die Überwindung der Integrationskonflikte für semi-strukturierte Datenquellen entwickelt [Hribernik et al. 2011; Franke et al. 2013] und seine Anwendbarkeit in einem großen Spektrum von Anwendungsgebieten von der Logistik bis hin zu Virtual Factories gezeigt [Franke et al. 2016] [Shani et al. 2017].

Die Datenintegration der Testfälle kann mit Hilfe eines semantischen Mediators erfolgen. Dazu muss für jede *Testskriptsprache* ein entsprechender Wrapper implementiert werden. Anschließend muss eine Menge von SPARQL-Anfragen definiert werden, die den Inhalt des Testfalls beschreiben. Dadurch können Informationen, die sich an einer Stelle des Testfalls befinden, mit Hilfe einer SPARQL-Abfrage übersetzt werden. Während dieser Ansatz der Datenintegration für prozedurale Sprachen wie CCDL erfolgreich möglich ist, ist dieser Ansatz für objektorientierte Sprachen problematisch. Der Grund hierfür ist, dass eine Information an mehreren Stellen im Testfall kodiert ist und damit nicht mit Hilfe einer SPARQL-Abfrage extrahiert, werden kann. Beispielsweise werden globale Variablen an verschiedenen Stellen deklariert, initialisiert und verwendet. Des Weiteren führen Methoden und Klassen zu einer weiteren Fragmentierung der Information. Dies führt dazu, dass für eine prozedurale Regel nur eine Position bei der Datenintegration verwendet werden kann und die anderen Positionen ignoriert werden. Das Ergebnis einer solchen Transformation ist, dass nur Fragmente der Information extrahiert werden kann und hierdurch ein Informationsverlust entsteht. Im Kontext von Testfällen haben sich andere Technologien durchgesetzt, die primär auf eine Transkompilierung abzielen.

Datenintegration mithilfe von Transcompilern

Die direkte Transformation eines Testfalls von einer *Testskriptsprache* in eine andere *Testskriptsprache* oder Informationsmodell ist möglich. Diese Art von Tool

wird als Transcompiler (Source-to-Source-Compiler) bezeichnet und dient hauptsächlich der Interoperabilität von Quellcode. Es gibt zwei Ansätze, nämlich die regelbasierten Ansätze und die Anwendung von KI-basierten Ansätzen. Beide Ansätze wurden betrachtet.

Das Ergebnis ist, dass es bereits für Quellcode erfolgreiche Übersetzungstools gibt, welche Testfälle mithilfe der Transformation übersetzen. Ein Beispiel ist das Werkzeug von Moses [Koehn et al. 2007] . Es wurde angewendet, um von Java nach C# oder von Python 2 nach Python 3 zu übersetzen [Lachaux et al. 2020]. Das Ergebnis ist, dass die Übersetzungsergebnisse ausführbar sind, aber nicht für menschliche Leser lesbar sind. Dadurch erfordert es eine manuelle Nachbesserung, um richtig zu funktionieren [Lachaux et al. 2020]. Die Übersetzungsergebnisse müssen für Testingenieure im Kontext der kollaborativen Fehlersuche lesbar und verständlich sein, was diesen Ansatz für diese Dissertation ungeeignet macht. Lachaux schlägt ein unüberwachtes Übersetzungsverfahren vor, das eine Sequenz-zu-Sequenz-Methode (seq2seq) verwendet, um ähnliche Funktionalität zwischen Sprachen mit einem einzigen Modell abzubilden. Der Ansatz funktioniert bei der Übersetzung von Quellcode zwischen Programmiersprachen, die dasselbe Programmierparadigma teilen, und kann mit riesigen Datenmengen (2,8 Millionen Open-Source-GitHub-Repositorien) trainiert werden. Der Austausch von Testfällen innerhalb der Zulieferpyramide benötigt eine Lösung, welche heterogene *Testskriptsprachen* unterstützt. Hierbei ist es notwendig, dass *Testskriptsprachen* mit unterschiedlichen Programmierparadigmen unterstützt werden. Die Übersetzung zwischen verschiedenen *Testskriptsprachen* mit unterschiedlichen Programmierparadigmen bedeutet, dass ein einstufiger Übersetzungsmechanismus nicht funktionieren kann. Der Grund dafür ist derselbe wie beim semantischen Mediator. Darüber hinaus enthalten die Trainingsdaten in der Anwendungsdomäne Avionik nur wenige Testfälle für eine bestimmte *Testskriptsprache*, sodass das Training von Methoden der Künstlichen Intelligenz unrealistisch ist.

3.4 Zusammenfassung

Die primäre Informationsquelle für die kollaborative Fehlersuche sind Testfälle, woraus sich folgende Herausforderungen für deren Nutzung ergeben.

Die Testfälle liegen in verschiedenen *Testskriptsprachen* vor, die sich sowohl in den Sprachmerkmalen, Sprachumfang als auch in den Dateiformaten und Schemata unterscheiden. Somit sind die eingesetzten *Testskriptsprachen* heterogen und benötigen eine Datenintegration.

Es wird ein Datenintegrationsansatz benötigt, der die Daten aus den Testfällen extrahieren und als Information zur Verfügung stellen kann. Dazu werden Datenintegrationsmethoden benötigt, die eine Integration nicht nur auf syntaktischer, sondern

auch auf semantischer Ebene anstreben. Um dieses Verständnis zu erreichen, muss die Ebene des *semantic understanding* erreicht werden. Das *semantic understanding* ermöglicht dabei das entscheidende Verständnis über die Inhalte der Testfälle und ermöglicht somit die Überbrückung der Lücke von einer Datensicht zu einer Informationssicht. Hierzu gibt es verschiedene Ansätze zur Datenintegration. Untersucht wurde der generische Ansatz mit Hilfe von Mediatoren oder eine quellcodespezifische Lösung aus dem Umfeld der Transcompiler.

Die Datenintegration von Testfällen kann mit Hilfe eines semantischen Mediators erfolgen. Dazu müsste für jede *Testskriptsprache* ein entsprechender Wrapper geschrieben werden. Der regelbasierte Ansatz führt bei objektorientierten *Testskriptsprachen* zu einem Informationsverlust und ist daher ungeeignet. Im Kontext von Testfällen haben sich andere Technologien durchgesetzt, die primär auf eine Transkompilierung abzielen. Die direkte Transformation eines Testfalls aus einer *Testskriptsprache* in ein Informationsmodell ist möglich. Diese Art von Werkzeugen wird als Transcompiler (Source-to-Source-Compiler) bezeichnet und dient in erster Linie der Austauschbarkeit von Quellcode. Es gibt zwei Ansätze: regelbasierte Ansätze und KI-basierte Ansätze. Beide Ansätze liefern entweder Quellcode, der nicht lesbar ist, oder müssen auf großen Datenmengen trainiert werden, was im Einsatzszenario der Dissertation nicht möglich ist.

Die kollaborative Fehlersuche benötigt einen neuartigen Datenintegrationsansatz, welcher auf die Extraktion von Informationen und nicht auf die Transformation von Syntaxelementen abzielt. Die notwendigen Informationen werden durch die Anweisungstypen *Stimulus, Laufzeitverhalten* und *Verdikt* definiert, worauf eine Fehlersuche aufbauen muss.

4 Konzept für die kollaborative Fehlersuche

In den vorangegangenen Kapiteln wurde die Notwendigkeit einer Fehlersuche als eine testprozessübergreifende Aktivität in der Zulieferpyramide identifiziert. Darüber hinaus wurden die Herausforderungen für den Austausch von testrelevanten Informationen aus Testfällen erarbeitet. Kapitel 4 beschreibt eine kollaborative Fehlersuche, welche auf den Austausch und der Anwendung von testrelevanten Informationen zugeschnitten ist.

Das Unterkapitel 4.1 führt die kollaborative Fehlersuche ein. Hierbei werden Anforderungen an die Fehlersuche hergeleitet und mit einem **(R)** markiert**.** Das Unterkapitel 4.2 spezifiziert das Informationsmodell, welches für den Austausch von testrelevanten Informationen notwendig ist. Hierfür werden Ontologien, Testmodelle und Graphen als mögliche Kandidaten für die Repräsentation der testrelevanten Informationen evaluiert. Das Unterkapitel 4.3 führt abschließend die drei Fehlersuchmethoden der kollaborativen Fehlersuche ein.

4.1 Einführung in die kollaborative Fehlersuche

Die Entwicklung komplexer mechatronischer Produkte erfolgt innerhalb einer Zulieferpyramide und beinhaltet *lose gekoppelte* Testprozesse. Beim Testen der unterschiedlichen Komponenten, Module und des Gesamtsystems kann der Fall eintreten, dass die Testprozesse zu unterschiedlichen Ergebnissen kommen, obwohl sie ähnliche Funktionen gegen ähnliche Anforderungen testen. In einem solchen Fall müssen die Gründe dafür im Rahmen einer testprozessübergreifenden Aktivität gefunden werden. Die Teilnehmer der testprozessübergreifenden Aktivität wären Testingenieure aus den *lose gekoppelten* Testprozessen. Als minimalen Input stehen hierfür der fehlgeschlagene Testfall des Integrators (zum Beispiel der OEM), der vergleichbare Testfall des Lieferanten (Wenn der Integrator ein OEM ist, dann der Modul-Lieferant) und die Testberichte der Testausführungen zur Verfügung. Diese Inputs werden verwendet, um eine kollaborative Fehlersuche durchzuführen. Die Herausforderung bei der Verwendung der Inputs ist, dass die Testfälle in unterschiedlichen *Testskriptsprachen* implementiert sind und nicht für alle Testingneiure verständlich und entsprechend verwertbar sind. Hierdurch können die Informationen in den Testfällen bezüglich der Stimulierung des SUT oder der Überwachung der SUT Reaktion nicht zwischen den Testingeniuren ausgetauscht werden, um mögliche Unterschiede und Fehlerquellen zu erkennen.

Ziel des Ansatzes ist der Austausch von Testfallinhalten als Informationen zwischen gekoppelten Testprozessen. Dazu wird der Ansatz der Interoperabilität von Testfällen in die kollaborative Fehlersuche integriert. IEEE definiert Interoperabilität als die Fähigkeit das mehrere Systeme sowohl Informationen austauschen als die

Informationen auch nutzen können. [Geraci et al. 1991]. Eine Information sind Daten, welche ihre Bedeutung durch die Beziehung zu einem Objekt erhält [Bellinger et al. 2004]. Hierbei ist die Menge an relevanten Informationen durch die Anweisungstypen der *Testskriptsprachen* definiert. Die Anweisungen sind in den Testfällen für die Anweisungstypen (*Stimulus, Laufzeitverhalten* und *Verdikt)* interoperabel austauschbar **(R1**). Die kollaborative Fehlersuche soll den Austausch der Informationen zwischen Stakeholdern in der Zulieferpyramide nicht nur ermöglichen, sondern daraus einen Mehrwert generieren. Der Austausch der Informationen ist nicht hinreichend, sondern erst die Nutzung für die Fehlersuche **(R2)**. Dieser Aspekt der Interoperabilität wird durch die Kommunikation mit anderen Systemen und die Möglichkeit der Nutzung von Funktionen auf anderen Systemen verstärkt [Vernadat 1996]. Im Kontext der kollaborativen Fehlersuche sind die anderen Systeme die unterschiedlichen Testautomatisierungswerkzeuge und die Testsysteme der *lose gekoppelten* Testprozesse. Die in diesem Zusammenhang relevanten interoperablen Funktionen ergeben sich aus der Testvorbereitung und der Testausführung aus dem Testprozessschritt *Realisierung und Durchführung*.

Die Integration der kollaborativen Fehlersuche in den Entwicklungsprozess ist in der Abbildung 28 als ein Sequenzdiagramm dargestellt. Die Anwendung der kollaborativen Fehlersuche wird notwendig, sobald zwei Testprozesse zu widersprüchlichen Ergebnissen gelangen. Dieser Fall kann überall in der Zulieferpyramide auftreten und wird exemplarisch für den Testprozess des Gesamtsystems (OEM) und für den Testprozess des Modul-Lieferanten (Tier-1) beschrieben.

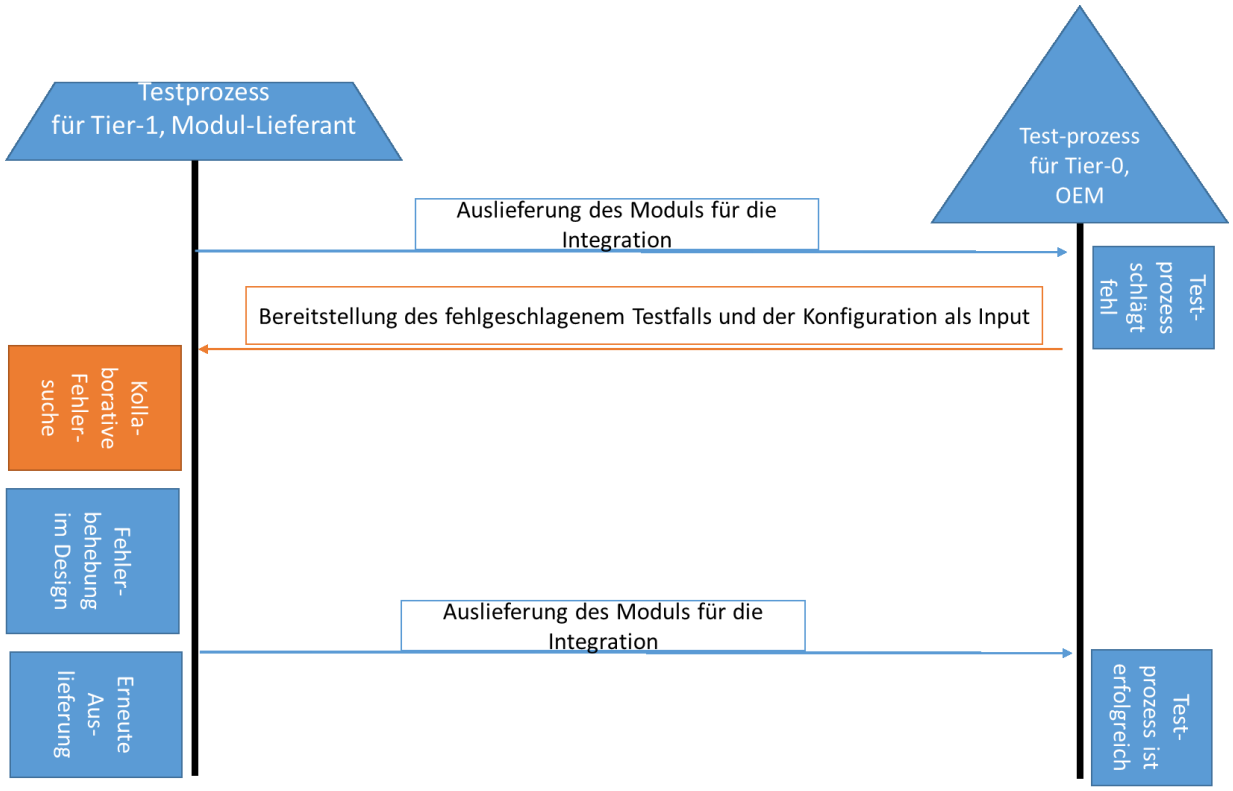

Abbildung 28: Integration der kollaborativen Fehlersuche in die Testprozesse

In dem gezeigten Beispiel entwickelt ein Modul-Lieferant auf Tier-1 ein Modul. Der Testprozess vom Lieferanten bescheinigt dem System die Funktionstüchtigkeit und es wird zum OEM ausgeliefert. Dort wird das Modul integriert und das Gesamtsystem vom OEM als Produkt auf Systemlevel getestet. In dem betrachteten Fall schlägt der Testprozess fehl und die Fehlersuche beginnt. Hierbei wird die Fehlersuche zuerst beim OEM durch die Überprüfung des Testfalls und der Konfiguration ausgeführt. Sollte der Fehler sich nicht beheben lassen, wird die Fehlersuche auch auf den Lieferanten ausgedehnt und die kollaborative Fehlersuche wird initiiert. Dieser Fall tritt ein, wenn der Testprozess des OEM den Fehler nicht in der Integration der Module als Gesamtsystem gefunden hat und den Fehler bei seinem Lieferanten vermutet. Nach erfolgreicher kollaborativer Fehlersuche und Fehlerbeseitigung seitens des Lieferanten wird das System erneut getestet. Hierfür wird zuerst der Testprozess beim Lieferanten erfolgreich durchlaufen und das Modul freigegeben. Anschließend wird es dem OEM für die erneute Integration und dem Systemtest bereitgestellt. Der beschriebene Ablauf ist eine Variation der möglichen Abläufe in dem Entwicklungsprozess und soll ausschließlich die Integration der kollaborativen Fehlersuche in den Entwicklungsprozess exemplarisch beschreiben.

Die kollaborative Fehlersuche soll die Interoperabilität von Testfällen ermöglichen und soll darauf aufbauend drei semi-automatische Fehlersuchmethoden (*Ausführung*

des Testfalls und Analyse des Signalverlaufs, *Mustersuche* und *Anpassung der Konfiguration und anschließende Ausführung*) implementieren **(R3)**. Das Ziel ist die Identifizierung der Fehlerursache für den fehlgeschlagenen Testprozess. Die Eingabe für die Fehlersuchmethoden sind die Testfälle und die Konfiguration der höheren Integrationsstufe **(R4)** (wie der Testfall vom OEM im gezeigten Beispiel). Die Anwendung der Fehlersuche erfolgt in drei Schritten, welche in Abbildung 29 abgebildet sind und im Folgenden im Detail beschrieben werden.

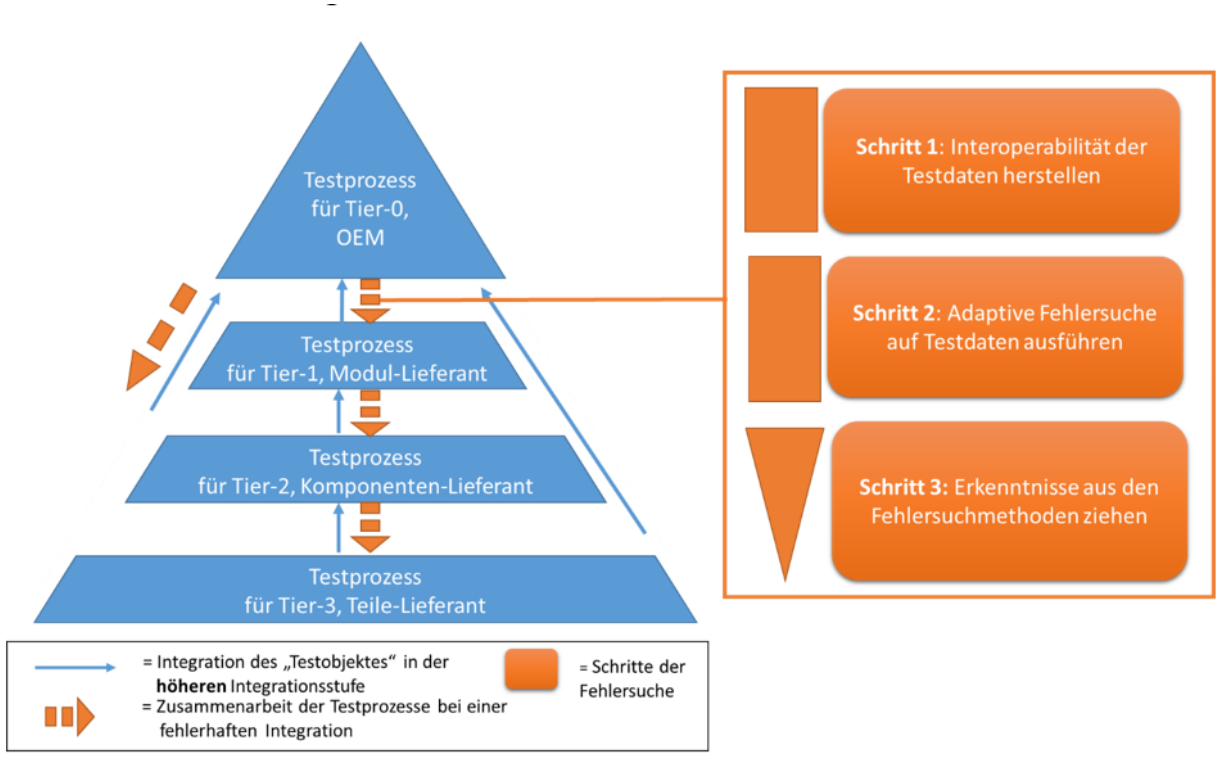

Abbildung 29: Schritte der kollaborativen Fehlersuche

Schritt 1: Interoperabilität der Testdaten herstellen

Die Testdaten, bestehend aus einem Testfall und einer Konfiguration, müssen so aufbereitet werden, dass sie für den Zulieferer interoperabel und damit direkt nutzbar sind. Hierfür stellt die Fehlersuche Methoden zur interoperablen Darstellung und Übersetzung mithilfe der Transformation von Testfällen zur Verfügung. Für die interoperable Darstellung von Konfigurationen werden verfügbare Werkzeuge genutzt.

Die interoperablen Versionen des Testfalls vom OEM und der Konfiguration vom OEM werden generiert und bereitgestellt. Es ist nicht garantiert, dass der fehlgeschlagene Testfall in eine interoperable Version übersetzt werden kann. Die verwendeten Funktionen der *Testskriptsprache* bestimmen, ob die Interoperabilität möglich ist. Die Detaillierung der erreichbaren Interoperabilitätsstufen des fehlgeschlagenen

Testfalls und deren Auswirkungen auf den Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen ist in der Tabelle 14 zusammengefasst.

Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen
Die interoperable Version des Testfalls und der Konfiguration sind der zentrale Input. Das Konzept sieht die Anwendung von drei Fehlersuchmethoden vor, die für unterschiedliche Ausgangssituationen geeignet sind und sich adaptiv an die Heterogenität der Testfälle und der Konfiguration anpassen können. Alle Fehlersuchmethoden versuchen, Unterschiede zwischen dem fehlgeschlagenen Testfall des Testprozesses für das Gesamtsystem (OEM) und dem vergleichbaren Testfall des Testprozesses für das System/Modul (Lieferant Tier-1) zu finden. Vergleichbar bedeutet hier, dass die Testfälle trotz ihrer Darstellung in unterschiedlichen *Testskriptsprachen* einen ähnlichen Sachverhalt beschreiben. Die Identifizierung aller Unterschiede zwischen zwei Testprozessen benötigt sowohl dynamische als auch statische Analysen **(R5)**.

Ähnliche Testfälle sind zwischen den Testprozessen gewährleistet, da ähnliche Anforderungen getestet werden. Der Zusammenhang zwischen Stakeholder- und System- bzw. Systemelementanforderungen und deren Einfluss auf die Ähnlichkeit von Testfällen wurde in Kapitel 2 diskutiert. Im Folgenden wird kurz auf die verschiedenen Fehlersuchmethoden aus dem Schritt 2 eingegangen. Dabei wird für jede Fehlersuchmethode die Vorgehensweise und deren Voraussetzungen dargestellt. Die ausführliche Darstellung erfolgt dann in Kapitel 4.3.

Drei Fehlersuchmethoden sind erforderlich, um alle möglichen Unterschiede zwischen zwei Testdurchführungen zu identifizieren. Die erste Gruppe von Unterschieden beschreibt Unterschiede im Laufzeitverhalten des Testfalls. Die zweite Gruppe adressiert strukturelle Unterschiede im Testfall und die dritte Gruppe beschreibt strukturelle Unterschiede in der Konfiguration.

Die erste Fehlersuchmethode ermöglicht die Identifizierung von Unterschieden in den Signalverläufen. Diese Methode kann nur angewendet werden, wenn die Voraussetzungen für die Ausführung des Testfalls gegeben sind. Die Fehlersuchmethode 1 kann nicht alle möglichen Unterschiede in den Testfällen und der Konfiguration erkennen. So kann z.B. der Einfluss der Stimulation, die ausschließlich durch den Testfall erzeugt wird, in den Signalverläufen nicht eindeutig erkannt werden. Ebenso kann ein Fehler in der Konfiguration nicht anhand der Signalverläufe erkannt werden. Daher sind die beiden anderen Fehlersuchmethoden erforderlich. Die zweite Fehlersuchmethode ermöglicht eine statische Analyse der Testfälle, wodurch Unterschiede in der Semantik der Testfälle gefunden werden können. Die dritte Fehlersuchmethode ermöglicht es zu identifizieren, ob der Fehler in der Konfiguration versteckt ist.

Hierfür gilt als weitere Voraussetzung: Testfälle der niedrigeren Integrationsstufe (Modul- oder Komponenten-Lieferant) werden als interoperable Eingabe benötigt **(R6)**. Die Aufschlüsselung der Anwendbarkeit der jeweiligen Fehlersuchmethoden und ihre Ergebnisse werden im Kapitel 4.3 detailliert beschrieben. Abbildung 30 definiert mit welchen Fehlersuchmethoden welche Unterschiede gefunden werden können. Hierfür wurden die Unterschiede in drei Gruppen eingeteilt, welche in Tabelle 17 aus dem Unterkapitel 4.3 detailliert werden.

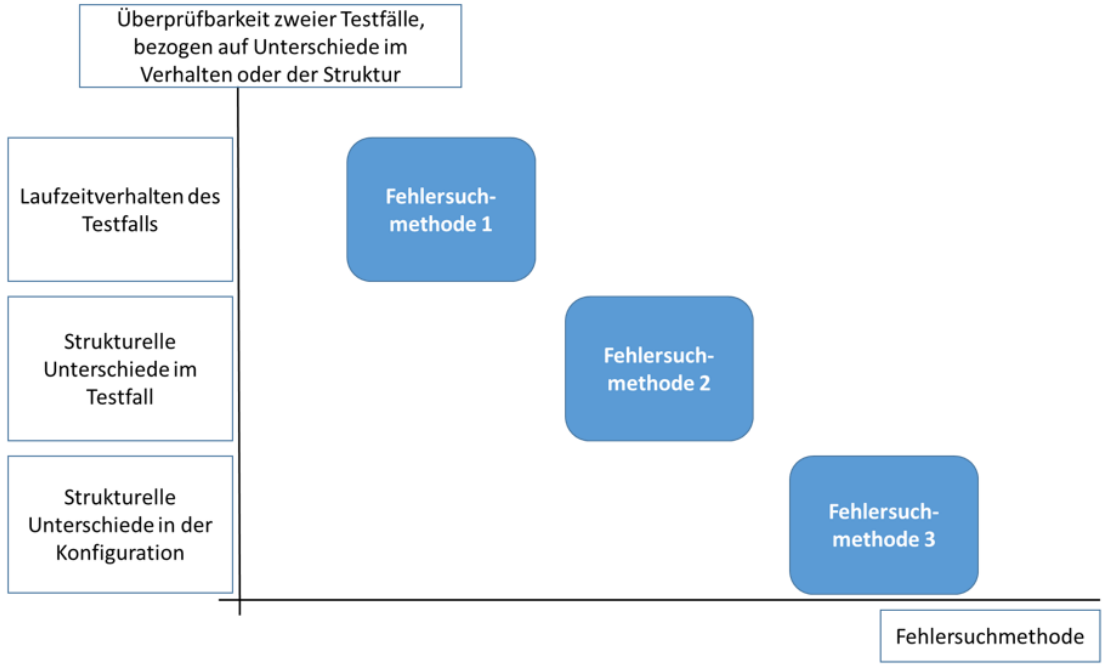

Abbildung 30: Identifikation von Unterschieden durch die Fehlersuchmethoden

Das Identifizieren der Unterschiede zwischen den Testfällen ist das Ergebnis der Fehlersuchmethoden. Es wird als Eingabe an den dritten Schritt der Fehlersuche weitergereicht.

Schritt 3: Erkenntnisse aus den Fehlersuchmethoden ziehen

Das Ziel des dritten Schrittes ist die Analyse der Fehlerursache basierend auf den gefundenen Unterschieden. Mit den identifizierten Unterschieden kann die Fehlerursache mit den Methoden aus dem Stand der Technik auf einzelne Signale zurückgeführt werden. Der erfolgreiche Testfall und die Fehlerursache können im Rahmen des Erkenntnisgewinn genutzt werden, um den fehlgeschlagenen Testfall anzupassen oder das Modul des Modul-Lieferanten entsprechend der Erwartungen neu zu entwickeln.

Die folgende Detaillierung der drei Fehlersuchmethoden basiert auf zwei Herausforderungen der Problemstellung. Eine Übersicht über die zu lösenden Herausforderungen der Dissertation und die dafür zu erfüllenden Anforderungen sind in der Tabelle 13 gegeben.

Tabelle 13: Herausforderungen und abgeleitete Anforderungen der Fehlersuche

Herausforderung	**Kapitel**	**Abgeleitete Anforderungen im Konzept**
Die aufgeführten Fehlerquellen können nur in den Testfällen erkannt werden, wenn die beiden Testfälle von den *lose gekoppelten* Testprozessen miteinander verglichen werden können **(H1)**.	2.5	• Der Austausch der Informationen ist nicht hinreichend, sondern erst die Nutzung für die Fehlersuche **(R2)**. • Die kollaborative Fehlersuche soll die Interoperabilität von Testfällen ermöglichen und soll darauf aufbauend drei semi-automatische Fehlersuchmethoden (Ausführung des Testfalls und Analyse des Signalverlaufs, Mustersuche und Anpassung der Konfiguration und anschließende Ausführung) implementieren **(R3)**. • Die Identifizierung aller Unterschiede zwischen zwei Testprozessen benötigt sowohl dynamische als auch statische Analysen **(R5).**
Zwischen den *Testskriptsprachen* muss die Heterogenität für lexikalische, syntaktische, morphologische und semantische Merkmale überwunden werden muss **(H2).**	3.3.1	• Die Anweisungen sind in den Testfällen für die Anweisungstypen (*Stimulus, Laufzeitverhalten* und *Verdikt)* interoperabel austauschbar **(R1)**. • Die Eingabe für die Fehlersuchmethoden sind die Testfälle und die Konfiguration der höheren Integrationsstufe **(R4)**. • Testfälle der niedrigeren Integrationsstufe (Modul- oder Komponenten-lieferant) werden als interoperable Eingabe benötigt **(R6)**.

Im Folgenden werden die Interoperabilitätsstufen für den Austausch der Testfälle und der Konfiguration des OEM detailliert dargestellt. Dabei erfolgt die Abbildung der Interoperabilitätsstufen auf die Ausführbarkeit der Fehlersuchmethoden.

Die Fehlersuchmethoden fokussieren auf die Identifikation von Unterschieden aus den in Abbildung 30 eingeführten Gruppen. Die Einteilung der Gruppen und die Definition der Unterschiede werden im Kapitel 4.2 detailliert dargestellt. Abschließend werden im Kapitel 4.3 die drei Fehlersuchmethoden hinsichtlich ihrer Anwendbarkeit, ihres Ansatzes und ihrer Vorgehensweise detailliert beschrieben.

Interoperabilitätsstufen für die Eingabe der Fehlersuchmethoden

Der **Schritt 1: Interoperabilität der Testdaten herstellen** der kollaborativen Fehlersuche sieht die interoperable Bereitstellung von Testfällen und Konfigurationen vor. Die ist nötig, weil die Interoperabilität durch die Heterogenität der *Testskriptsprachen* und Testsysteme in der Zulieferpyramide nicht gegeben ist. Der Ansatz der kollaborativen Fehlersuche transformiert die Testfälle in Informationen, welche in einem Informationsmodell gespeichert werden. Anschließend können die Informationen im Rahmen der Fehlersuchmethoden des zweiten Schrittes in die Zielformate transformiert werden.

Der Erfolg der Transformation eines Testfalls in das Informationsmodell und die nachgelagerte Transformation in das Zielformat beeinflussen die Anwendbarkeit der Fehlersuchmethoden im **Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen**. Die Tabelle 14 zeigt den Erfolg der Transformation der Testfälle und Konfigurationen und die anwendbare Fehlersuchmethode. Hierbei bedeutet die partielle Transformation eines Testfalls, dass die Fehlersuchmethode nur partielle Ergebnisse liefern kann.

Tabelle 14: Interoperabilitätsstufen für die Anwendung der Fehlersuchmethoden

Erfolg der Transformation des Testfalls und der Konfiguration	**Testfall des OEM**	**Konfiguration des OEM**	**Testfall des OEM kann beim Lieferanten ausgeführt werden**	**Anwendbare Fehlersuchmethode**
vollständig	Fehlgeschlagener Testfall kann vollständig in die Zielsprache übersetzt werden	Konfigurationen sind verfügbar und können in das Zielformat übersetzt werden	Ja	Methode 1
Unvollständig bezüglich der Konfiguration	Fehlgeschlagener Testfall kann vollständig in die Zielsprache übersetzt werden	Konfigurationen können nicht oder nur teilweise genutzt werden	Nein	Methode 2 Methode 3
Unvollständig bezüglich des Testfalls: Nur der Ablauf ist austauschbar	Fehlgeschlagener Testfall kann die testsystemspezifischen Funktionen nicht in die Zielsprache übersetzen	Konfiguration ist nicht relevant	Nein	Methode 2
Unvollständig bezüglich des Testfalls: Funktionssicht ist nur teilweise möglich	Fehlgeschlagener Testfall enthält Funktionen für den Testablauf, welche nicht in die Zielsprache verfügbar sind	Konfiguration ist nicht relevant	Nein	Methode 2
Unvollständig, Testfall und Konfiguration können nicht transformiert werden	Fehlgeschlagener Testfall und Konfiguration können nicht die Zielsprache überführt werden.	Konfiguration ist nicht relevant	Nein	Methode 3

4.2 Informationsmodell für die Interoperabilität von Testfällen

Die in Kapitel 4.1 vorgestellten Fehlersuchmethoden 1 und 2 benötigen als Eingabe Informationen aus dem fehlgeschlagenen Testfall, die in einem Informationsmodell

abgelegt sind. Ein Informationsmodell ist hierbei ein relationales Datenmodell, welches u.a. sämtliche Entitäts- und Attributnamen exakt ausdrückt und hierfür eine Definition bereitstellt [Berner 2016, S. 17]. Des Weiteren werden alle Entitäten mit Beziehungen verknüpft und alle verwendeten Begriffe sind in der Domäne verständlich [Berner 2016, S. 17].

Im Rahmen dieser Arbeit und in Anlehnung an [Berner 2016] ist ein *Informationsmodell* ein Modell zur abstrakten Beschreibung einer Domäne, mit dem die Interoperabilität von Testfällen erreicht werden soll. Dazu werden Konzepte, Datentypen und Regeln über den Inhalt von Testfällen definiert. Das Informationsmodell kann dabei in verschiedenen Dateiformaten vorliegen. Durch die Darstellung in einem Dateiformat können die Informationen persistiert und ausgetauscht werden. Im Folgenden werden die Inhalte des Informationsmodells definiert. Anschließend wird auf Basis des Standes der Technik ein geeignetes Dateiformat ausgewählt.

Die relevanten Informationen ergeben sich aus der Struktur typischer Testfälle, die bereits in Kapitel 3.2.1 in der Abbildung 21 beschrieben wurde. Daraus ergibt sich, dass es in Testfällen drei verschiedene Schritte gibt, deren Anweisungen in die Anweisungstypen *Stimulus, Verdikt* und *Laufzeitverhalten* eingeteilt werden können. Im Folgenden werden diese Typen als zu identifizierende Informationsgruppen kurz vorgestellt.

- In die Gruppe *Stimulus* werden alle Anweisungen zugewiesen, welche den Wert eines Signals aktiv ändern. Hierzu zählen beispielsweise Anweisungen, die den Wert des Signals einmalig ändern oder auch Anweisungen, welche den Wert kontinuierlich anpassen. Diese Informationsgruppe beschreibt die Stimulierung des SUT zu einem Zielzustand, um den Test durchführen zu können.
- In der Gruppe *Verdikt* werden alle Anweisungen zugewiesen, welche das SUT auf eine Bedingung überprüfen und entsprechend das Ergebnis protokollieren können. Hierzu zählen beispielsweise einmalige Überprüfungen als auch Überprüfungen, welche sich über eine Zeitspanne erstrecken.
- In der Gruppe *Laufzeitverhalten* werden alle Anweisungen zugewiesen, welche das Laufzeitverhalten aktiv steuern. Hierzu zählen beispielsweise Anweisungen, welche die Testausführung um eine definierte Zeitspanne pausieren lässt oder die maximal erlaubte Ausführungszeit von Blöcken definiert.

Jede der genannten Informationsgruppen beinhaltet Informationen vom selben Anweisungstyp (siehe Unterkapitel 3.2). Die Beschreibung der jeweiligen Anweisungen benötigt eine allgemein gültige Struktur, um Unterschiede identifizieren zu kön-

nen. Die Menge an zu unterstützenden Anweisungen wurde bereits in dem Forschungsprojekt AGILE-VT [dSPACE GmbH 2021] herausgearbeitet und ist in der folgenden Tabelle 15 den Informationsgruppen zugewiesen.

Tabelle 15: Abbildung des TASCXML-Befehlssatzes auf Informationsgruppen

ID	TASCXML Befehlssatz	Beschreibung	Gruppe
1	<tascxml:set>	Setzt ein Signal auf einen bestimmten Wert	*Stimulus*
2	<tascxml:get>	Liest den aktuellen Wert eines Signals	Es lässt sich in keine Gruppe einordnen
3	<tascxml:result>	Protokolliert ein Ergebnis (Urteil) des Testfalls	*Verdikt*
4	<tascxml:ramp>	Löst eine Rampe aus, die ein Signal für eine bestimmte Dauer ändert	*Stimulus*
5	<tascxml:sine>	Erzeugt eine Sinuskurve	*Stimulus*
6	<tascxml:sawtooth>	Erzeugt eine Wellenform eines Sägezahns	*Stimulus*
7	<tascxml:pulse>	Erzeugt eine Wellenform eines Pulses	*Stimulus*
8	<tascxml:verifytolerance>	Überprüft, ob der Wert eines Signals innerhalb eines bestimmten Toleranzbereichs liegt,	*Verdikt*

Die aufgeführten Anweisungen beinhalten keine Anweisungen für den Anweisungstyp *Laufzeitverhalten*. Diese Anweisungen sind aber zwingend erforderlich, um ein System zu stimulieren und auf die Reaktion des SUT zu warten. Entsprechend der Definition eines minimalen Testfalls (siehe Abbildung 18) werden Anweisungen für das Pausieren eines Testfalls benötigt. Dabei werden die Varianten für das Pausieren der Testausführung und für das Warten auf die Erfüllung einer Bedingung benötigt.

Abgesehen von den benötigten Anweisungstypen beinhalten Testfälle auch immer Metainformationen. Zu diesen gehören beispielsweise der Testschritt, Abdeckungskriterien und zu berücksichtigende Timeouts. Die Menge an benötigten Informationen hängen von dem jeweiligen Testprozess ab. Dementsprechend werden diese Informationen als Metainformationen modelliert und optional über Parameter integriert.

4.2.1 Kandidaten für die Informationsmodelle

Der Inhalt der Testfälle kann mithilfe unterschiedlicher Modelle repräsentiert werden. Hierbei wurden Ontologien als Kandidat ausgewählt, weil sie die Abbildung von beliebigen Inhalten als ontologische Konzepte ermöglichen. Zusätzlich spricht für Ontologien, dass die Syntax und Semantik der *Testskriptsprachen* in einer Ontologie abbildbar sind.

Generell können alle Modelle berücksichtigt werden, mit denen Zustandsübergangsdiagramme repräsentiert werden können. Dementsprechend wurden Testmodelle ausgewählt, weil sie die Abstrahierung von Testfällen anbieten. Abschließend wurden auch Graphen ausgewählt, welche die Fähigkeit der Abbildung von beliebigen Inhalten als Graph besitzen. Die folgenden Unterkapitel sind aus der eigenen Veröffentlichung [Franke und Thoben 2022] übernommen.

4.2.1.1 Ontologien

Ontologien werden häufig verwendet, um Informationen unabhängig von einem bestimmten Datenformat und Tool zu modellieren. Wie in der Künstlichen Intelligenz (KI) diskutiert, sind Ontologien formale, partielle Spezifikationen einer Vereinbarung über die Beschreibung einer Domäne [Guarino 1998]. Eine Ontologie besteht aus Konzepten und Relationen und deren Definitionen, Eigenschaften und Beschränkungen, die durch Axiome ausgedrückt werden [Uschold und Gruninger 1996]. Ontologien wurden als Teil des Semantic Web angewendet, um unzählige Domänen zu beschreiben. Dabei sind RDF/XML- und OWL-Ontologien die Dateiformate zur Formalisierung von Domänenwissen im Semantic Web. Dabei existieren Ontologien auf verschiedenen Ebenen, jede mit einem bestimmten Ziel. Es wird angestrebt, grundlegende Konzepte und Eigenschaften mit sogenannten Foundation/Upper-Ontologien abzudecken. Upper-Ontologien werden eine Schlüsselrolle für die Beschreibung verschiedener Domänen einnehmen [Mascardi et al. 2007]. Beispiele für Upper-Ontologien umfassen Basic Formal Ontology (BFO), Business Objects Reference Ontology (BORO), Conceptual Reference Model (CIDOC), und Descriptive Ontology for Linguistic or Cognitive Engineering (DOLCE). Domänenontologien verwenden Upper-Ontologien für grundlegende Konzepte und Eigenschaften, beschreiben jedoch alle Besonderheiten der Domäne mit eigenen Konzepten.
Der Entwurf von Testfällen als Ontologie liegt in der Verantwortung einer Domänenontologie. Eine entsprechende Domänenontologie muss die Informationsgruppen, die Metainformationen und die gesamte Testfallstruktur repräsentieren können. Die verfügbaren Domain-Ontologien zum Testen sind z.B. STOWS [Zhang und Zhu 2008], SWTO [Bezerra et al. 2009], OntoTest [Barbosa et al. 2006] und aktuelle Verbesserungen in [Guido Tebes et al. 2020], welche den Fokus auf Aufbau und Ablauf eines Testprozesses im Bereich Softwaretest legen.

Die aufgeführten Ontologien können Testfälle nur als Ressource und nicht deren Inhalt darstellen. Spezifische Domänenontologien für die Domäne des HIL-Testens, die für den **Schritt 1: Interoperabilität der Testdaten herstellen** als Informationsmodell benötigt werden, sind nicht bekannt. Eine betrachtete Lösung für die Abbildung von Testfällen besteht darin, eine dieser Domänenontologien zu erweitern und die Testfallinhalte grundlegend zu modellieren. Die Erweiterung würde die Struktur eines Testfalls und testautomatisierungsspezifische Inhalte beinhalten. Die beabsichtigte Verwendung der Ontologie besteht darin, sie als Informationsmodell zu verwenden, ohne dass Abfragen und Konsistenzprüfungen einer Ontologie erforderlich sind. Die hier präsentierte Lösung ist flexible in der Wahl des Informationsmodells. Andere Modelle in der Anwendungsdomäne des modellbasierten Testens und der Softwarespezifikation sind näher an der formalen Spezifikation der Testfallinhalte und könnten die Rolle eines Informationsmodells übernehmen. Aus diesem Grund wird die mögliche Wiederverwendung dieser Modelle dargestellt.

4.2.1.2 Testmodelle

Ein Testmodell ist ein formales Model und beschreibt das benötigte Systemverhalten [Dias Neto et al. 2007]. Das beschriebene Verhalten ist eine Aggregation von Testfällen und wird für die fallspezifische Ableitung von Testfällen genutzt. Hierfür werden Abdeckungskriterien, wie beispielsweise Abdeckung der Anweisungen oder Abdeckung der Strukturen [Giese et al. 2007] genutzt, um den abzudeckenden Testraum zu definieren. Basierend auf dem Testraum werden manuelle oder automatische Methoden zur Ableitung und Parametrisierung von Testfällen angewendet. Hierbei ist beispielsweise die Klassifikationsbaummethode eine gängige Methode. Bei dieser Methode definiert der Testingenieur die relevanten Signale mit ihren Werten als Äquivalenzklassen. Anschließend bestimmt die Methode automatisiert die relevante Menge an Testfällen [Franke et al. 2012]. Die aggregierte Darstellung von Testfällen ermöglicht nicht die Identifizierung eines Testfalls als eigenständiges Element, sondern ist das Ergebnis einer Parametrisierung und anschließenden Ableitung. Das Verhalten des Systems wird überwiegend in Zustandsdiagrammen oder davon abgeleitete Varianten gespeichert. Tabelle 16 zeigt eine Übersicht über die verwendeten Testmodelle aus dem Jahre 2007 [Dias Neto et al. 2007].

Tabelle 16: Verwendete Testmodelle, übernommen von [Dias Neto et al. 2007]

Verhaltensmodell	**Ansätze für das modellbasierte Testen**
Zustandsdiagramm	27
Klassendiagramm	19

Sequenzdiagramm	19
Anwendungsfalldiagramm	11
Object Constraint Language (OCL)	11
(Erweitert) Endlicher Automat	10
Aktivitätsdiagramm	9
Kommunikationsdiagramm	8
Objektdiagramm	7
Graph	7
Z Spezifikation	4

Die verfügbaren Anbieter von Modell basierten Testlösungen für das Testen von mechatronischen Systemen in den Sektoren Automotive und Avionik sind u.a. dSPACE GmbH, Verified Systems International GmbH und Vector Informatik GmbH. Deren Lösungen für Testmodelle basieren immer noch auf Zustandsdiagrammen. Zum Beispiel verwendet der RT-Tester von Verified Systems International GmbH UML und SysML [Peleska 2013] und dSPACE verwendet den ASAM XIL Standard.

Testmodelle als Aggregation von Testfällen haben gegenüber dem Ansatz der Darstellung eines Testfalls in einem Zustandsdiagramm Nachteile, welche im Folgenden beschrieben werden. Um einen Testfall aus einem Testmodell in einer anderen „fremden“ *Testskriptsprache* zu erhalten, müsste die Transformation des Testmodells vom Integrator in das Testmodell des Lieferanten übersetzt werden. Anschließend könnten die Testfälle aus dem Testmodell des Lieferanten generiert und ein geeigneter Testfall ausgewählt werden. Hierfür müsste eine geeignete Methode für die Parametrierung ausgewählt werden. Die Transformation von Modellen, welche auf XML basieren, ist bereits mit generischen Transformationstools möglich. Ein Beispiel ist das OPM-Modellierungstool [Shani et al. 2017] oder Transformationssprachen, wie etwa Query View Transformation (MOF QVT). Die Anwendung dieses Ansatzes hat zwei Nachteile. Der erste ist die richtige Auswahl von Testfällen innerhalb eines Korpus von Testfällen, und der zweite ist die geeignete Parametrierung, um denselben Satz ähnlicher Testfälle zu generieren. Daher ist diese Lösung nicht auf die Interoperabilität von Testfällen in der Zulieferpyramide anwendbar.

4.2.1.3 Wissensgraphen

Es gibt verschiedene Definitionen von Wissensgraphen [Singhal 2012]. Im Rahmen der Arbeit wird die Definition von Hogan angewendet: *Herein, we define a knowledge graph as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities* [Hogan et al. 2021]. Es sind keine Graphen bekannt die bereits Testfälle beschreiben können. Ein Graph müsste hierfür das SUT darstellen, welches durch eine unterschiedliche Anzahl von Testfällen (d. h. Knoten) beschrieben würde. Es würde Informationen über die Stimuli und korrekten Verhaltensweisen enthalten, die für eine bestimmte Anforderung oder einen funktionalen Aspekt des Testfalls definiert sind. Dieses Wissen kann je nach angewandtem Modell unterschiedlich dargestellt werden. Genauer gesagt ist der Wissensgraph definiert als ein *Directed Edge-labelled Graph* [Hogan et al. 2021]. Jeder Knoten repräsentiert ein reales Objekt. Die gerichteten und beschrifteten Kanten definieren die Beziehungen zwischen den Zuständen. Um Taxonomien einzuschließen, kann jeder Knoten und jedes Label einen Typ haben, was durch *Heterogeneous Graphs* unterstützt wird [Hogan et al. 2021]. Wenn ein reales Objekt über eine Eigenschaft beschrieben werden soll, wird ein *Property Graph* [Hogan et al. 2021] benötigt. Dementsprechend ist die Repräsentation von Testfällen als Graph möglich. Das hierfür benötigte Schema müsste komplett neu entwickelt werden.

4.2.2 Auswahl der Informationsmodelle

Es werden für die Umsetzung des Konzeptes zwei Informationsmodelle benötigt: für den **Schritt 1: Interoperabilität der Testdaten herstellen** und für die Fehlersuchmethode 2. Der Hintergrund ist, dass beide Informationsmodelle für ihr Einsatzszenario optimiert werden. Im Folgenden wird eine Auswahl für beide Informationsmodelle getroffen.

Die kollaborative Fehlersuche erzeugt im **Schritt 1: Interoperabilität der Testdaten herstellen** aus den Testfällen ein Informationsmodell, um eine Transformation in die Zielformate für die Fehlersuchmethoden 1 und 2 zu ermöglichen. Die gezielte Abfrage von testrelevanten Informationen aus dem Modell ist nur für die Fehlersuchmethoden relevant und nicht für das Informationsmodell. Diesbezüglich können auch existierende Modelle für die Realisierung von Zustandsdiagrammen genutzt werden, solange man diese um testfallspezifische Elemente erweitert. Der Vorteil hierbei ist, dass Testfälle sich als Zustandsdiagramme abbilden lassen und hierfür die benötigten Elemente beinhalten. Die Wahl des Zustandsdiagramms wird im Kapitel 5 erfolgen.

Im Gegensatz zum Informationsmodell des ersten Schrittes der Fehlersuche benötigt die Fehlersuchmethode 2 für eine statische Analyse eine obligatorische Anfragesprache. Die Mächtigkeit der Anfragesprache hängt von der Art des zugrundeliegenden Modells ab. Der übliche Ansatz besteht darin, eine Anfragesprache wie SPARQL, Cypher oder Gremlin für Informationen bereitzustellen. Dementsprechend sind Ontologien und Graphen geeignet. Für jeden Graphentyp und jede Anfragesprache gibt es jedoch kommerzielle Lösungen, die ständig verbessert werden und deren detaillierte Darstellung den Rahmen dieser Arbeit übersteigt.

Ein geeignetes Informationsmodell für die Fehlersuchmethode 2 besteht aus den Anweisungen der Informationsgruppen und den Metainformationen der Testfälle. Die unterschiedlichen Anweisungen haben einen gleichen Aufbau bei unterschiedlicher Bedeutung. Dementsprechend werden die Struktur und die darauf basierenden Suchanfragen für Anweisungen einen ähnlichen Aufbau besitzen.

Um Unterschiede innerhalb von Anweisungen über alle Gruppe gleich identifizieren zu können, werden Informationen als Tripels {Subjekt, Prädikat, Objekt} auf der Informationsebene abgebildet. Die Testfälle werden als eine Menge von Tripeln für die Suche nach Unterschieden in die Fehlersuchmethoden eingespeist. Zum Beispiel ist das Triple {Sensor A, Wert, 20} eine mögliche Eingabe oder {Sensor A, Soll-Wert größer, 21}. Ein Informationsmodell bestehend aus den Tripeln soll die Semantik der Testfälle beschreiben und wird im Rahmen der Spezifikation in einem Informationsmodell unabhängig von einer spezifischen *Testskriptsprache* für die Analyse vorgehalten werden.

Um Unterschiede in den Metainformationen identifizieren zu können, werden diese auch als Tripels dargestellt, wobei das Subjekt die annotierte Anweisung ist. Hierdurch ergibt sich für jede Metainformation ein Tripel. Diese Anforderungen werden sowohl von Ontologien als auch von Graphen erfüllt. Die konkrete Auswahl der Technologie erfolgt im Kapitel 5.

4.3 Fehlersuchmethoden der Fehlersuche

Der interoperable Testfall ist das Ergebnis von **Schritt 1: Interoperabilität der Testdaten herstellen** und wird als Eingabe für den **Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen** genutzt. Der Schritt 2 umfasst drei Fehlersuchmethoden: *Ausführung des Testfalls und Analyse des Signalverlaufs*, *Mustersuche* und *Anpassung der Konfiguration und anschließende Ausführung*, welche in Abbildung 31 abgebildet sind. Die Eingabe für die ersten beiden Fehlersuchmethoden ist der interoperable Testfall des Integrators. Das Ergebnis der drei Fehlersuchmethoden sind identifizierte Unterschiede in den Testfällen oder der Konfiguration der *lose gekoppelten* Testprozesse. Die Bedeutung der Unterschiede für die Testprozesse wird anschließend skizziert.

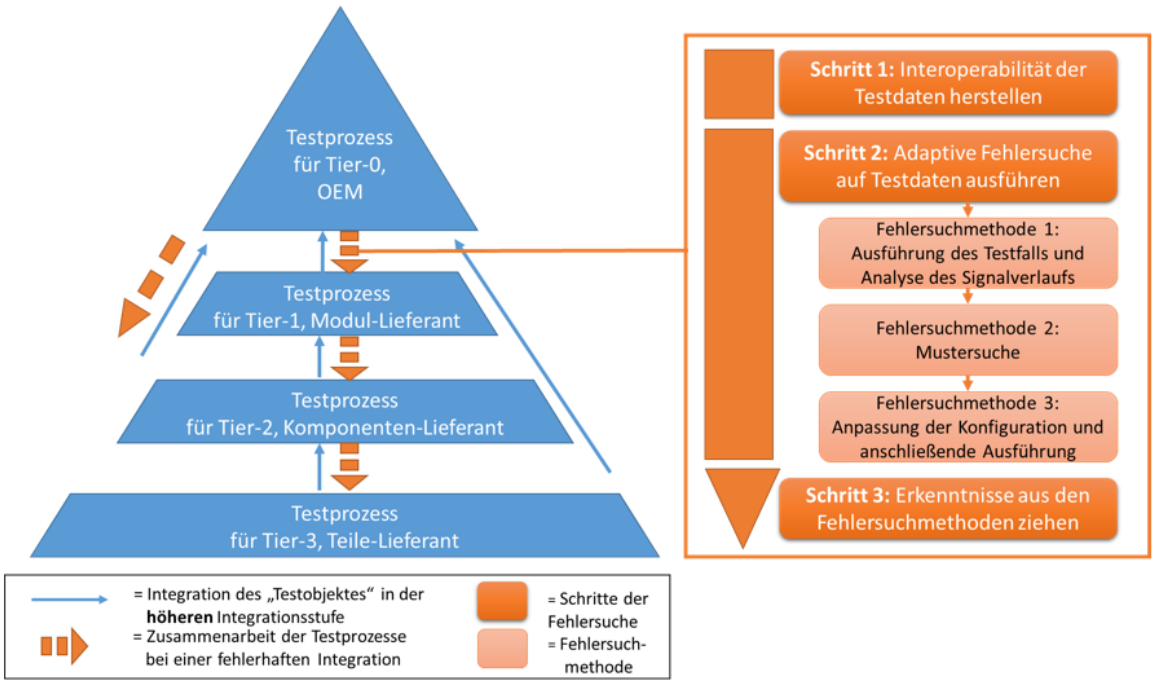

Abbildung 31: Position der Fehlersuchmethoden in der kollaborativen Fehlersuche

Die Fehlersuchmethoden testen auf Unterschiede, wie das SUT in den Testausführungen der Testprozesse stimuliert und geprüft wurde. Solche Unterschiede können zu einem unterschiedlichen Verhalten des SUT in den Testprozessen führen und entsprechend zu unterschiedlichen Testergebnissen. Die gefundenen Unterschiede werden als Eingabe für den **Schritt 3: Erkenntnisse aus den Fehlersuchmethoden ziehen** als Indikatoren bereitgestellt.

Ein Indikator ist in der Dissertation ein Hilfsmittel, mit dessen Hilfe der prüfbare Unterschied zwischen zwei Testprozessen klassifiziert werden kann. Hierbei kann ein Indikator basierend auf einer dynamischen Analysemethode [ISTQB Glossary 2021] oder mithilfe einer statischen Analysemethode [ISTQB Glossary 2021] ermittelt werden. Im Rahmen der dynamischen Analysemethode kann der Testfall ausgeführt, das Verhalten des Prüflings beobachtet und ausgewertet werden. Hierfür kann im Rahmen der Testvorbereitung der Testfall oder die Konfiguration gewählt bzw. angepasst werden. Dabei ist zu erwähnen, dass über die Konfiguration nicht nur die Konstanten, sondern auch beispielsweise das Prüfmittel (HIL Testsystem oder Testsystem mit realem System), das Simulationsmodell oder die Abbildungen der Signale konfiguriert werden können. Die Veränderung des Testfalls oder der Konfiguration kann zu gleichen Fehlerbildern in der Testausführung führen, aber die Ursache des Fehlverhaltens kann unterschiedlich sein. Die Zuordnung von Fehlerbildern auf Unterschiede zwischen den Testprozessen, wie beispielsweise die abweichende

Stimulierung des Prüflings in den Testfällen, soll mithilfe der Indikatoren beschrieben werden. Zum Beispiel kann ein Verdikt fehlschlagen (Fehlerbild), was durch eine falsche Stimulierung des Prüflings (Fehler) über die Stimulierung im Testfall (Unterschied in der Struktur der Testfälle wurde gefunden) verursacht wurde. Der Grund hierfür kann aber in einem Unterschied innerhalb der Testfälle oder der Konfigurationen liegen. Um die Quelle des Fehlers für dynamische Analysemethoden einordnen zu können, werden hierfür zwei Indikatorgruppen definiert, nämlich: *Unterschiede im Laufzeitverhalten des Testfalls* und *Strukturelle Unterschiede in der Konfiguration.* Abgesehen von der dynamischen Analysemethode gibt es auch eine statische Analysemethode, in denen der Testfall nicht ausgeführt wird. Bei der statischen Analyse können Unterschiede in den Testfällen gefunden und als Indikator definiert werden. Dementsprechend wird hierfür die dritte Indikatorgruppe *Strukturelle Unterschiede im Testfall* ergänzt. Die verfügbaren Gruppen sind zusammengefasst *Unterschiede im Laufzeitverhalten des Testfalls*, *Strukturelle Unterschiede im Testfall* und *Strukturelle Unterschiede in der Konfiguration*.

Ein Indikator besitzt somit den Fehler und den Unterschied, welcher der Grund für den Fehler sein könnte. Des Weiteren wird ein Indikator auch einer Informationsgruppe zugewiesen, um die Abbildung zwischen Anweisungen des Testfalls, welche auch einer Gruppe zugeordnet sind, und dem Indikator zu ermöglichen. Hierdurch wird die Lokalisierung des Unterschieds innerhalb eines Testfalls erleichtert, weil Filtermethoden eingesetzt werden können. Tabelle 17 listet alle Indikatoren. Die Menge an Indikatoren ergibt sich aus der Kombination von dynamischen und statischen Analysemethoden, die durch die Wahl der Testfälle und Konfigurationen parametrisiert werden können. Eine Fehlersuchmethode verwendet nur eine Art der Analysemethode, weswegen keine Fehlersuchmethode alle Indikatoren erkennen kann. Eine Auflistung der erkennbaren Indikatoren pro Fehlersuchmethode ist in Tabelle 18 gegeben.

Tabelle 17: *Indikatoren* der Fehlersuchmethoden

Gruppe	**Informationsgruppe**	**Auswirkungen auf das SUT**	**Unterschiede als mögliche Gründe**	**Beschreibung des Fehlers**	**Indikator**
Unterschiede im Laufzeitverhalten des Testfalls	*Stimulus*	Signal welches direkt stimuliert wurde	Signal wird unterschiedlich durch die Testfälle oder durch die Konfiguration stimuliert.	Das Signal wird über eine Anweisung im Testfall stimuliert und verändert anschließend seinen Wert in der Testausführung. Dieser Wert ist unterschiedlich in den Signalverläufen	STIMULUS-1

				der beiden Testausführungen.	
		Signal welches transitiv stimuliert wurde	Signal wird unterschiedlich durch den Testfall oder der Konfiguration stimuliert.	Das Signal wird transitiv über ein anderes Signal stimuliert, welches als eine Anweisung im Testfall enthalten ist. Dieser Wert ist unterschiedlich in den Signalverläufen der beiden Testausführungen.	STIMULUS-2
	Verdikt	Das Ergebnis des Verdikts	Es gibt zwei mögliche Gründe: Signal der Verdickt Bedingung wird unterschiedlich durch die Testfälle oder durch die Konfiguration stimuliert. Die Bedingungen der Verdickte sind unterschiedlich	Das Ergebnis eines Verdikts ist entweder erfolgreich oder fehlgeschlagen. Die Ergebnisse des Verdikts mit derselben ID unterscheiden sich in den Testberichten	VERDIKT-1
	Laufzeitverhalten	Events für das Zeitverhalten	Die Stimulierung hat bei beiden Testsystemen unterschiedlich lang gedauert Die Events wurden unterschiedlich im Testfall positioniert	Es gibt Events, welche den Fortschritt des Testfalls bezogen auf Testschritte anzeigen. Diese Events können bezüglich des Auftretens und dem Zeitpunkt des Auftretens variieren, was in den Testberichten enthalten ist.	LAUF-1
Strukturelle Unterschiede	*Stimulus*	Stimulus eines Signals	Signal wird unterschiedlich durch die Testfälle stimuliert.	Die Definition des Stimulus ist über den Signalnamen und den Wert definiert. Hierüber können Unterschiede in der Stimulierung des SUT in	STIMULUS-3

im Testfall				den Testfällen identifiziert werden.	
	Verdikt	Definition eines Verdikts	Verdickt wird unterschiedlich durch die Testfälle stimuliert.	Die Definition des Verdikts ist über den Signalnamen und ihrer Bedingung definiert. Hierüber können Unterschiede in den Testfällen identifiziert werden.	VERDIKT-2
	Laufzeitverhalten	Definition einer Anweisung zum Warten	Die Events wurden unterschiedlich im Testfall positioniert	Die Definition der Anweisung zum Warten ist über die Dauer oder über eine Bedingung definiert. Hierüber können Unterschiede in den Testfällen identifiziert werden.	LAUF-2
	Stimulus	Konstanten	Die unterschiedlichen Ausprägungen von Konstanten sind in den Dateien unterschiedlich	Konstanten werden in der Konfiguration mit einem Typ und einem Wert erstellt. Die unterschiedlichen Ausprägungen von Konstanten können als Unterschiede in der Stimulierung erkannt werden.	STIMULUS-4
Strukturelle Unterschiede in der Konfiguration	*Stimulus*	Konstanten	Die unterschiedlichen Ausprägungen von Konstanten sind in den Dateien unterschiedlich	Die Stimulierung der Signale sehen in den Signalverläufen und Testberichten unterschiedlich aus.	STIMULUS-5
		Abbildung der Signale des Testfalls auf die echten Signale des SUT	Die Abbildung eines Signals verknüpft den Signalnamen des Testfalls mit dem Signalnamen innerhalb des Testsystems, welche durch eine IO oder durch ein Simula-	Das gleiche Signal hat unterschiedliche Signalverläufe	STIMULUS-6

			tionsmodell repräsentiert werden kann. Hierüber können Unterschiede in den Konfigurationen identifiziert werden.		
		Definition des Signals	Die Definition des Signals definiert die physikalischen Eigenschaften wie Datentyp oder Kommunikationsmedium. Hierüber können Unterschiede in den Konfigurationen identifiziert werden.	Das gleiche Signal hat unterschiedliche Signalverläufe	STIMULUS-7
	Verdikt	Konstanten	Die Definition der Konstante ist über einen Wert definiert. Hierüber können Unterschiede in den Konfigurationen identifiziert werden.	Die Auswertung der Verdickte sind in den Testberichten unterschiedlich	VERDIKT-3

Tabelle 18: Abbildung von *Indikatoren* auf die Fehlersuchmethoden

Fehlersuchmethode	**Indikator**
Fehlersuchmethode 1	STIMULUS-1, STIMULUS-2, VERDIKT-1, LAUF-1
Fehlersuchmethode 2	STIMULUS-3, VERDIKT-2, LAUF-2, STIMULUS-4
Fehlersuchmethode 3	STIMULUS-5, STIMULUS-6, STIMULUS-7, VERDIKT-3

Keine der Fehlersuchmethoden kann alle Indikatoren identifizieren. Deswegen müssen im **Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen** alle Fehlersuchmethoden nacheinander ausgeführt werden. Hierbei ist der Lieferant (Tier-i+1), welcher ein Komponenten-Lieferant oder Modul-Lieferant ist, der Anwender der Fehlersuchmethode.

Die drei Fehlersuchmethoden haben unterschiedliche Voraussetzungen. Die Fehlersuchmethode 1 benötigt die Ausführung des fehlgeschlagenen Testfalls in dem Testprozess des Lieferanten (Tier-i+1). Hierfür wird die Fähigkeit der Ausführung des fehlgeschlagenen Testfalls auf dem Testsystem des Lieferanten (Tier-i+1) benötigt. Ist die Voraussetzung nicht gegeben, können die Fehlersuchmethoden 2 und 3 trotzdem angewendet werden. Die Voraussetzung der zweiten Fehlersuchmethode ist, dass der fehlgeschlagene Testfall des Integrators (Tier-i) und des Lieferanten (Tier-i+1) interoperabel sind und darauf eine Mustersuche angewendet werden kann. Die Fehlersuchmethode3 stellt keine zusätzlichen Voraussetzungen an den Testprozess des Lieferanten (Tier-i+1). Eine Übersicht über die Fehlersuchmethoden und ihre Anwendungsreihenfolge ist in der Abbildung 32 gegeben.

Jede Fehlersuchmethode wird folgend detailliert mithilfe der **Ausgangssituation**, dem **Ansatz** und der **Vorgehensweise** vorgestellt. Hierbei wird in der Ausgangssituation beschrieben, unter welchen Bedingungen die Fehlersuchmethode angewendet werden kann. Zu diesem Zweck wird die notwendige Interoperabilitätsstufe (sie Unterkapitel 4.1) des Testfalls und der Konfiguration als Eingabe für die Fehlersuchmethode definiert. Basierend auf der Ausgangssituation wird der Ansatz vorgestellt. Abschließend wird die Vorgehensweise für die Fehlersuchmethode vorgestellt. Bei der Vorgehensweise werden die Unterschiede (Tabelle 18) als Indikatoren aufgezählt, welche identifiziert werden können.

Jede Fehlersuchmethode wird von zwei Testprozessen begleitet, wobei ein Testprozess die höhere Integrationsstufe und der andere Testprozess die niedrigere Integrationsstufe darstellt, was der losen Kopplung der Testprozesse entspricht. Für die folgenden Ausführungen wird davon ausgegangen, dass der Testprozess des OEM der Testprozess mit der höheren Integrationsstufe und der Testprozess des Modul-Lieferanten der Testprozess mit der niedrigeren Integrationsstufe ist.

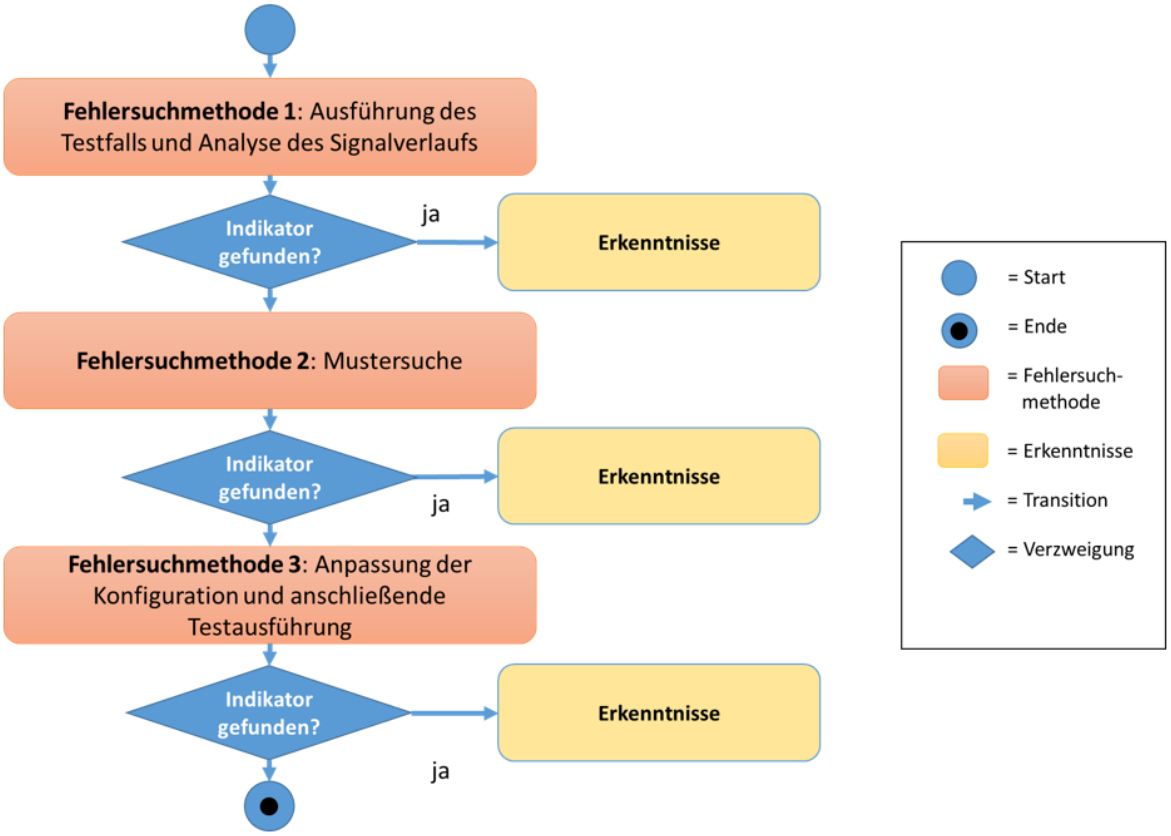

Abbildung 32: Sequenzieller Ablauf der Anwendung der Fehlersuchmethoden (Schritt 2 der Abbildung 29)

4.3.1 Fehlersuchmethode 1: Ausführung des Testfalls und Analyse des Signalverlaufs

Ausgangszustand
Der Ausgangszustand ist der Start der kollaborativen Fehlersuche. Hierbei wird ein Testfall im Testprozess vom OEM und ein Testfall vom Modul-Lieferanten für dieselbe Funktion getestet und die Ergebnisse sind unterschiedlich. In der konkreten Situation ist der Testfall vom OEM fehlgeschlagen.

Diese Fehlersuchmethode benötigt als Eingabe die Konfiguration des Modul-Lieferanten und beide Testfälle, wobei der Testfall des OEM bereits in einem interoperablen Format vorliegt.

Ansatz
Diese Methode ist eine dynamische Analysemethode [HEICON Global Engineering GmbH 2020; ISTQB Glossary 2021]. Hierbei wird nicht primär die Struktur des Testfalls, sondern die Auswirkungen der Testfallausführung auf das Verhalten des

SUT untersucht. Das Ergebnis der Testausführung ist der Testbericht vom fehlgeschlagenen Testfall, welcher in dem gleichen Format wie der vergleichbare bestandene Testfall ist. Der Untersuchungsgegenstand sind die Signalverläufe, die Ergebnisse des Verdikts und Statusinformationen aus den beiden Testberichten.

Erkennbare Indikatoren: STIMULUS-1, STIMULUS-2, VERDIKT-1, LAUF-1

Vorgehensweise

Es gibt zwei Varianten der Methode. Die erste Variante überprüft, ob die Ausführung der Testfälle zu unterschiedlichen Verhalten des SUT führen. Hierdurch können Fehler in der Art und Wiese festgestellt werden, wie ein SUT stimuliert und anschließend überprüft wird. Die zweite Variante überprüft, ob das Testbed, bestehend aus Testsystem und Simulationsmodell, sich beim OEM und Modul-Lieferanten gleich verhält. Es werden nachfolgend beide Varianten vorgestellt.

Variante 1

Um die Testausführung der beiden Testfälle vergleichen zu können, müssen diese auf dem gleichen Testsystem ausgeführt werden. Um dies zu ermöglichen, wird das Testsystem des Testprozesses vom Modul-Lieferanten als Basis festgelegt. Dabei wird der folgende Ablauf sowohl mit der Konfiguration des Modul-Lieferanten als auch mit der Konfiguration des OEM durchlaufen.

Die Ausführung des Testfalls vom OEM ist beim Modul-Lieferanten nicht direkt möglich, da die Testprozesse unterschiedliche *Testskriptsprachen* verwenden. Im Rahmen des 1. Schrittes der Fehlersuche wurde der Testfall bereits in ein Informationsmodell überführt. Dieses Informationsmodell wird verwendet, um den Testfall aus dem Informationsmodell in die Zielsprache zu übersetzen. Die Zielsprache ist die *Testskriptsprache* des Modul-Lieferanten. Anschließend werden beide Testfälle auf demselben Testsystem und mit derselben Konfiguration ausgeführt. Als Ergebnis erhält man zwei Testberichte, die auf Unterschiede im Laufzeitverhalten untersucht werden können. Jeder gefundene Unterschied wird manuell durch die Testingenieure auf einen Indikator abgebildet. Es werden die Signalverläufe des OEM-Testfalls und des Modul-Lieferanten aus den Testberichten extrahiert und in einem Diagramm hinzugefügt. Dieses Diagramm ermöglicht die grafische Auswertung der Signalverläufe.

Anhand der jeweiligen Signalverläufe können die Stimuli über die Zeit verglichen werden. Zeigen sich hier bereits Unterschiede, so kann sich das SUT bereits unterschiedlich verhalten und zu unterschiedlichen Zielzuständen gelangen. Im zweiten Schritt werden die Verdikte dahingehend analysiert, ob sie die gleichen Ergebnisse repräsentieren. Das Ergebnis der Verdikte ist in den Testberichten enthalten. Zeigen die vergleichbaren Verdikte in den Testberichten unterschiedliche Ergebnisse, kann

dies an den verwendeten Anweisungen für das Zeitverhalten oder an unterschiedlichen Stimuli liegen. Aus der manuellen Analyse durch die Testingenieure ergeben sich weitere Analysemöglichkeiten basierend auf den Testberichten, die jedoch nicht weiter betrachtet werden.

Variante 2

Die Variante 2 unterscheidet sich zu der ersten Variante, indem beide Testprozesse den fehlgeschlagenen Testfall des OEM ausführen. Anschließend erfolgt der Vergleich der Signalverläufe der Testausführung des fehlgeschlagenen Testfalls beim OEM mit der Testausführung des fehlgeschlagenen Testfalls beim Modul-Lieferanten. Die Analyse erfolgt ebenfalls über die Auswertung der Signalverläufe und der Verdikte. Unterschiede in den Signalverläufen oder den Verdikten können bei semantisch gleichen Testfällen auf unterschiedliche Eigenschaften des Testsystems, auf Unterschiede in den verwendeten Simulationsmodellen oder auf Unterschiede zwischen Simulationsmodell und realem System hinweisen.

4.3.2 Fehlersuchmethode 2: Mustersuche

Ausgangszustand
Der Ausgangszustand ist der gleiche wie bei der ersten Fehlersuchmethode.

Die Fehlersuchmethode benötigt zusätzlich als Eingabe beide Testfälle, wobei der Testfall vom OEM bereits in einem interoperablen Format vorliegt.

Ansatz
Diese Methode ist eine statische Analysemethode und testet das SUT ohne es auszuführen [HEICON Global Engineering GmbH 2020; ISTQB Glossary 2021]. Hierbei werden primär die Struktur des Testobjektes und seine Dokumente, wie Testfälle, analysiert. Hierdurch können Analysen durchgeführt werden, wie das SUT stimuliert und welche Systemreaktion erwartet werden. Die konkreten Auswirkungen des Testfalls können hierbei nicht am SUT nachvollzogen werden, weil der Testfall nicht ausgeführt wird. Der Untersuchungsgegenstand sind Stimuli, die definierten Verdikte und das definierte Laufzeitverhalten aus den Testfällen.

Erkennbare Indikatoren: STIMULUS-3, STIMULUS-4, VERDIKT-2, LAUF-2

Vorgehensweise
Um Muster aus der Gruppe Stimuli, Verdikte und Laufzeitverhalten zwischen Testfällen erkennen zu können, reicht ein herkömmlicher Textvergleich nicht aus. Hintergrund ist, dass Testfälle einen unterschiedlichen strukturellen Aufbau und eine unterschiedliche Benennung der Zustände haben können und dennoch semantisch identisch sind. Um eine Mustersuche zu ermöglichen, werden beide Testfälle zunächst in das Informationsmodell aus Schritt 1 und anschließend in einen Graphen

überführt. Der Graph enthält nur die konkreten Anweisungen des Testfalls als Knoten und Kanten. Die strukturellen Informationen der Testfälle, wie die Zugehörigkeit zu Blöcken und Testschritten, werden als Metainformationen gespeichert. Anschließend wird die Suche nach Subgraphen angewendet, um Muster in Testfällen zu finden. Hierfür wird ein Testfall in ein Muster überführt und mithilfe einer Ähnlichkeitssuche über die Testfälle gesucht. Basierend auf den gefundenen Testfällen können in einem zweiten Schritt die Unterschiede ausgelesen und die Indikatoren mit Hilfe einer manuellen Analyse identifiziert werden.

4.3.3 Fehlersuchmethode 3: Anpassung der Konfiguration und anschließende Ausführung

Ausgangszustand
Der Ausgangszustand ist der gleiche wie bei der ersten Fehlersuchmethode.

Die Fehlersuchmethode benötigt zusätzlich als Eingabe Konfigurationen, welche erfolgreich für den fehlgeschlagenen Testfall funktioniert haben.

Ansatz
Diese Methode ist eine dynamische Analysemethode [HEICON Global Engineering GmbH 2020; ISTQB Glossary 2021]. Hierbei wird nicht primär die Struktur des Testfalls, sondern die Auswirkungen der Testfallausführung auf das Verhalten des SUT bei der Anwendung von unterschiedlichen Konfigurationen analysiert. Das Ergebnis sind Testberichte. Der Untersuchungsgegenstand sind die Signalverläufe, die Ergebnisse des Verdikts und Statusinformationen, welche in Form von Ereignissen und Logs im Testbericht vorliegen. Hierfür soll der fehlgeschlagene Testfall vom OEM so lange mit unterschiedlichen Konfigurationen ausgeführt werden, bis der Testbericht dem Testbericht vom Modul-Lieferanten ähnelt.

Erkennbare Indikatoren: STIMULUS-5, STIMULUS-6, STIMULUS-7, VERDIKT-3, LAUF-1

Vorgehensweise
Zunächst werden ähnliche Konfigurationen für den fehlgeschlagenen Testfall identifiziert. Dazu werden vergangene Testläufe nach dem Testfall gefiltert. Das Ergebnis der Filterung ist eine Menge von Testläufen, bei denen der Testfall mit verschiedenen Konfigurationen ausgeführt wurde. Die Konfigurationen aus den erfolgreichen Testläufen werden dann als Eingabe für die Fehlersuchmethode 3 verwendet.

Die Testausführung mit dem Testfall und den gefilterten Konfigurationen identifiziert die Konfiguration, die zum gleichen Testergebnis des Modul-Lieferanten führt. Anschließend können die Unterschiede zwischen den getesteten Konfigurationen und der Originalkonfiguration des OEM verglichen werden. Schließlich kann der

identifizierte Satz fehlerhafter Konstanten als Indikator für die manuelle Analyse verwendet werden.

5 Spezifikation und Implementierung der kollaborativen Fehlersuche

Das Ziel dieses Kapitels ist die Spezifikation und Implementierung der Schritte 1 und 2 der kollaborativen Fehlersuche. Für Schritt 3 der kollaborativen Fehlersuche werden keine Funktionsmuster entwickelt, da es sich bei Schritt 3 um eine manuelle Analyse durch den Testingenieur handelt. Als nächstes werden die Unterkapitel kurz vorgestellt.

In Unterkapitel 5.1 wird das Informationsmodell für die interoperable Darstellung von Testfällen für den Schritt 1 der kollaborativen Fehlersuche spezifiziert. Anschließend wird die Transformation der Testfälle in das Informationsmodell beschrieben. Die Testfälle im Informationsmodell sind das Ergebnis von Schritt 1 der kollaborativen Fehlersuche und werden anschließend als Input für die Fehlersuchmethoden 1 und 2 verwendet, welche Teil von Schritt 2 der Fehlersuche sind. In den folgenden Unterkapiteln (5.2- 5.4) werden die drei Fehlersuchmethoden spezifiziert. Für jede der drei entwickelten Fehlersuchmethoden werden die erforderlichen Informationsmodelle und Transformationen vorgestellt.

Das Ergebnis der Implementierung sind zwei Prototypen, nämlich die RCP-Anwendung *Test Case Translator* und die Web-Anwendung *Test Advisor*. Dabei ermöglicht der *Test Case Translator* die Transkompilierung von Testfällen im Rahmen von der Fehlersuchmethode 1 und der *Test Advisor* die Aggregation und Nutzung von testprozessrelevanten Informationen im Rahmen der Fehlersuchmethode 2 und 3. Somit ermöglichen die Prototypen die Durchführung der im Konzept beschriebenen Fehlersuchmethoden. Dementsprechend ist die Vorgehensweise der Fehlersuchmethoden als Anwendungsfall in das User Interface aufgenommen worden.

5.1 Schritt 1 der kollaborativen Fehlersuche: Interoperabilität der Testdaten herstellen

Der erste Schritt der Fehlersuche konzentriert sich auf die Interoperabilität von Testfällen und Konfigurationen. Die Interoperabilität von Konfigurationen, die aus Wertepaaren bestehen und mittels JSON, YAML oder XML kodiert sind, kann mit existierenden Datenintegrationslösungen für semistrukturierte Datenquellen realisiert werden. Dazu können beispielsweise Lösungen auf Basis des Semantic Mediator, der SPARQL als Anfragesprache verwendet, [Franke et al. 2021] oder Lion, welche GraphQL als Anfragesprache nutzen würde, [Tubbs 2018] integriert werden. Dementsprechend liegt der Fokus auf der Entwicklung eines Informationsmodells und Datenintegrationsansätzen für Testfälle, die strukturierte Datenquellen sind. Die folgenden Unterkapitel sind aus [Franke und Thoben 2022] übernommen und stellenweise angepasst.

In Unterkapitel 5.1 wird das Informationsmodell für Testfälle spezifiziert und hierbei werden die Anforderungen für den Aufbau und den Inhalt eines Testfalls aus dem dritten Kapitel berücksichtigt. Im Unterkapitel 5.1.2 wird anschließend die Transformation für den Datenintegrationsansatz im Detail vorgestellt. Hierbei wird die Transformation basierend auf der *Testskriptsprache* CCDL vorgestellt, aber sie ist grundsätzlich anwendbar auf alle *Testskriptsprachen*.

5.1.1 Informationsmodell eines Testfalls als Zustandsübergangsdiagramm

Das Informationsmodell ist ein Informationsmodell. Der Ansatz ist die Repräsentation eines Testfalls als ein Zustandsübergangsdiagramm, welches um testfallspezifische Anweisungen erweitert wird. Das Informationsmodell soll die Rolle der interoperablen Darstellung von Testfällen einnehmen. Basierend darauf soll es verwendet werden, um einen Testfall nahtlos in andere Testprozesse zu integrieren.

Jeder Testfall ist als Zustandsübergangsdiagramm darstellbar, unter Berücksichtigung der Abfolge von Anweisungen. Hierbei führt jede Stimulierung des SUT oder das Pausieren der Testausführung zu einem neuen Zustand. Testfälle besitzen eine logische Struktur, in welche der Testfall in logische Blöcke unterteilt sind. Diese Blöcke sind sogenannte Testschritte und müssen im Zustandsübergangsdiagramm abbildbar sein. Hintergrund ist, dass die Testschritte die Lesbarkeit der Testfälle steigern und die Auswertung der Testausführung vereinfachen. Somit wird ein Zustandsübergangsdiagramm benötigt, welches hierarchische Zustände abbilden kann. Nur das Harel-Zustandsdiagramm [Harel 1987] ist in der Lage, hierarchische Zustände zu definieren. Daher können bestehende Standards für Harel-Zustandsdiagramme einen Testfall repräsentieren, wenn alle Sprachmerkmale, Anweisungstypen und entsprechenden Anweisungen eines Testfalls abgedeckt sind. Dazu muss ein geeignetes Zustandsübergangsdiagramm, wie z.B. Harel-Zustandsdiagramme, gefunden werden oder die fehlenden Sprachmerkmale können durch Erweiterungen ergänzt werden.

Der gewählte Ansatz basiert auf der Verwendung von SCXML [W3C 2018] als Datenformat des Informationsmodells. Es ist ein Standard des World Wide Web Konsortium (W3C) zur Darstellung von Zustandsübergangsdiagrammen, welche Harel-Zustandsdiagrammen in einem XML-basierten Dateiformat abbilden können. Die Integration von Harel-Zustandsdiagrammen ermöglicht die transparente Darstellung von Zuständen, zusammengesetzten Zuständen und parallelen Zuständen. Darüber hinaus unterstützt es die Erweiterbarkeit der Sprachmerkmale durch eine boolesche Ausdruckssprache, Ortsausdruckssprache, Wertausdruckssprache und Skriptsprache. Die Abbildung der erforderlichen testfallbezogenen Anweisungen, welche im Kapitel 3 vorgestellt wurde, ist in Tabelle 19 dargestellt und wurde in dem Forschungsprojekt STEVE [BIBA - Bremer Institut für Produktion und Logistik 2018] abgeleitet.

Tabelle 19: Repräsentation von Testfällen in SCXML [Franke und Thoben 2022]

Kriterium	Abdeckung durch SCXML	Muss durch die Erweiterung von SCXML abgedeckt werden
Kriterien für Datentypen		
Verkapselung		X
Sichtbarkeit		X
Primitive Datentypen		X
Komplexe Datentypen		X
Wertebereiche für Datentypen		X
Kriterien für Transitionen		
Arithmetische Ausdrücke		X
Temporale Bedingungen	X	
Ereignis basierte Bedingungen	X	
boolesche Bedingung		X
Sequenzen	X	
Parallele Sequenzen	X	
Quantoren	X	
Kriterien für globale Bedingungen		
Verkapselung		X
Arithmetische Ausdrücke	X	
Temporale Bedingungen	X	
Zyklische Bedingungen	X	
Prädikatenlogik		X
Test spezifische Funktionen		
Abbildung von Anforderungen		X
Protokollierung	X	
Einspeisung von Fehlern		X

Tabelle 19 zeigt in der Spalte Abdeckung durch SCXML, dass die Sprachmerkmale von SCXML nicht ausreichen, um einen Testfall darzustellen. Der verfügbare Erweiterungsmechanismus soll die fehlenden Sprachmerkmale hinzufügen. Die Spalte „Muss durch Erweiterung abgedeckt werden“ fasst die benötigen Sprachmerkmale

zusammen. Die Anpassung und Erweiterung von SCXML für die Darstellung von SCXML resultierte in *Generic SCXML*.

Generic SCXML ist das entwickelte Informationsmodell zur Repräsentation von interoperablen Testfällen für Funktionstest, welche für Hardware-in-the-Loop Tests implementiert werden. Generic SCXML basiert auf SCXML und erweitert dieses um testspezifische Inhalte. Der folgende Abschnitt beschreibt die spezifische Erweiterung für jedes Sprachmerkmal im Detail.

5.1.1.1 Datentypen

Die vorgeschlagene Erweiterung erfordert die Möglichkeit, sowohl primitive als auch komplexe Datentypen darzustellen. Außerdem ist eine Kapselung erforderlich, um objektorientierte Daten darstellen zu können. Zu diesem Zweck werden alle Daten (<data>) als JSON-String dargestellt. Hierzu wird das Attribut *expr* verwendet. Jeder JSON-String enthält Attribute, die die Signale des SUT und lokale Variablen einschließlich Typ und aktuellem Wert beschreiben. Die obligatorischen JSON-Attribute werden verwendet, um den Datentyp selbst zu beschreiben. Die Definition, ob es sich um einen primitiven oder komplexen Datentyp handelt, ist obligatorisch. Der verwendete Datentyp, die Einheit und ihr ursprünglicher Name sind obligatorische Attribute, die die korrekte Interpretation unterstützen.

Die Spezifikation des JSON-Strings hat sich durch die kontinuierliche Evaluation in den Forschungsprojekten STEVE und AGILE-VT angepasst. Die aktuelle Version ist in der Version *Generic SCXML* Issue 5.1 aus dem Jahr 2021 definiert und ist in der Abbildung 33 gezeigt.

```
1 {
2     "name": "1OxygenConsumption_Value",
3     "type": "Float",
4     "unit": "None",
5     "value": 0
6 }
```

Abbildung 33: Beispiel für die Definition einer Variable in JSON

Das JSON-Attribut *name* definiert den Signalnamen im System. Der Zugriff auf das Signal erfolgt im Testsystem über diesen Namen. Der Name wird bei der Konfiguration für ein spezifisches Testsystem angepasst. Der Datentyp der Variable ist in dem JSON-Attribut *type* kodiert und umfasst alle primitiven Datentypen. Somit erlaubt Datentyp die eindeutige Interpretation eines Signals im Testsystem. Es werden aber zusätzliche Metainformationen benötigt. Zusätzlich zum Datentyp muss die Einheit über das JSON-Attribut *unit* definiert werden. Hierüber werden übliche Fehler in den Definitionen des Testfalls vermieden. Gängige Fehler sind beispielsweise

die fehlerhafte Angabe zum Winkel oder der Zeit. Die JSON-Struktur kann zukünftig durch neue JSON-Attribute ergänzt werden, ohne die Abwärtskompatibilität zu verlieren.

5.1.1.2 Arithmetische Ausdrücke

Die Erweiterung um arithmetische Terme und boolesche Bedingungen hängt von der Sprache ab, die als Erweiterung zu SCXML gewählt wird. Geeignete Sprachen wie General Expression Language oder ECMAScript sind anwendbar. Dies führt zu entsprechenden Einschränkungen bei der Formulierung von Stimuli oder Verdikten. Zudem sind nicht alle Sprachen für alle Anwendungsszenarien geeignet. Beispielsweise ist die Definition eines Integrals für ein Verdikt nicht möglich, wenn der Testfall die harte Echtzeitbeschränkung erfüllen soll. Ähnliche Beispiele gibt es für andere Anwendungsdomänen. Im Rahmen des *Generic SCXML* Issue 5.1 wurde im Kontext von STEVE und AGILE-VT eine eigene Expression Language definiert. Diese deckt Funktionen aus den Gruppen arithmetische Operatoren, logische Operatoren, Vergleichsoperatoren, Operationen auf Zeichenketten, Operationen auf Zeitangaben und mathematische Funktionen ab.

5.1.1.3 Test spezifische Funktionen

Die Erweiterung um testspezifische Funktionen erfordert die Integration von Anforderungsabbildung und Fehlerinjektion. Dazu werden Anweisungen zur gezielten Stimulation und Prüfung mit Anforderungen verknüpft. Diese Ausrichtung ist notwendig, um die automatisierte Auswertung und Berichterstellung im Rahmen der Testautomatisierung zu unterstützen. Daher können die testspezifischen Funktionen als Teil der Parameterdefinition mit einer Anforderungs-ID versehen werden. Die Unterstützung von Fehlerinjektionen und anderen testsystemspezifischen Methoden kann über vordefinierte Aufrufe (<invoke>) abgebildet werden. Nicht alle testspezifischen Funktionen sind über <invoke> zugänglich. In den Forschungsprojekten STEVE [BIBA - Bremer Institut für Produktion und Logistik 2018] & AGILE-VT [dSPACE GmbH 2021] wurde eine Teilmenge von gängigen testspezifischen Funktionen ausgewählt, um als nativ implementierte Menge unterstützt zu werden. Dieses Set heißt TASCXML-Befehlssatz und wurde von Airbus und den führenden Testsystemherstellern dSPACE, Vector und TechSAT erarbeitet.

Tabelle 20: Übersicht über den TASCXML-Befehlssatz

ID	TASCXML-Befehlssatz	Beschreibung	Parameter

1	<tascxml:set>	Setzt ein Signal auf einen bestimmten Wert	id, dataid, value, valueexpression
2	<tascxml:get>	Liest den aktuellen Wert eines Signals	id, dataid,
3	<tascxml:result>	Protokolliert ein Ergebnis (Urteil) des Testfalls	id, verdict, timestamp, teststep, description, condition, resulted, requirement, currentvalue, valuemapping
4	<tascxml:ramp>	Löst eine Rampe aus, die ein Signal für eine bestimmte Dauer ändert	id, dataid, slope, startexpr, endcond, duration
5	<tascxml:sine>	Erzeugt eine Kurvenform einer Sinuskurve	id, dataid, frequency, duration, amplitude, phase, startexpr
6	<tascxml:sawtooth>	Erzeugt eine Wellenform eines Sägezahns	id, dataid
7	<tascxml:pulse>	Erzeugt eine Wellenform eines Pulses	id, dataid, duration, slope, phase, startexpr
8	<tascxml:verifytolerance>	Überprüft, ob der Wert eines Signals innerhalb eines bestimmten Toleranzbereichs liegt	id, dataid, expected, duration, upper, lower

5.1.1.4 Grundstruktur und Metainformationen

Die Grundstruktur eines Testfalls in SCXML muss spezifiziert werden, um austauschbare Testfälle zu erhalten. Das Forschungsprojekt STEVE & AGI-LE-VT hat hierzu Best Practices erarbeitet. Die aktuelle Version von *Generic SCXML* (Version 5.2) definiert die Zustände *StartUP*, *Logic* und *TearDown* als obligatorische Zustände für Testobjekte (TA). Ein Testobjekt kann ein Testfall oder ein Monitor sein. Die Grundstruktur eines Testfalls ist in Abbildung 34 dargestellt.

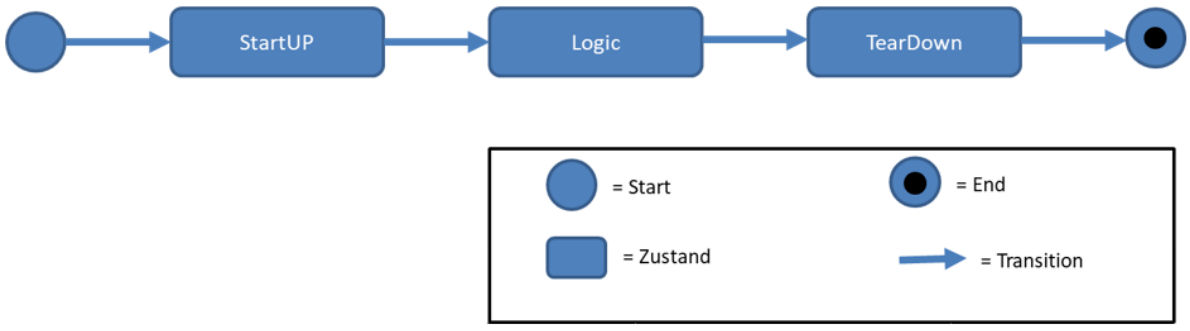

Abbildung 34: Gleiche Grundstruktur eines Testfalls [Franke et al. 2023]

Testfälle in u. a. in CCDL, Python und RTT, definieren Testschritte, um den logischen Testablauf zu strukturieren. Diese Information ist erforderlich und wird normalerweise in den Testberichten hinzugefügt. Zu diesem Zweck wurde SCXML um

eine Reihe von vordefinierten Ereignissen (<send>) erweitert, um Metainformationen wie Testschritte hinzuzufügen. Eine Übersicht der derzeit unterstützten Metainformationen in Form von Ereignissen ist in Tabelle 21dargestellt.

Tabelle 21: Metainformationen für Testschritte in einem SCXML-Testfall

Ereignis als SCXML Event	**Payload in JSON**
testcase.enter.teststep.*	{ "label": "<label>", "stateid": "<stateid>", "teststepdescription": "<teststepdescription>", "timestamp": "<timestamp>" }
testcase.enter.state.*	{ "stateid": "<stateid>", "timestamp": "<timestamp>" }
testcase.enter.{startup, teardown}.*	{ "timestamp": "<timestamp>" }
testcase.done.{startup, teardown}.*	{ "timestamp": "<timestamp>" }
testcase.done.{state, teststep}	{ "stateid": "<stateid>", "timestamp": "<timestamp>" }

Die Integration eines Ereignisses erfolgt in drei Schritten. Im ersten Schritt wird das Ereignis im Datenmodell als eine interne Variable mittels <data> angelegt. Anschließend werden in dem entsprechenden SCXML Zustand die spezifischen Para-

meterwerte mittels <assign> gesetzt. Hierzu zählt beispielsweise der aktuelle Zeitstempel. Abschließend wird im dritten Schritt das Ereignis mithilfe der SCXML Anweisung <send> gesendet. Ein Beispiel für den ersten Schritt ist in Abbildung 35 und ein Beispiel für den zweiten und dritten Schritt ist in Abbildung 36 gezeigt.

```
<data expr="{&quot;timestamp&quot;:&quot;NA&quot;}" id=" eventEnterstartup"/>
```

Abbildung 35: Beispiel für ein Testschritt-Ereignis (Schritt 1)

```
<state id="startup">
    <onentry>
        <assign expr="now()" location="_eventEnterstartup.timestamp"/>
        <send delay="0ms" event="testcase.enter.startup.Step11_OxyCrew_TASCXML_V3"
            id="testcase-enter-startup-Step11_OxyCrew_TASCXML_V3"
            namelist="_eventEnterstartup"
            target="#_external"
            type="#testautomation"/>
    </onentry>
```

Abbildung 36: Beispiel für ein Testschritt-Ereignis (Schritte 2 & 3)

Eine weitere Besonderheit sind Timeouts in Testfällen. Hier definiert ein Timeout, dass die Ausführung eines Testschrittes oder einer Anweisung eine vordefinierte Zeit nicht überschreiten darf. Die Modellierung von Timeouts erfolgt über Transitionen, die mit Ereignissen versehen sind und den Zustand *TearDown* als Ziel haben. Beim Eintreten in einen Zustand wird das Senden des Ereignisses entsprechend der definierten Zeit veranlasst. Beim Verlassen des Zustands wird das Ereignis abgebrochen, um das Timeout nicht auszulösen.

5.1.2 Transformation eines Testfalls in das Informationsmodell

Das Ziel der Transformation ist die Übersetzung eines Testfalls von einer *Testskriptsprache* nach *Generic SCXML*. Dabei müssen die spezifischen Eigenschaften der *Testskriptsprache* berücksichtigt und im Rahmen der Transformation entfernt werden. Der gewählte Ansatz verfolgt die Reduzierung des Testfallinhalts auf die Informationen aus den Gruppen *Stimulus, Laufzeitverhalten* und *Verdikts* und den anschließenden Aufbau eines neuen Testfalls in *Generic SCXML*. Im Folgenden werden die Transformationsschritte im Detail vorgestellt.

Die Übersetzung eines Testfalls basiert auf den üblichen Schritten der syntaktischen und anschließenden semantischen Verarbeitung im Compilerbau. Während die syntaktische Verarbeitung auf Parsebäumen basiert, wurde die semantische Verarbeitung gezielt in Richtung einer zustandsbasierten Perspektive ausgerichtet. Der Übersetzungsprozess von einem Testfall in einen SCXML-Testfall ist in Abbildung 37 dargestellt.

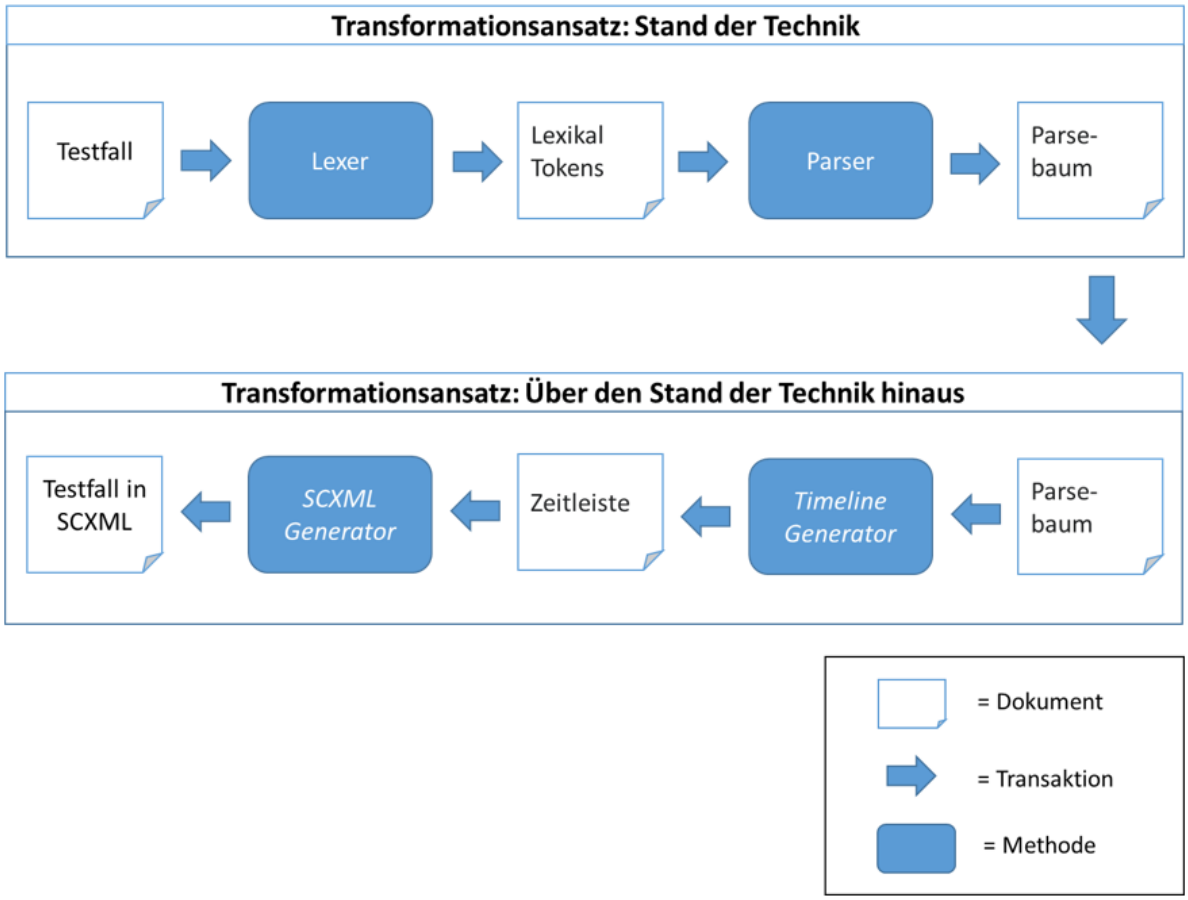

Abbildung 37: Transkompilierung von Testfällen in SCXML-Testfälle

Für jede *Testskriptsprache* ist ein spezifischer Lexer und Parser erforderlich. Die Definition und Erstellung dieser Werkzeuge ist Stand der Technik und funktioniert für alle *Testskriptsprachen* ähnlich. Das Ergebnis der Ausführung eines Parsers ist ein Parsebaum, der die Testfallstruktur und den Testablauf als Baum modelliert. Dieser Baum hängt von den Eigenschaften der *Testskriptsprache* ab.

Im Rahmen der Prototypenentwicklung wurden Lexer, Parser, *Timeline Generator* und *SCXML Generator* für die *Testskriptsprachen* CCDL und Python entwickelt. Die Lexer und Parser für CCDL und Python wurden mit Hilfe der Software ANTLR und selbst entwickelten Grammatiken für beide Sprachen generiert. Die Entwicklung des *Timeline Generators* und des *SCXML Generators* basiert auf keinen Vorarbeiten. Im Folgenden wird detailliert beschrieben, wie die Interoperabilität anhand des Prototyps erreicht wurde.

Der Harmonisierungsschritt zur Erreichung der Interoperabilität erfolgt im *Timeline Generator*. Der *Timeline Generator* ist spezifisch für eine *Testskriptsprache* und er-

zeugt aus dem Parse-Baum eine Folge von Anweisungen. Die unterstützten Anweisungen ergeben sich durch die Informationsgruppen *Stimulus* und *Verdikt*. Die Anweisungssequenz ist eine Zeitleiste, die definiert, wann jede Anweisung ausgelöst werden soll.

Die Zeitleiste definiert nicht, wie lange die Anweisung ausgeführt wird, sondern sie definiert, ob die Anweisung blockiert oder nicht blockiert. Basierend auf der Timeline erstellt der *SCXML Generator* zunächst die gesamte Testfallstruktur (*StartUP*, *Logic*, *TearDown*) und fügt anschließend alle Anweisungen in chronologischer Reihenfolge hinzu. Dabei werden Anweisungen als Aktionen innerhalb der vordefinierten Zustände hinzugefügt. Parallele Anweisungen können hinzugefügt werden, indem ein paralleler Zustand erstellt wird.

Im Folgenden wird eine exemplarische Übersetzung von einem Testfall in CCDL nach SCXML anhand von Screenshots des Prototyps gezeigt. Die Abbildung 38 zeigt einen Testfall, indem drei Signale in dem Testschritt *Initial Condition* nacheinander stimuliert werden. Hierbei dauert das Einstellen der Flugphase 9 Zeit beim SUT, weswegen die Testfallausführung in ihrer Stimulierung pausiert wird. Anschließend wird die Geschwindigkeit gesetzt und wiederum die Testfallausführung pausiert.

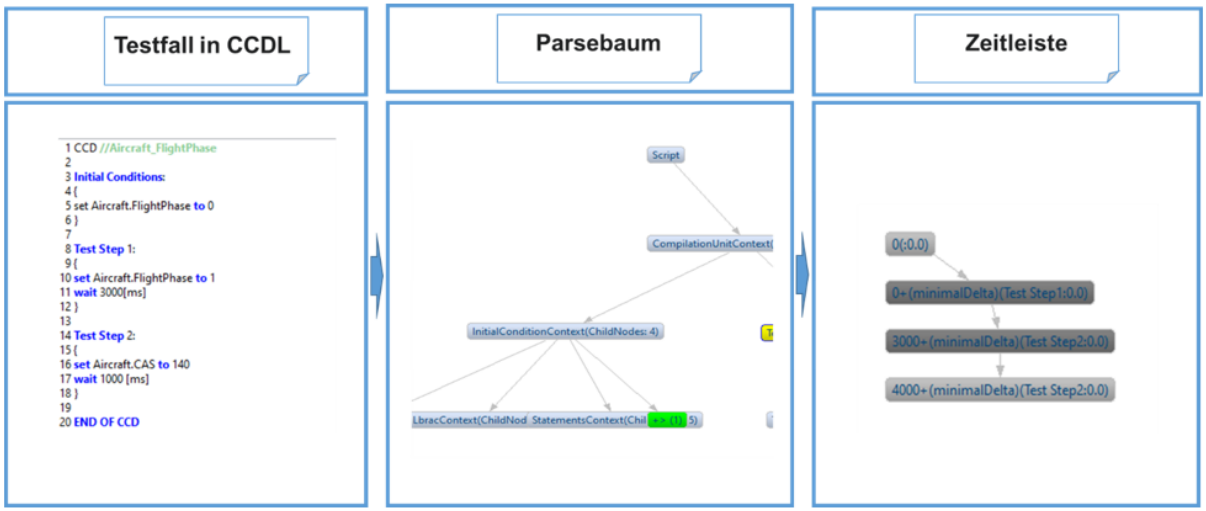

Abbildung 38: Beispiel für die Zwischenergebnisse der Transkompilierung von einem Testfall in CCDL in einen SCXML-Testfall

Basierend auf dem Testfall wird ein Parsebaum generiert, welcher als Screenshot des Prototyps in Abbildung 38 gezeigt ist. Die generierten Knoten des Baums entsprechen der Nichtterminalsymbolen aus der CCDL-Grammatik. Hierbei ist das Startsymbol der *CompilationKomponenteContext* Knoten. Die in dem Testfall gezeigte Stimulierung befindet sich in dem linken Zweig des Baumes als *InitialConditionContext* inklusive ihrer Anweisungen als *StatementContext*.

Der entwickelte *Timeline Generator* für CCDL traversiert den Baum. Für jeden Testschritt werden die Anweisungen in die Informationsgruppen eingeteilt. Für jede Anweisung wird dann ein Zeitstempel erzeugt. Dementsprechend sind in der Zeitleiste (siehe Abbildung 38) Zeitstempel von +3000 und +1000 enthalten.

Abschließend wird die Zeitleiste aus der Abbildung 38 in den *SCXML Generator* eingespeist und basierend darauf ein SCXML Modell generiert. Ein Ausschnitt des generierten Testfalls in *Generic SCXML*, der das Signal Aircraft_Flightphase auf 1 setzt und anschließend drei Sekunden wartet, ist in der Abbildung 39 gegeben.

```
47  <state id="logic_c0_c0"><!--   node-size-and-position x=20 y=43 w=310 h=452  -->
48   <onentry>
49    <assign expr="Date.now() * 1000000" location="_eventEnterTS1.timestamp"></assign>
50    <send delay="0ms" event="testcase.enter.teststep.Aircraft_FlightPhase" id="testcase-enter-teststep-
51   </onentry>
52   <onexit>
53    <assign expr="Date.now() * 1000000" location="_eventDoneTS1.timestamp"></assign>
54    <send delay="0ms" event="testcase.done.teststep.Aircraft_FlightPhase" id="testcase-done-teststep-Ai
55   </onexit>
56   <state id="logic_c0_c0_c0"><!--   node-size-and-position x=20 y=43 w=270 h=264  -->
57    <onexit>
58     <send delay="3000ms" event="logic_c0_c0_s4_wait_event" id="logic_c0_c0_s4_wait_event"></send>
59    </onexit>
60    <transition event="done.state.logic_c0_c0_c0_p0" target="logic_c0_c0_s1"></transition>
61    <parallel id="logic_c0_c0_c0_p0"><!--   node-size-and-position x=20 y=43 w=230 h=201  -->
62     <state id="logic_c0_c0_c0_p0_c0"><!--   node-size-and-position x=20 y=43 w=190 h=138  -->
63      <final id="logic_c0_c0_c0_p0_c0_s0"><!--   node-size-and-position x=20 y=43 w=150 h=75  -->
64       <onentry>
65        <tascxml:set dataid="Aircraft_FlightPhase" value="1"></tascxml:set>
66       </onentry>
67      </final>
68     </state>
69    </parallel>
70   </state>
71   <state id="logic_c0_c0_s1"><!--   node-size-and-position x=110 y=357 w=90 h=75  -->
72    <transition event="logic_c0_c0_s4_wait_event" target="logic_c0_c1"></transition>
73   </state>
74  </state>
75  <state id="logic_c0_c1"><!--   node-size-and-position x=20 y=545 w=310 h=452  -->
76   <onentry>
77    <assign expr="Date.now() * 1000000" location="_eventEnterTS2.timestamp"></assign>
78    <send delay="0ms" event="testcase.enter.teststep.Aircraft_FlightPhase" id="testcase-enter-teststep-
79   </onentry>
80   <onexit>
81    <assign expr="Date.now() * 1000000" location="_eventDoneTS2.timestamp"></assign>
82    <send delay="0ms" event="testcase.done.teststep.Aircraft_FlightPhase" id="testcase-done-teststep-Ai
83   </onexit>
84   <state id="logic_c0_c1_c0"><!--   node-size-and-position x=20 y=43 w=270 h=264  -->
85    <onexit>
86     <send delay="10000ms" event="logic_c0_c1_s4_wait_event" id="logic_c0_c1_s4_wait_event"></send>
87    </onexit>
88    <transition event="done.state.logic_c0_c1_c0_p0" target="logic_c0_c1_s1"></transition>
```

Abbildung 39: Ausschnitt des generierten SCXML-Testfalls

5.2 Schritt 2 der kollaborativen Fehlersuche: Adaptive Fehlersuche auf Testdaten ausführen

Der **Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen** der Fehlersuche konzentriert sich auf die Entwicklung von drei Fehlersuchmethoden. Alle Fehlersuchmethoden versuchen, Unterschiede zwischen dem fehlgeschlagenen Testfall des Testprozesses für das Gesamtsystem (OEM) und dem vergleichbaren Testfall des Testprozesses für das System/Modul (Lieferant Tier-1) zu finden. Im Folgenden werden die drei Fehlersuchmethoden detailliert vorgestellt.

5.2.1 Fehlersuchmethode 1: Ausführung des Testfalls und Analyse des Signalverlaufs

Die Fehlersuchmethode wird von zwei Testprozessen begleitet, wobei ein Testprozess die höhere Integrationsstufe und der andere Testprozess die niedrigere Integrationsstufe darstellt. Für die folgenden Ausführungen wird davon ausgegangen, dass der Testprozess des OEM der Testprozess mit der höheren Integrationsstufe ist und der Testprozess des Modul-Lieferanten der Testprozess mit der niedrigeren Integrationsstufe ist. Die Ausführung des Testfalls vom OEM ist nicht direkt beim Modul-Lieferanten möglich, da die Testprozesse unterschiedliche *Testskriptsprachen* verwenden. Im Rahmen des **Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche wurde der Testfall bereits in ein Informationsmodell überführt. Dieses Informationsmodell entspricht dem Testfall in SCXML. Der Testfall in SCXML kann nicht auf dem Testsystem des Modul-Lieferanten ausgeführt werden. Dazu ist es notwendig, den Testfall aus SCXML zurück in die verwendete *Testskriptsprache* des Modul-Lieferanten zu übersetzen. Anschließend können beide Testfälle ausgeführt und die Ergebnisse in Form von Signalverläufen, Logs und Testberichten analysiert werden. Der fehlende Lösungsbaustein ist die Übersetzung eines Testfalls von *Generic SCXML* in eine *Testskriptsprache*. Im Folgenden wird die entwickelte Lösung zur Übersetzung eines Testfalls von SCXML in eine *Testskriptsprache* im Detail vorgestellt.

5.2.1.1 Transformation eines Testfalls von SCXML in eine Testskriptsprache

Die Übersetzung eines Testfalls von SCXML in eine *Testskriptsprache* basiert auf einem sprachunabhängigen und einem sprachspezifischen Teil. Dies ermöglicht die Transformation eines Testfalls von SCXML in eine beliebige *Testskriptsprache*. Die aktuelle Version des *Test Case Translator* erlaubt die Transformation nach CCDL und RTT. Eine Übersicht über die Transformationsmethode ist in Abbildung 40 gegeben.

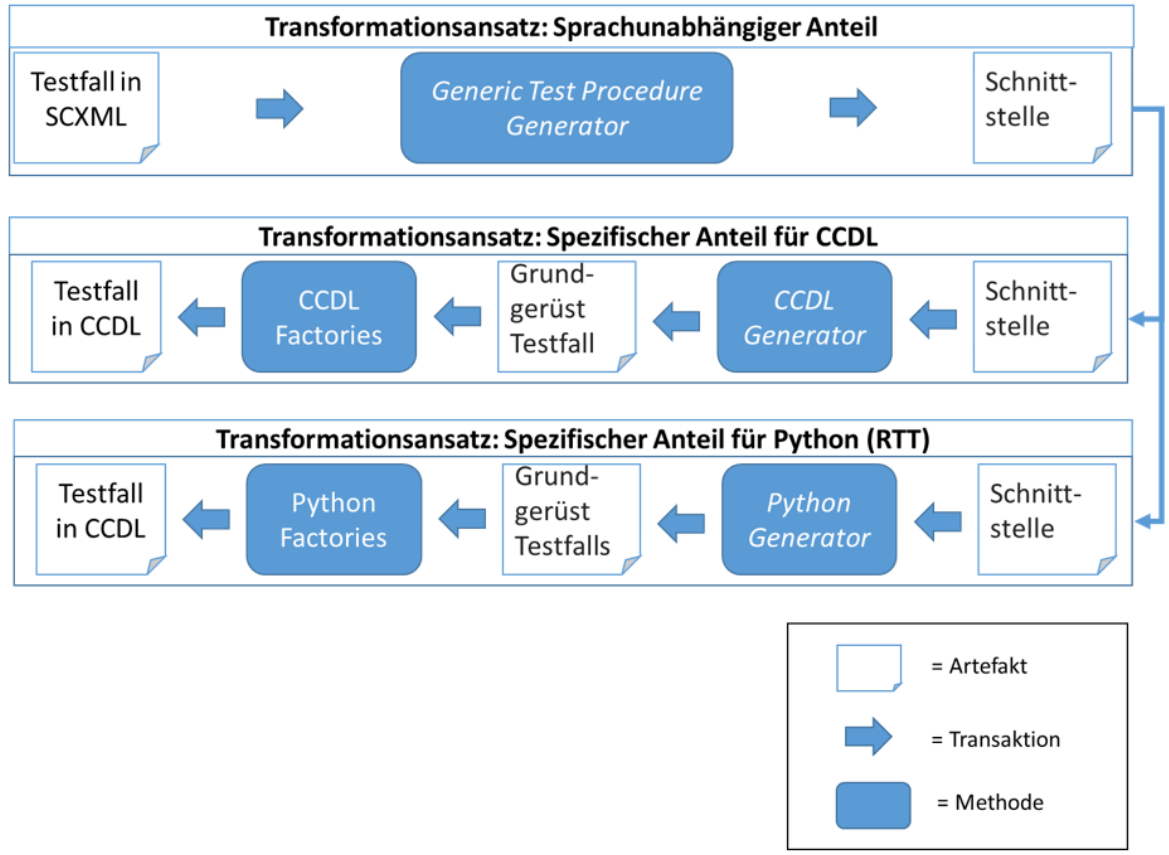

Abbildung 40: Transformation eines SCXML-Testfalls in eine *Testskriptsprache*

Die Transformationsmethode ermöglicht im sprachunabhängigen Teil die Traversierung des SCXML Modells in Richtung Testausführung. Dementsprechend startet der *Generic Test Procedure Generator* die Traversierung im Zustand *StartUP* und beendet die Transformation im Zustand *TearDown*. In jedem Zustand sammelt der *Generic Test Procedure Generator* Informationen darüber, welche Variablen und Klassen deklariert und initialisiert werden müssen. Diese Informationen übergibt er an den sprachspezifischen Generator. Zusätzlich ruft der *Generic Test Procedure Generator* während der Traversierung für jede gefundene Anweisung eine spezifische Factory auf. Die anweisungsspezifischen Factories werden vom sprachspezifischen Generator implementiert und vom *Generic Test Procedure Generator* aufgerufen. Dabei kann der sprachspezifische Generator Besonderheiten bezüglich des Programmierparadigmas oder der Typsicherheit berücksichtigen.

Die Transformation des exemplarischen SCXML-Testfalls nach RTT (Python) und CCDL ist mit den Prototypen möglich und führt zu den in Abbildung 41 und Abbildung 42 gezeigten Ergebnissen.

```
from testcaselib import *
import math

AircraftCAS=InitVariable("SIM_OXY_CP::IN.CST_R1StowageboxOpen_Request", "AircraftCAS", float)
Aircraft_FlightPhase=InitVariable("Aircraft_FlightPhase", "Aircraft_FlightPhase", float)

def MainGenerator():
        yield TestrunGenerator(InitialConditions, [TestSteps1, TestSteps2], teardown,
        Finally, TestCaseName = "Aircraft_FlightPhase")
def TestSteps1():
        event1 = TestCaseEventEnterTeststep("TS1","TS1", testStepDescription="TS1:
        Automatically generated test step based on data recordings")
        event1.send()
        Aircraft_FlightPhase.Value=1
        yield wait(3.0)
        event1 = TestCaseEventDoneTeststep("testcase-done-teststep-aircraft_flightphase","TS1")
        event1.send()
def TestSteps2():
        event1 = TestCaseEventEnterTeststep("NA","NA", testStepDescription="NA")
        event1.send()
        Aircraft.CAS.Value=140
        yield wait(10.0)
        event1 = TestCaseEventDoneTeststep("testcase-done-teststep-aircraft_flightphase","NA")
        event1.send()
def InitialConditions():
        event1 = TestCaseEventEnterStartup()
        event1.send()
        Aircraft_FlightPhase.Value=0
        event1 = TestCaseEventDoneStartup()
        event1.send()
def teardown():
        event1 = TestCaseEventEnterTeardown()
        event1.send()
def TestSteps():
        pass
def Finally():
        event1 = TestCaseEventDoneTeardown()
        event1.send()
        pass
```

Abbildung 41: Transformationsergebnis des SCXML-Testfalls in der *Testskriptsprache* RTT

```
CCD //Aircraft_FlightPhase

Initial Conditions:
{
set Aircraft_FlightPhase to 0
}

Test Step 1:
{
set Aircraft_FlightPhase to 1
wait 3000[ms]
}

Test Step 2:
{
set Aircraft.CAS to 140
wait 10000[ms]
}

END OF CCD
```

Abbildung 42: Transformationsergebnis des SCXML-Testfalls in der *Testskriptsprache* CCDL

Beide Testfälle haben die gleiche Semantik, aber eine unterschiedliche Syntax. Besonders deutlich wird dies bei der Verwendung von Signalen und Testschritten. Während in CCDL die Signale als Variablen und Testschritte als Blöcke direkt verwendet werden können, müssen diese in RTT erst global deklariert und instanziiert werden. Diese Unterschiede können nur von den spezifischen Generatoren umgesetzt werden. Dazu benötigen sie die notwendigen Informationen und den Zugriff auf den Header, den Body und das Ende eines Testfalls. Die notwendigen Informationen, wie z.B. der Datentyp, sind im Datenmodell enthalten und werden an die spezifischen Generatoren weitergegeben. Der Zugriff auf die verschiedenen Elemente eines Testfalls erfolgt über die Methoden einer Schnittstelle. Damit können nicht nur Variablen über den Testfall verteilt deklariert, instanziiert und verwendet werden, sondern auch Klassen inklusive ihrer Konstruktoren und Destruktoren generiert werden.

5.2.2 Fehlersuchmethode 2: Mustersuche

Die folgenden Unterkapitel wurden aus der Veröffentlichung [Franke et al. 2023] übernommen und stellenweise angepasst.

Die Fehlersuchmethode wird von zwei Testprozessen begleitet, wobei ein Testprozess die höhere Integrationsstufe und der andere Testprozess die niedrigere Integrationsstufe darstellt. Für die folgenden Ausführungen wird davon ausgegangen, dass der Testprozess des OEM der Testprozess mit der höheren Integrationsstufe ist und der Testprozess des Modul-Lieferanten der Testprozess mit der niedrigeren Integrationsstufe ist.

Im Rahmen des 1. Schrittes der Fehlersuche wurde der Testfall vom OEM bereits in ein Informationsmodell überführt. Dieses Informationsmodell entspricht dem Testfall in SCXML. Zuerst wird der Testfall des Modul-Lieferanten auch in SCXML übersetzt. Anschließend wird ein Testfall in ein Muster überführt und der andere Testfall wird in einem Graph G überführt. Schlussendlich wird eine Suche für unbekannte Muster mittels Clique durchgeführt. Der fehlende Lösungsbaustein ist die Aggregation von Testfällen und eine darauf aufbauende Mustersuche. Die hierfür benötigte Anfragesprache für Muster und der benötigte Graph G werden im Folgenden hergeleitet und ihre Spezifikation und Implementierung gezeigt.

Die verfügbaren SCXML-Testfälle sind Zustandsübergangsdiagramme und unabhängig von ihrer ursprünglichen *Testskriptsprache*. Damit ist für diese Testfälle bereits Interoperabilität auf der Daten-/Informationsschicht erreicht. Damit ist die Voraussetzung für die Aggregation und anschließende Mustersuche von Testfällen in einem Knowledge Graph prinzipiell erfüllt. Aus technischer Sicht benötigt die Aggregation ein Repository und eine Datenbank. So kann ein Testfall in SCXML als Large Binary Object (BLOB) in SQL-Datenbanken oder nativ als parsbares XML-Dokument in eine NoSQL-Datenbank hochgeladen werden. Die Schwäche eines BLOBs ist, dass es wie eine Blackbox funktioniert und keine Informationen über die Abfragesprache der Datenbank abgefragt werden können. Eine SQL-Datenbank ist daher keine praktikable Lösung. Die XML-zentrierte Abfragesprache einer NoSQL-Datenbank verwendet teilweise die XML Path Language (XPath) oder XQuery als Basis.

Im Folgenden werden die Nachteile der Verwendung von XPath (die Ergebnisse gelten auch für XQuery) zur Implementierung einer facettierten und explorativen Suche dargestellt. XPath definiert Suchanfragen über die XML-Baumstruktur. Die Schwäche bei der Verwendung der Baumstruktur zur Abfrage bestimmter Elemente in Testfällen besteht darin, dass Testfälle unterschiedliche Baumstrukturen haben. Dieses Problem wird anhand eines abstrakten Testfalls in SCXML beschrieben. Alle Testfälle in einer NoSQL-Datenbank haben die gleiche hierarchische Struktur, die durch eine Folge von zusammengesetzten Zuständen definiert ist: *StartUP*, *Logic* und *TearDown*. Diese Struktur ermöglicht es bereits, in allen Testfällen die gleiche Abfrage für die zusammengesetzten Zustände der obersten Ebene zu definieren, was notwendig ist, um Ketten zu erstellen und generische Muster zu finden. Ein Beispiel für die Abfrage von *StartUP* in allen Testfällen wäre *scxml/state[@id ='StartUP']*

in der XPath Notation. In keinem der zusammengesetzten Knoten ist ein Entwurfsmuster spezifiziert. Dies hat zur Folge, dass die Reihenfolge der XML-Elemente innerhalb der zusammengesetzten Zustände von Testfall zu Testfall unterschiedlich ist. Dementsprechend kann der Testingenieur selbst entscheiden, wie viele Zustände er für die Stimulation des Flugzeugs verwenden möchte. Abbildung 43 zeigt zwei mögliche Entwurfsmuster, wie die Stimulation in einem SCXML-Testfall aussehen könnte.

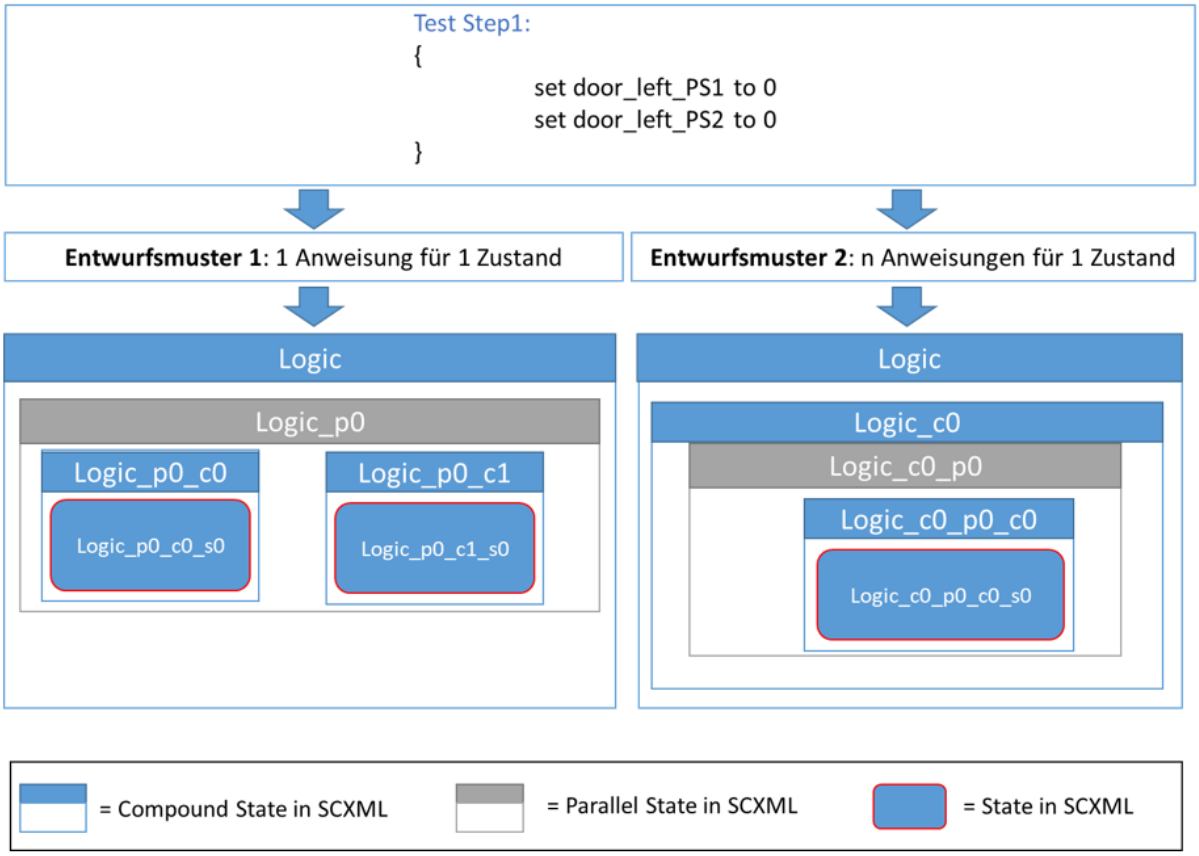

Abbildung 43: Beispiele für die Anwendung unterschiedlicher Entwurfsmuster für SCXML-Testfälle

Das Entwurfsmuster 1 besteht darin, dass in jedem Zustand nur ein Signal mit einem neuen Wert stimuliert wird. Daher hat das Beispiel für Entwurfsmuster 1 zwei Zustände {*Logic_p0_c0_s0, Log-ic_p0_c1_s0*}.

Im Gegensatz dazu empfiehlt das Entwurfsmuster 2, alle Anweisungen in einem Zustand zusammenzufassen. Das Beispiel für Entwurfsmuster 2 hat daher einen Zustand {*Logic_c0_p0_c0_s0*}. Beide Entwurfsmuster führen zu einer völlig unterschiedlichen Baumstruktur. Weitere Entwurfsmuster sind verfügbar und erhöhen die

strukturelle Heterogenität. Dies ist kein Problem für die Source-to-Source-Kompilierung, sondern für die Implementierung der geplanten Abfragemethode, für die eine zusätzliche Interoperabilitätsherausforderung gelöst werden muss.

Die Suche als Teil des Anfrageprozesses in Schritt 2 erfordert jedoch eine ähnliche Struktur der SCXML-Testfälle. Der Grund dafür ist, dass Anfragesprachen, die direkt über den XML-Baum definiert werden (z.B. XPath) und Anfragesprachen, die direkt über einen Graphen definiert werden (z.B. Cypher), die Struktur des Testfalls in der Anfrage berücksichtigen. Durch die Verwendung unterschiedlicher Entwurfsmuster ist die Struktur von Testfällen aus unterschiedlichen Quellen heterogen. Dementsprechend muss die Suche nach Testfallinhalten innerhalb der Anbieterpyramide eine zusätzliche Interoperabilitätsherausforderung bewältigen. Die Herausforderung besteht darin, dass die Struktur der Testfälle nicht nur für die Zustände *StartUP*, *Logic* und *TearDown* konsistent sein muss, sondern in allen Zuständen.

Der gewählte Ansatz ist die Entfernung der Zustände aus dem SCXML-Testfall und die Fokussierung auf die Semantik des Testfalls. Die Annahme der Arbeit ist, dass die Entfernung der Zustände weniger komplex ist als die Umstrukturierung aller Testfälle anhand einer Vorlage. Die Entfernung von Zuständen widerspricht dem Grundaufbau eines SCXML Modells und dementsprechend wird ein Informationsmodell benötigt, welches für die Suche optimiert ist. Im Folgenden wird das hierfür benötigte Informationsmodell vorgestellt. Anschließend wird die Transformation eines SCXML-Testfalls in das Informationsmodell und abschließend eine geeignete Anfragesprache vorgestellt.

5.2.2.1 Informationsmodell eines Testfalls als Graph G

Das Ziel des Informationsmodells ist die semantische Beschreibung eines Testfalls, welche sich ausschließlich aus den Informationsgruppen (*Stimulus, Verdikt* und *Laufzeitverhalten*) und den Metainformationen eines Testfalls ergeben. Die Konkretisierung der Inhalte ist wie folgt:

- Die Abbildung der Informationsgruppen *Stimulus* und *Verdikt* wird durch den TASCXML-Befehlssatz bestimmt.
- Die Abbildung der Informationsgruppe *Laufzeitverhalten* wird durch SCXML Anweisungen bestimmt, mit denen die Ausführung eines Testfalls pausiert werden kann.
- Die Abbildung der Metainformationen bezüglich der Testschritte und der Referenz zu einem SCXML Knoten werden durch den Bezeichner eines SCXML Knotens und den in *Generic SCXML* definierten Ereignissen bestimmt.

Der reduzierte Inhalt ermöglicht die Darstellung eines Testfalls als eine Sequenz von Blöcken, in denen jeder Block eine Aufgabe übernimmt. Hierbei kann ein Block für die Stimulierung des SUT stehen oder für die Überprüfung des SUT. Diese Vereinfachung ermöglicht die Definition und anschließende Suche von Mustern. Die Muster können hierbei in den verschiedenen Testschritten und über die Grenzen von Testschritten hinweg auftreten. Die Zuordnung der Muster zu den Testschritten wird mithilfe der Metainformationen ermöglicht. Im Folgenden wird das Informationsmodell basierend auf den oben definierten Informationsgehalt spezifiziert.

Das Informationsmodell wird als Wissensgraph abgebildet (Knowledge Graph). Der Vorteil von Wissensgraphen ist, dass Suchanfragen über Subgraphen definiert werden können, was die Voraussetzung für eine Mustersuche ist. Der notwendige Wissensgraph, repräsentiert alle Testfälle. Für jeden Testfall wird die explorative Suche für die Abfrage von den Informationsgruppen. Das bedeutet, dass sich der Wissensgraph auf die Semantik des Testfalls konzentrieren soll, anstatt auf seine Struktur. Zu diesem Zweck können verschiedene Graph Typen eingesetzt werden, um Knoten, Kanten, Beschriftungen und Eigenschaften zu beschreiben. Im Folgenden wird die Darstellung eines Testfalls als Graph detailliert vorgestellt.

5.2.2.1.1 Knoten und Labels

Jeder TASCXML-Befehl, abgesehen von <tascxml:get>, wird als ein Knoten im Graph dargestellt. Die Unterscheidung zwischen zwei TASCXML-Befehlen wird über Labels realisiert. Für jeden TASCXML-Befehl wird ein eigenes Label definiert. Somit implementiert der vorgeschlagene Graph sieben Etiketten, nämlich $\{'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE','VERIFYTOLERANCE'\}$.

Abgesehen vom TASCXML-Befehlssatz definiert ein typischer Testfall, welche Anweisungen parallel, sequenziell und innerhalb eines vordefinierten Zustands ausgeführt werden. Um die Gruppierung von TASCXML-Befehlen zu einem bestimmten Zustand zu ermöglichen, wird ein zusätzlicher Graph Knoten mit der Bezeichnung $\{'STATE'\}$ definiert.

5.2.2.1.2 Kanten und Labels

Kanten verbinden zwei Knoten in einem Graphen. Im Rahmen eines Testfalls verbindet eine Kante zwei Knoten, wobei der Knoten einen Zustand oder einen TASCXML-Befehl repräsentiert. Die Kante definiert das Timing zwischen den Knoten. Zu diesem Zweck hat eine Kante ein Label. Es beschreibt, ob die Ausführung dieses Knotens parallel, sequenziell oder verzögert ausgeführt werden soll. Im letzteren Fall wird die Eigenschaft benötigt, welches die Dauer definiert. Die Verwendung dieses Labels ist eingeschränkt, um die Schwäche von Testfällen in

SCXML zu vermeiden. Im Folgenden wird die vollständige Definition des Graphen G gegeben.

5.2.2.1.3 Graph Definition

Wir modellieren den Testfall als gerichteten Graphen G= (V, E), in dem die Knoten $v_{(1...i)}$ in V einen Zustand innerhalb des Testfalls bezeichnen; d.h. ein Zustand der Testausführung. Ein Zustand innerhalb der Testfallausführung besteht aus zwei Teilen. Der erste Teil behandelt die Ausführung der Anweisungen zu Beginn des Zustands als Mikrozustände und der folgende Teil behandelt das Ergebnis der Stimulation in Bezug auf das SUT als Makrozustand. Damit kann ein Knoten einen Mikrozustand oder einen Makrozustand repräsentieren, aber nicht beide. Das bedeutet, dass zwei verbundene Knoten mindestens einen SCXML-Zustand definieren, der einen TASCXML-Befehl und den resultierenden SUT-Zustand enthält. Die Trennung zwischen den Mikro- und Makrozuständen wird mit STATE-Labels implementiert. Dabei ist immer ein Satz von Mikrozuständen, die TASCXML-Befehle sind, mit einem Makrozustand verbunden. Die Semantik eines Makrozustands ist die Darstellung eines neuen SUT-Zustands im Rahmen einer Testfallausführung. Beispielsweise sind der Mikrozustandsbefehl zum Schließen der Tür und das Ergebnis der Befehlsausführung bereits im Makrozustand verfügbar. Die erforderlichen Informationen für einen Mikro- und Makrozustand werden innerhalb des Knotens dargestellt. Jeder Knoten v_i ist ein geordnetes Paar $(\{p_i\},l_i)$ aus einer Menge von Eigenschaften in P und einem Label l_i in L1, wobei

$$L1 = \begin{Bmatrix} 'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE','VERIFYTOLERANCE' \\ 'STATE' \end{Bmatrix} \quad (1)$$

Knoten mit dem Label 'STATE' definieren einen Makrozustand, in dem alle verbundenen TASCXML-Befehle gleichzeitig ausgeführt werden sollen. Dabei werden die unterstützten TASCXML-Befehle durch die Labels als {'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE','VERIFYTOLERANCE'} definiert. Eine Eigenschaft $p_i=(n_i,z_i)$ ist selbst ein Paar mit einem String n_i, der der Eigenschaft einen Namen als Schlüssel zuweist, und z_i, der den Wert der Eigenschaft erfasst. Der Typ des Werts deckt die primitiven Datentypen ab, die als Zeichenfolge, Zahl oder boolescher Wert gespeichert werden können. Die Menge der Eigenschaften als $P= \{p_0, ... p_n\}$ definiert jeweils die Menge aller relevanten Eigenschaften für einen Knoten. Ein Beispiel für Eigenschaften für den TASCXML-Befehl TASCXML:set sind p_0 = {"variable", "string"} und p_1 = {"value", "number"}. Mit diesen Eigenschaften weiß der Befehl TASCXML:set, welcher Wert dem Signal zugewiesen werden soll.

Es gibt eine Kante e in E von Knoten v_1 zu Knoten v_2 entweder a), wenn es einen Übergang zwischen zwei definierten Zuständen der Testausführung gibt oder b), wenn ein TASCXML-Befehl innerhalb desselben Zustands der Testausführung ausgeführt wird. Jede Kante e in E umfasst den Startknoten v_1, den Endknoten v_2, ein

Label l_j und eine Eigenschaft p_j, die die Länge der Dauer angibt; d.h. E= (V, V, L2, P2). Da Z ganze Zahlen sind, sind L2 und P2 definiert als

$$L2 = \{'SIMPLE', 'PARRALLEL\_STATEMENT', 'BLOCKING\_STATEMENT', 'WAIT'\} \text{ (2)}$$

$$P2 = \{('duration', Z)\} \text{ (3)}$$

5.2.2.2 Transformation eines Testfalls in den Graph G

Der definierte Graph G und der SCXML-Testfall als Informationsmodell des **Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche repräsentieren beide Testfälle in unterschiedlichen Formaten und für unterschiedliche Zwecke. Während die Source-to-Source-Kompilierung für einen Testfall in SCXML angewendet werden kann, soll der entwickelte Graph G für Such- und Autovervollständigungsfunktionalität eingesetzt werden. Hierfür wird nur eine Teilmenge des SCXML-Testfalls benötigt und als Graph G definiert.

Die Abbildung eines Testfalls aus dem SCXML-Schema und seiner Erweiterung auf generisches SCXML ist möglich. Dazu sind das Traversieren eines SCXML-Testfalls und die Ausführung von zwei Funktionen, nämlich f und f2, erforderlich. Im Folgenden werden beide Funktionen im Detail vorgestellt.

f ist eine nicht-injektive Funktion.

f: x → v, wobei x ein XML-Element des SCXML-Testfalls und v ist in V.

Im Folgenden wird die Zuordnung von x zu y in Tabelle 22 definiert, wobei sich die Zuordnung auf die XML-Elemente als Eingabe beim Durchlaufen des SCXML-Testfalls und V aus dem Diagramm konzentriert. Dabei werden alle Attribute eines XML-Elements als Eigenschaften P im Graph Knoten V als P abgebildet. Eine explizite Auflistung aller Attribute für jedes XML-Element ist nicht gegeben.

Tabelle 22: Abbildung von einem SCXML-Testfall auf Knoten des Graph G

x	v
<xsd:element name="state" type="scxml.state.type"/>	v = {P, 'STATE'}
<xsd:element name="set" type="scxml.state.type"/>	v = {P, 'SET'}
<xsd:element name="result" type="scxml.state.type"/>	v = {P, 'RESULT'}
<xsd:element name="ramp" type="scxml.state.type"/>	v = {P, 'RAMP'}

<xsd:element name="'sine" type="scxml.state.type"/>	v = {P, 'SINE'}
<xsd:element name="sawtooth" type="scxml.state.type"/>	v = {P, 'SAWTOOTH'}
<xsd:element name="pulse" type="scxml.state.type"/>	v = {P, 'PULSE'}
<xsd:element name="verifytolerance" type="scxml.state.type"/>	v = {P, 'VERIFYTOLERANCE'}

Ein Beispiel für die Zuordnung eines SCXML-Elements zum Diagrammknoten v ist in Abbildung 44 gezeigt.

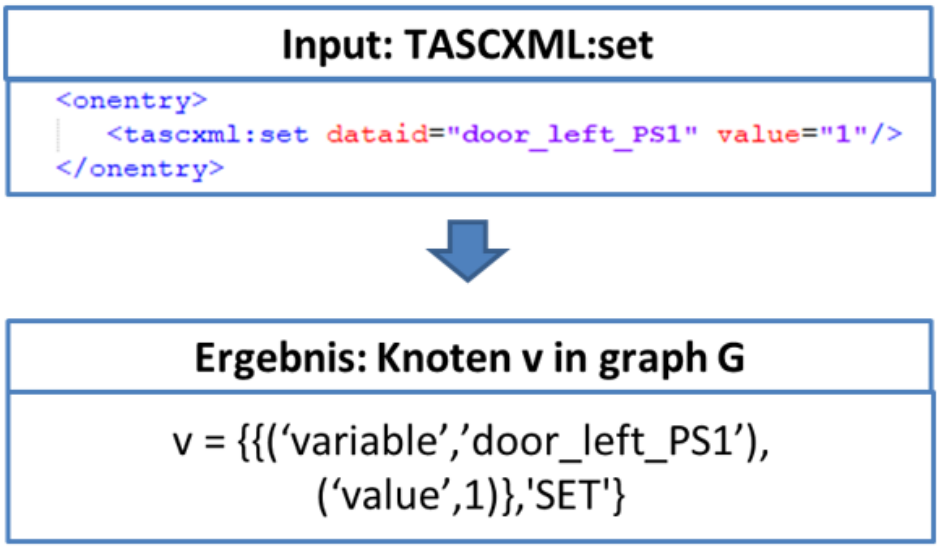

Abbildung 44: Beispiel für die Abbildung des TASCXML-Befehlssatzes auf einem Graph Knoten

Das Traversieren des SCXML-Testfalls erfordert nicht nur eine Abbildung von XML-Elementen auf V des Graphen, sondern auch eine Abbildung von XML-Elementen auf E eines Graphen. Dabei ist die Zuordnung nicht von einem XML-Element zu einem Graph-Knoten definiert; stattdessen wird ein Muster, das über ein paar XML-Elemente beschrieben wird, einem E des Diagramms zugeordnet. Dazu ist die nicht-injektive Funktion f2 erforderlich, die im Folgenden im Detail vorgestellt wird.

f2: x->e wobei x der Satz von XML-Elementen des SCXML-Testfalls und e ist in V. Im Folgenden wird die Abbildung von x auf y in Tabelle 23 definiert.

Tabelle 23: Abbildung von einem SCXML-Testfall auf Kanten des Graph G

x	e
<xsd:group ref="scxml.state.mix" minOccurs="0" max-Occurs="unbounded"/> <xsd:group name="scxml.state.mix"> <xsd:choice> <xsd:element ref="state" minOccurs="1" maxOccurs="unbounded"/> </xsd:choice> </xsd:group>	e = ((P,'STATE'), (P,'STATE'), 'SIMPLE', P)
<xsd:group ref="scxml.state.mix" minOccurs="0" maxOccurs="unbounded"/> <xsd:group name="scxml.state.mix"> <xsd:choice> <xsd:element ref="onentry" minOccurs="1" maxOccurs="1"/> </xsd:choice> </xsd:group>	e = ((P,'STATE'), (P, l ∈ {'SET','RESULT','RAMP','SINE','SAWTOOTH','PULSE','VERIFYTOLERANCE'}), 'PARRALLEL_STATEMENT', P)
<xsd:complexType name="scxml.onentry.type"> <xsd:group ref="scxml.onentry.content"/> <xsd:attributeGroup ref="scxml.onentry.attlist"/> </xsd:complexType> <xsd:group name="scxml.onentry.content"> <xsd:sequence> <xsd:group ref="scxml.core.executablecontent" minOc-curs="0" maxOccurs="unbounded"/> </xsd:sequence> </xsd:group> <xsd:group name="scxml.core.executablecontent"> <xsd:choice> <xsd:group ref="scxml.extra.content" minOccurs="1" maxOccurs="1"/> <xsd:element ref="send"/> </xsd:choice> </xsd:group>	e = (P, ('STATE'), (P, 'STATE'), 'WAIT', ('duration',Z))

Ein Beispiel für die Zuordnung eines TASCXML-Befehls zu einer Kante in Graph G, die das Label PARRALLEL_STATEMENT verwendet, ist in Abbildung 45 gezeigt.

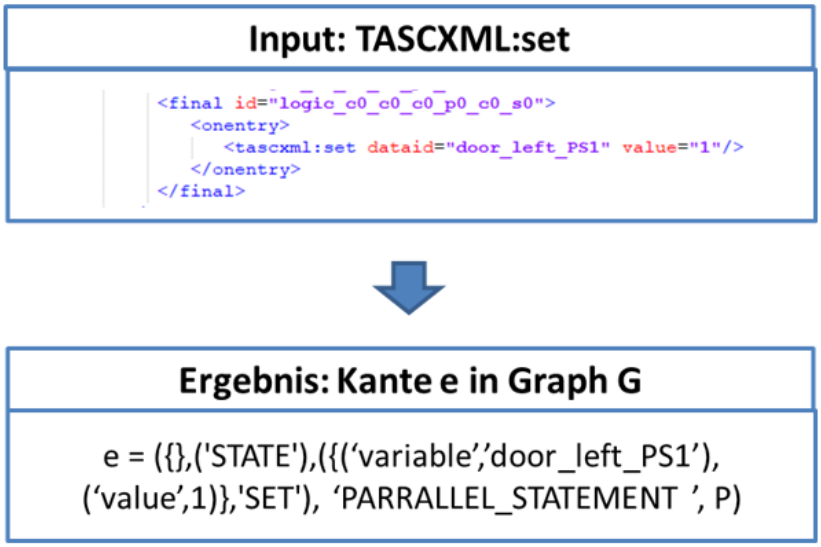

Abbildung 45: Beispiel für die Abbildung vom TASCXML:set auf eine Kante im Graph

5.2.2.3 Anfragesprache für die Suche im Graph

Die Ergebnisse sind Testfälle in Graph G und die Fähigkeit, jeden Testfall aus einem Testskript in Sprache in Graph G zu übersetzen. Die Struktur von Graph G ist mit bestehenden Graphdatenbanken wie Neo4J kompatibel. Es garantiert, dass jeder Graph G in eine Graphdatenbank hochgeladen werden kann.

Das Abfragen von Mustern ist essenziell, um Differenzen zwischen Testfällen zu finden, um Facetten zu erstellen und die explorative Suche zu ermöglichen. Die Definition der Abfragen ist über die bereitgestellte Abfragesprache der zugrundeliegenden Graphdatenbank möglich. Der vorgesehene Benutzer ist ein Testingenieur, der mit Abfragesprachen nicht vertraut ist. Nach seiner Erfahrung ist ein Testfall eine Abfolge von Blöcken, in denen das SUT stimuliert und anschließend überprüft wird, wobei ein Stimulus ein Signal ist, das er auf dem SUT stimulieren möchte (Semantik). Für ihn ist ein Stimulus kein Knoten in einem Graphen, der durch eine Bezeichnung und Eigenschaften (Syntax) definiert ist. Eine geeignete Abfragesprache muss auf der Definition von Anweisungen und Blöcken anstelle von Knoten und Kanten aufgebaut werden. Die Herausforderung besteht darin, dass Abfragesprachen für Graphen genau Knoten und Kanten verwenden, um Abfragen zu definieren. Daher ist eine Abstraktion der nativ bereitgestellten Abfragesprache erforderlich, um die Lücke zwischen dem anvisierten Anwendungsszenario und dem zugrundeliegenden Technologie-Stack zu schließen.

Der Ansatz dieses Beitrags besteht darin, eine Abfragesprache als Mustersprache zu definieren, in der der Testingenieur seine Abfragen so definiert, wie er Testfälle erstellen würde. Dazu wurde eine Abfragesprache entwickelt, deren Grundstruktur im Folgenden dargestellt wird:

```
grammar TestPatternReduced;                                                    (1)

compilationunit: startExpression statements? EOF;                              (2)

startExpression: 'Pattern'  ENDCharacter;                                      (3)

statements: (statement | block) +;                                             (4)

statement: blockingStatement | expectExpression | waitExpression |
waitExpressionUntil | lambdaExpression;                                        (5)

blockingStatement: rampExpression | setExpression | verdictExpression;         (6)

block: LBRACE blockingStatement+ RBRACE;                                       (7)

setExpression: SET qualifiedNameWithOptionalDot (value unit?) ENDCharacter ;   (8)

expectExpression: EXPECT logicalexpression comparison ENDCharacter;            (9)

waitExpression: WAIT value unit? ENDCharacter; (10)

lambdaExpression: MULTIPLICATION qualifiedNameWithOptionalDot (value unit?)? ENDCharacter;
                                                                               (11)
ENDCharacter: '; '

SET: 'set';                                                                    (12)

EXPECT: 'expect';                                                              (13)

WAIT: 'wait';                                                                  (14)
```

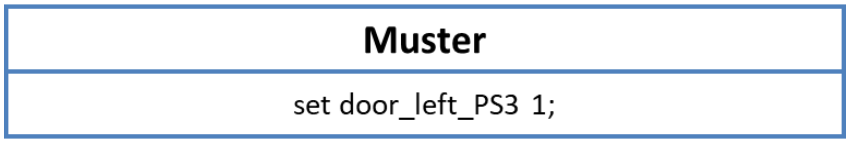

Abbildung 46: Anfrage für das Beispiel Stimulus I

Die vorgeschlagene Mustersprache definiert eine entsprechende Anweisung für jede TASCXML-Anweisung des Graphen G. Die Labels und Eigenschaften des Graphknotens V werden auf eine Musteranweisung abgebildet. Eine Musteranweisung besteht aus einer Sequenz an Token, wobei die Position des Tokens die Semantik definiert. Nach dieser Vorgehensweise lassen sich spezifische Wertzuweisungen für ein Signal oder eine Rampe realisieren. Ein Beispiel ist in Abbildung 46 gezeigt. Es

zeigt, dass das Signal door_left_PS3 auf 1 gesetzt ist. In den meisten gängigen Testfällen werden die Stimuli nicht durch eine einzelne Anweisung, sondern durch eine Folge von Anweisungen definiert. Um diese Funktion zu unterstützen, kann der Benutzer auch eine Folge von Anweisungen definieren, indem er der Musterdefinition weitere Anweisungen hinzufügt.

Ein Beispiel ist in Abbildung 47 dargestellt. Das Beispiel zeigt eine Stimulierung, welche aus vier Signalen besteht. Die Unterstützung von Sequenzen ermöglicht die Bestimmung der Unterschiede zwischen Testfällen. Der Benutzer der Fehlersuchmethode 2 kann Testfälle als Muster überführen und erhält eine Sequenz aus Informationen. Jede Information ist hierbei einer Informationsgruppe zugeordnet und dementsprechend können die Unterschiede als *Indikatoren* für die Fehlersuchmethode 2 abgeleitet werden (siehe Tabelle 17 aus Unterkapitel 4.3). Des Weiteren können Platzhalter als Ersatz für jedes Token einer Anweisung hinzufügt werden,

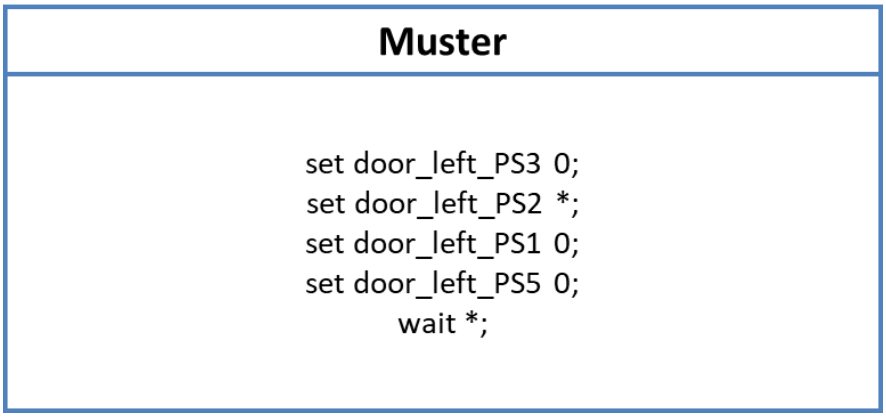

Abbildung 47: Anfrage für das Beispiel Stimulus II

um irrelevante Unterschiede für die Mustersuche ausschließen zu können. Das bedeutet, dass der Testingenieur die Funktion, das Signal und den Wert auf einen Platzhalter setzen kann. Im gegebenen Beispiel gibt der Benutzer alle Details für die linke Tür an und fügt die Platzhalter für den Wert des Signals set_door_left_PS2 hinzu.

Die entwickelte Mustersprache kann nicht direkt in einer Graphdatenbank wie Neo4J ausgeführt werden, da die Datenbank Cypher als Abfragesprache verwendet. Der Benutzer definiert sein Muster in der Mustersprache, und das entwickelte Mapping wandelt es in eine Cypher-Abfrage um. Dabei übersetzt das Abbildungsverfahren jede Musteranweisung in einen Knoten, wobei das Schlüsselwort der Anweisung auf ein Label abgebildet wird und alle Parameter ebenfalls als Knotenparameter abgebildet werden. Ein Beispiel für die Zuordnung ist in Abbildung 48 gezeigt.

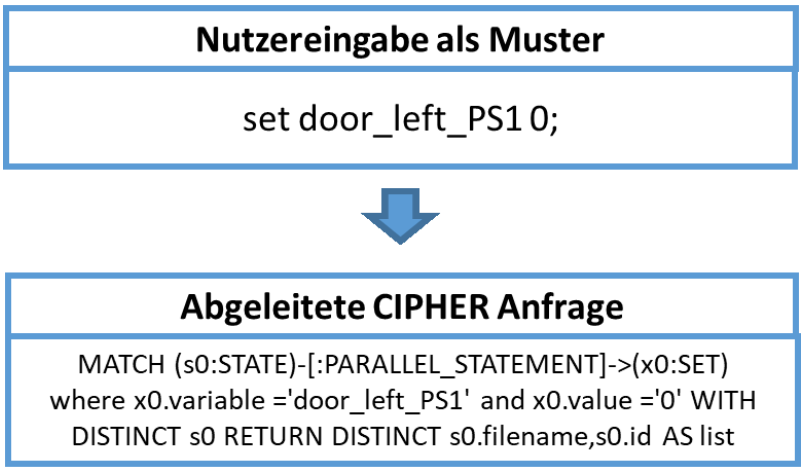

Abbildung 48: Abbildung eines Stimulus auf eine Cypher Anfrage

5.2.3 Fehlersuchmethode 3: Anpassung der Konfiguration und anschließende Ausführung

Die Fehlersuchmethode 3 ist eine dynamische Analysemethode und analysiert nicht primär die Struktur des Testfalls, sondern die Auswirkungen der Testfallausführung bei Anwendung unterschiedlicher Konfigurationen auf das Verhalten des SUT. Der fehlende Lösungsbaustein ist eine Funktion für die Testautomatisierung, die alternative Konfigurationen bestehend aus Testfall und Konfiguration erzeugen kann. Dazu werden die Funktion, das benötigte Informationsmodell und die Transformation der Konfigurationen im Detail vorgestellt. Die Inhalte basieren auf der Veröffentlichung [Franke et al. 2020b] und [dSPACE GmbH 2021].

Der Ansatz konzentriert sich auf die gezielte Änderung der Konfiguration eines fehlgeschlagenen Testfalls und die anschließende erneute Testausführung desselben Testfalls. Die Auswahl der nächsten Konfiguration basiert auf einem erkannten Problem in der Testausführung und einem Regelwerk. Die Funktion wird im Folgenden adaptive *Testrückkopplungsschleife* genannt. Im Folgenden wird der Ablauf der Testausführung mit der integrierten *Testrückkopplungsschleife* in der Abbildung 49 gezeigt. Hierbei sind alle kursiv gedruckten Schlüsselwörter Referenzen auf das Informationsmodell.

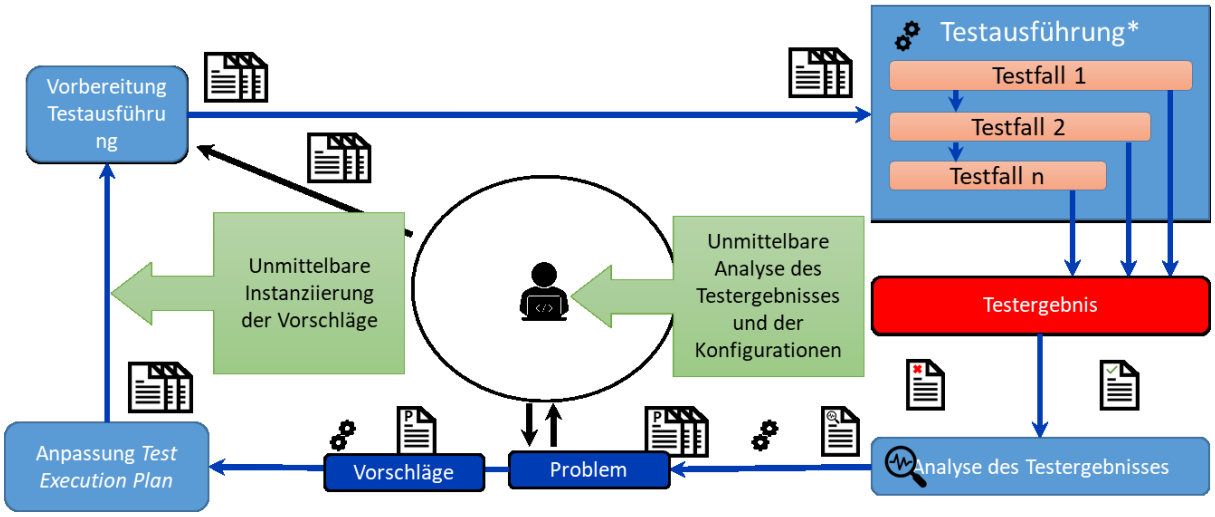

Abbildung 49: Detaillierter Ablauf der Methode 3, direkt übernommen von [Franke et al. 2020b]

Die Vorbereitung der Testausführung beginnt mit der Erstellung des *Test Execution Plans*. Dieser beschreibt die Abfolge der Testfälle einschließlich ihrer Konfiguration. Im Rahmen der kollaborativen Fehlersuche enthält der *Test Execution Plan* nur den fehlgeschlagenen Testfall.

Der *Test Execution Plan* und alle referenzierten Dateien werden für die Testausführung zentral verwaltet und für die Ausführung bereitgestellt. Das dazu notwendige Informationsmodell ist im Unterkapitel 5.2.3.1 beschrieben. Anschließend wird der Testfall inklusive seiner Konfiguration an ein Testsystem zur Ausführung übergeben. Während der Testausführung generiert das Testsystem Logmeldungen, welche sowohl die aufgetretenen Fehler, das Ergebnis der Verdikte als auch die ausgelösten Ereignisse enthalten. Die Methode 3 verwendet insbesondere die Verdikte und die Ereignisse für ihre Analysen. Die Ereignisse sind durch die Testfälle in *Generic SCXML* Issue 5.1 Bestandteil eines Testfalls und beschreiben den Zustand des Testobjekts.

Dadurch kann sofort festgestellt werden, in welchem Testschritt sich das Testobjekt befindet und welche beobachteten Ereignisse aufgetreten sind. Anschließend ermittelt das Testsystem aufgrund der Datenlage das Problem und fügt diese Information ebenfalls dem *Test Execution Series Objekt* hinzu. Das *Test Execution Series Objekt* enthält somit den *Test Execution Plan*, den Status der Testausführung und eine Problemdefinition. Das aktualisierte *Test Execution Series Objekt* wird dann verwendet, um Vorschläge für die Änderung einer Konfiguration zu berechnen. Hierfür müssen dem zentralen Repository bereits ähnliche Konfigurationen und Test Execution Plans vorliegen. Die Art der Vorschläge ergibt sich aus einer Regel, die die Abbildung der Problemdefinition auf den Vorschlagstyp beschreibt.

Jeder Vorschlagstyp erzeugt mehrere Vorschläge auf der Grundlage der Problembeschreibung und des Status der Testausführung. Die Anzahl der Vorschläge ergibt sich aus dem Wissen, das aus den verfügbaren Testfällen und Konfigurationen abgeleitet werden kann. Die generierten Vorschläge unterscheiden sich semantisch und reichen von der Änderung der Konfiguration bis hin zur Einbeziehung von generierten *Observern* für die Zustandsüberwachung. Allen Vorschlägen ist gemeinsam, dass sie als *Proposals* im *Test Execution Series Objekt* gespeichert werden und alle die gleiche Struktur haben. Jeder Vorschlag ist ein neuer *Test Execution Plan*. Die Vorschläge können direkt von einem Testsystem ausgeführt werden und die Ergebnisse der Ausführung können im Test Execution Series Objekt gespeichert werden.

Die automatische Bereitstellung und Ausführung des Testfalls in variierenden Konfigurationen ermöglichen die explorative Identifikation fehlerhafter Konfigurationen. Das Ergebnis der Fehlersuchmethode 3 ist der Indikator, ob die Konfiguration der Grund für den fehlgeschlagenen Testfall ist.

Die notwendigen Informationen für die *Testrückkopplungsschleife* sind der Testfall, seine Konfiguration, Konfigurationen aus anderen Testprozessen und die Ergebnisse der Testausführung. All diese Informationen sind im folgenden Informationsmodell detailliert beschrieben.

5.2.3.1 Informationsmodell der Testrückkopplungsschleife

Das Informationsmodell konzentriert sich auf die Informationen, die im Lebenszyklus eines Testfalls im Rahmen der Testvorbereitung, Testdurchführung und Testauswertung benötigt werden. Eine Übersicht ist in der Abbildung 50 gegeben. Dabei endet der Lebenszyklus eines Testfalls nicht nach der Auswertung der Testergebnisse, sondern der in Abbildung 49 dargestellte Schritt der Neuplanung (Generierung von Vorschlägen und Auswahl eines Vorschlags) schließt den Testvorbereitungszyklus.

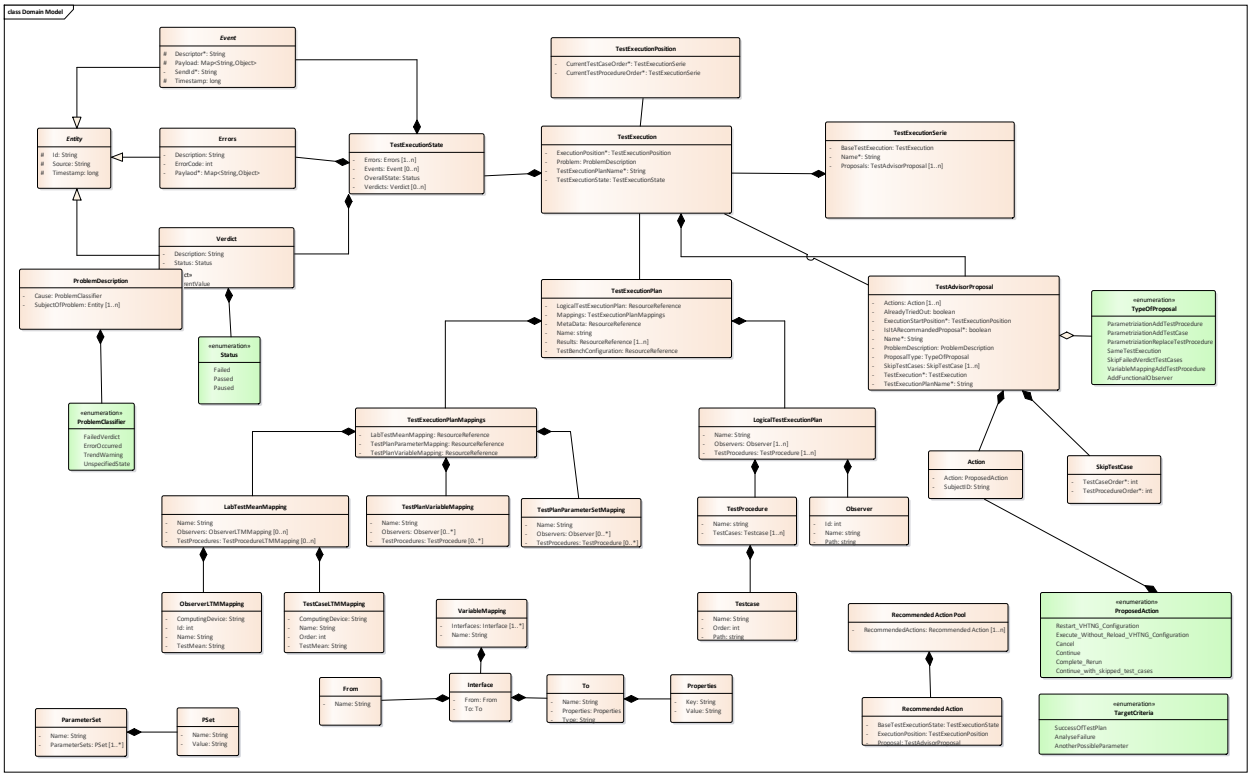

Abbildung 50: Domainenmodell für die Fehlersuchmethode 3

In allen Phasen des Lebenszyklus werden Informationen benötigt und Ergebnisse erzeugt. Um diese Informationen verwalten zu können, werden sie in der Klasse *Test Execution Series Object* gespeichert. Der Lebenszyklus des *Test Execution Series Object* beginnt mit einem Testfall. Dieser wird über einen *Test Execution Plan* mit einer Konfiguration für den Testfall und einer Konfiguration für das Testsystem verknüpft. Dadurch wird eine automatisierte Testfallausführung ermöglicht [dSPACE GmbH 2021] .

Die nächste Phase des Lebenszyklus ist die Testfallausführung. Dabei werden Informationen generiert, die eine spätere Auswertung der Testausführung ermöglichen. Diese Informationen werden daher in der Klasse *Test Execution State* repräsentiert. Der Ausschnitt des *Test Execution State* ist in Abbildung 51 dargestellt.

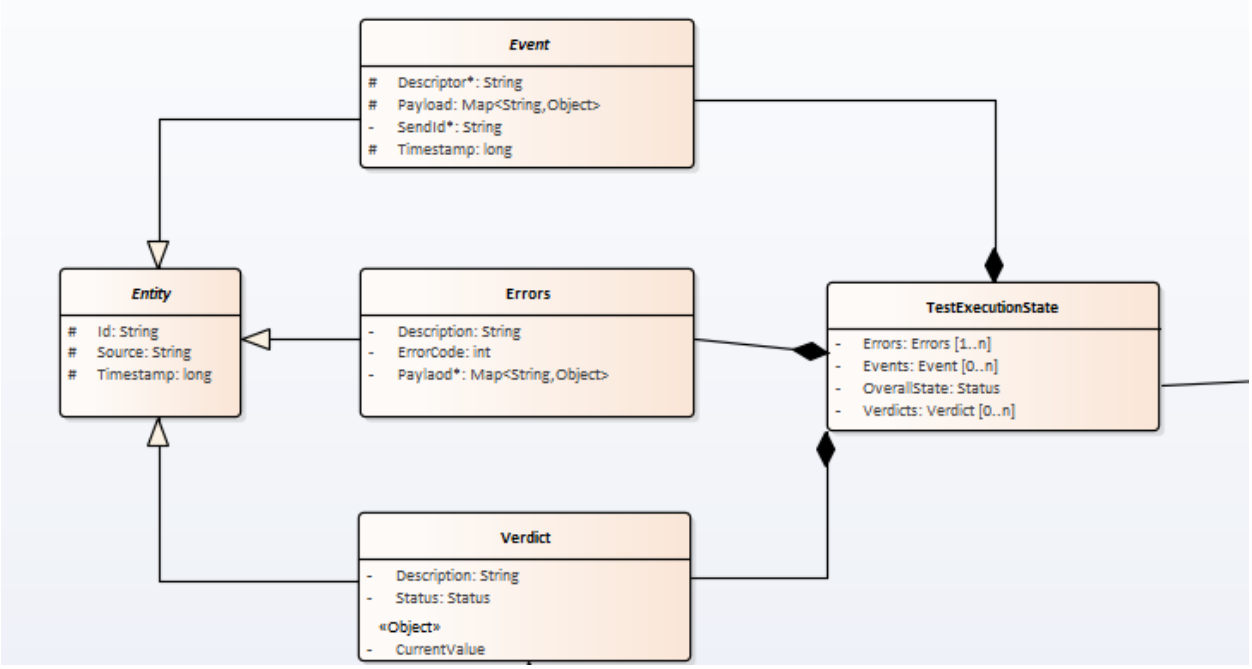

Abbildung 51: *Test Execution State* als Teil des Informationsmodells

Im Rahmen der Testausführung werden Ereignisse als *Events* aufgezeichnet. Beispiele für relevante Ereignisse sind z.B. welche Testschritte erfolgreich betreten und verlassen wurden. Zu jedem Ereignis werden der Zeitstempel und die Position im Testfall gespeichert. Diese Informationen sind notwendig, um später Fehler im Laufzeitverhalten des Testfalls finden zu können. Die aufgezeichneten Fehler geben Hinweise auf fehlerhafte Konfigurationen des Testsystems, die sich in Abstürzen mit entsprechenden Fehlercodes äußern. Die letzte Kategorie sind die *Verdicts*. Mit ihnen wird überprüft, ob der Testfall inhaltliche Fehler enthält. Diese können z.B. durch eine fehlerhafte Stimulation entstehen. Anhand dieser Informationen wird berechnet, ob die Ausführung des Testfalls erfolgreich war. Das Ergebnis wird in der Variablen *Overall State* des *Test Execution State* gespeichert. Wenn der Testfall fehlgeschlagen ist, wird eine Problembeschreibung erzeugt. Diese ist in Abbildung 52 dargestellt.

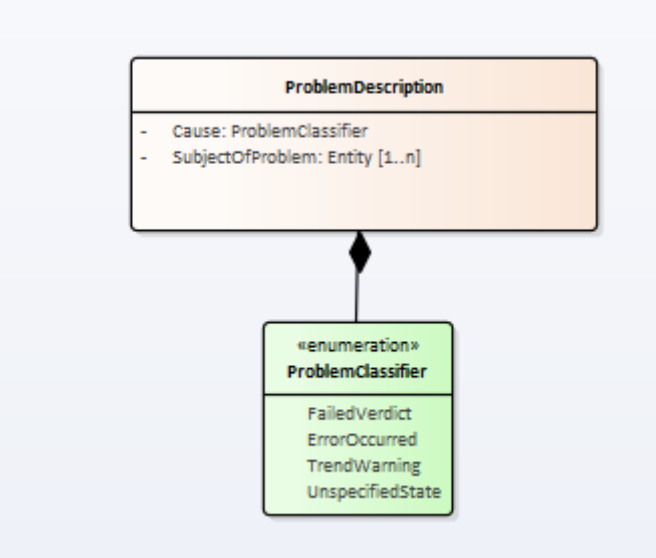

Abbildung 52: Ausschnitt der Problembeschreibung der *Test Execution Series*

Basierend auf der Problembeschreibung werden anschließend Vorschläge als *Proposals* generiert, welche den Testfall inklusive der neuen Konfiguration enthalten. Diese Vorschläge werden dem *Test Execution Series* zugewiesen. Dieser Abschnitt des Informationsmodells ist in Abbildung 53 zu sehen.

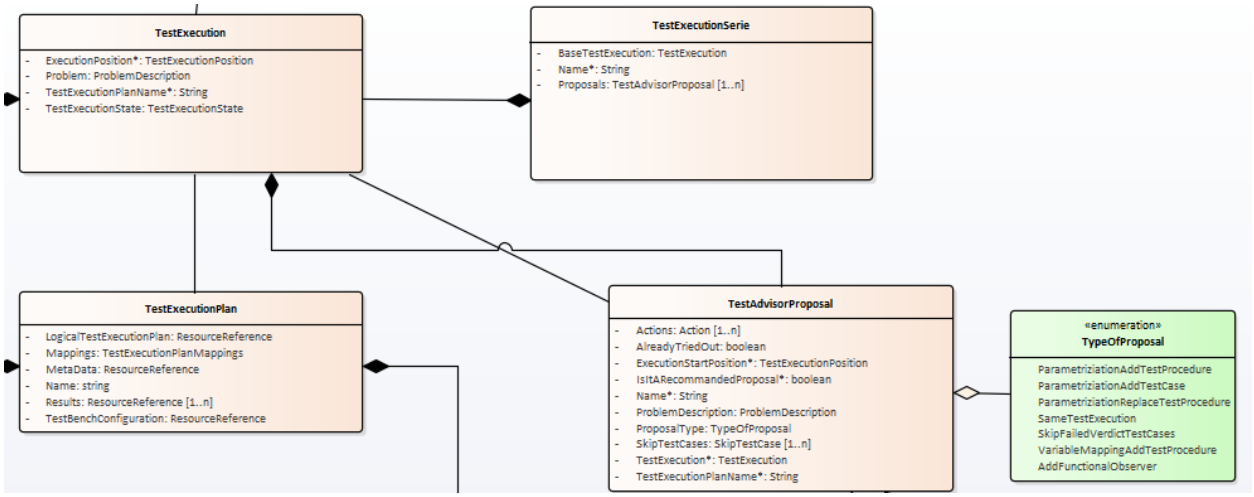

Abbildung 53: Ausschnitt der Vorschläge der *Test Execution Series*

Abschließend kann ein Vorschlag ausgewählt werden und es können alle Lebenszyklus Phasen durchlaufen und die Informationen innerhalb des Vorschlags protokolliert werden.

5.2.3.2 Generierung von Varianten der Testdurchführung

Ziel der *Testrückkopplungsschleife* ist die Generierung von Varianten einer Testdurchführung. Dazu benötigt der Generierungsprozess als Eingabe den Testfall mit seiner Konfiguration, den protokollierten Ereignissen, Fehlermeldungen, Verdikten und der identifizierten Problembeschreibung. Zu jeder Problembeschreibung können eine Reihe spezifischer Vorschläge als Varianten einer Testdurchführung generiert werden. Dabei wird jede Variante als Vorschlag (*Proposal*) in die *Test Execution Series* aufgenommen. Die Abbildung einer Problembeschreibung auf die zu generierende Variante ist in Tabelle 24 dargestellt.

Tabelle 24: Anpassung des *Test Execution Plans* für die erneute Testausführung

Problem	Typ des Vorschlags	Beschreibung
Ein fehlgeschlagenes *Verdikt* ist aufgetreten	Ein Austausch der Konfiguration bezog sich auf die Abbildung der Signale	Die dem fehlgeschlagenen Testfall zugeordnete Variablenzuordnung wird ersetzt. Dazu werden alle gültigen Variablenzuordnungen für den angewendeten logischen Testfall gesucht und für jede davon ein neuer Testausführungsplan abgeleitet.
Ein fehlgeschlagenes *Verdikt* ist aufgetreten	Ein Austausch der Konfiguration bezog sich auf die Abbildung der Parameter	Die dem fehlgeschlagenen Testfall zugeordnete Parameterzuordnung wird ersetzt. Dazu werden alle gültigen Abbildungen der Parameter für den angewandten logischen Testfall gesucht und jeweils ein neuer Testausführungsplan (*Test Execution Plan*) abgeleitet.
Ein Testsystem spezifischer Fehler ist aufgetreten	Ein Austausch der Konfiguration bezogen auf das Testsystem	Die dem Testausführungsplan zugeordnete Konfiguration des Testsystems wird ersetzt. Dazu werden alle gültigen Konfigurationen für den angewendeten logischen Testausführungsplan (*Test Execution Plan*) gesucht und für jede davon ein neuer Testausführungsplan (*Test Execution Plan*) abgeleitet.
Die Testausführung ist einem unbekannten Zustand eingetreten	Die Konfiguration wird um einen Beobachter erweitert	Es wird ein Beobachter abgeleitet, der den konkreten Zustand über die gesamte Dauer der Ausführung beobachtet. Der entsprechende Beobachter wird einem Klon des ursprünglichen Testausführungsplans (*Test Execution Plan*) hinzugefügt.

Die Auswahl der Vorschläge ist domänenspezifisch und muss entsprechend konfigurierbar sein. So kann eine Abbildung eines Problems auf einen Vorschlagstyp konfiguriert werden. Dabei wird jedoch nicht direkt auf den Vorschlagstyp verlinkt, sondern auf die Entität, die den Vorschlag erzeugen kann. Im Kontext des Funktionsmusters sind dies JAVA-Klassen. Ein Beispiel für eine solchen Regelsatz ist in Abbildung 54: Regelsatz für die Konfiguration gegeben.

```
{
   "branches":[
      {
         "classifier":"FailedVerdict",
         "factories":[
            {
               "javaFactoryClasss":"ChangeVariableMappingOfFaultyTestCaseServiceAddTestProcedure",
               "actionsToBePerformed":[
                  {
                     "environmentAction":"Execute_Without_Reload_VHTNG_Configuration",
                     "executionAction":"Continue"
                  }
               ]
            }
         ]
      },
      {
       "classifier":"UnspecifiedState",
         "factories":[
            {
               "javaFactoryClasss":"AddFunctionalStateObserver",
               "actionsToBePerformed":[
                  {
                     "environmentAction":"Restart_VHTNG_Configuration",
                     "executionAction":"Complete_Rerun"
                  }
               ]
            }
         ]
      }
   ]
}
```

Abbildung 54: Regelsatz für die Konfiguration der *Testrückkopplungsschleife*

Der gezeigte Regelsatz aus Abbildung 54 besteht aus zwei Regeln. Die erste Regel definiert, dass sie Probleme vom Typ *FailedVerdict* zuständig ist. Das bedeutet, dass die Regel nur angewendet werden kann, wenn in der Testausführung ein Verdikt fehlgeschlagen ist und dieses im *Test Execution State* auftaucht. Dieses Problem ist auch das adressierte Problem in der Fehlersuchmethode 3 der Fehlersuche. Sobald das Problem erkannt wird, definiert die Regel, welche Art von Vorschlägen generiert werden sollen. In der ersten Regel wurde eine Java Klasse gewählt, welche die Konfiguration für die Signale austauscht. Abschließend definiert die Regel wie die Testausführung fortzusetzen ist. In der ersten Regel wird definiert, dass die Testausführung ohne einen Neustart fortgesetzt werden soll.

6 Evaluation der kollaborativen Fehlersuche

Kapitel 5 spezifiziert drei Fehlersuchmethoden zur Unterstützung der kollaborativen Fehlersuche in *lose gekoppelten* Testprozessen der Zulieferpyramide. Die Ergebnisse der Fehlersuchmethoden sind *Indikatoren* für die Gründe des Scheiterns des Testfalls im Testschritt *Realisierung und Durchführung*.

Im Rahmen der Evaluation werden drei Fehlersuchmethoden einzeln und nicht zusammen evaluiert. Die getrennte Auswertung ist möglich, weil jede der drei Fehlersuchmethoden in Bezug auf ihre Ein- und Ausgaben unabhängig voneinander ist. Die Ausgabe der Fehlersuchmethoden sind drei disjunkte Sätze von *Indikatoren*, welche die drei Gruppen der Unterschiede (Unterschiede im Laufzeitverhalten des Testfalls, Strukturelle Unterschiede im Testfall, Strukturelle Unterschiede in der Konfiguration) abdecken.

Die drei Evaluationen wurden im Rahmen der Forschungsprojekte STEVE und AGI-LE-VT durchgeführt. Der Fokus der Evaluation lag dabei immer auf den Funktionsmustern, welche die entwickelten Algorithmen der Fehlersuchmethoden enthielten. So konnten die Algorithmen evaluiert und damit ihre Funktionalität für die Fehlersuche untersucht werden. Jede Evaluation wurde an realen Anwendungsfällen von Airbus durchgeführt. Dazu wurden für jeden Airbus-Anwendungsfall ein mechatronisches System als Prüfling sowie Testfälle und Simulationsmodelle für den Testschritt *Realisierung und Durchführung* bereitgestellt.

Die Evaluierung der drei Funktionsmuster hat gezeigt, dass die entwickelten Informationsmodelle und die darauf aufbauenden Algorithmen funktionieren. Somit sind die entwickelten Fehlersuchmethoden auch im Verbund als kollaborative Fehlersuche einsetzbar. Diese Annahme wird zusätzlich dadurch gestützt, dass die Eingaben für die Evaluation mit den spezifizierten Eingaben der Fehlersuchmethoden übereinstimmen. Das Funktionsmuster für den Schritt 1: Interoperabilität der Testdaten der Fehlersuche ist Bestandteil der Fehlersuchmethode 1. In den folgenden drei Unterkapiteln (6.1-6.3) wird die Evaluation im Detail vorgestellt.

6.1 Ausführung des Testfalls und Analyse des Signalverlaufs

Die Evaluation der **Fehlersuchmethode 1: Ausführung des Testfalls und Analyse des Signalverlaufs** der kollaborativen Fehlersuche wurde bereits im Rahmen des Forschungsprojektes STEVE durchgeführt und veröffentlicht [Rasche et al. 2018; Franke und Thoben 2022].

Ausgangssituation

Die Ausgangssituation ist, dass eine kollaborative Fehlersuche notwendig ist, weil die Testprozesse des OEM und des Modul-Lieferanten zu unterschiedlichen Ergebnissen gekommen sind (*Variante 2*). In diesem Fall ist der Testfall beim OEM fehlgeschlagen und soll nun beim Modul-Lieferanten ausgeführt werden. Die Herausforderung besteht darin, dass in der Zulieferpyramide unterschiedliche *Testskriptsprachen* verwendet werden, weswegen der Testfall vom OEM nicht beim Modul-Lieferanten ausgeführt werden kann.

Ziel

Das Ziel der Evaluation ist der Nachweis, dass die Übersetzung von Testfällen zwischen zwei *Testskriptsprachen*, welche sich strukturell unterscheiden, möglich ist. Hierfür wurden CCDL (Prozedural) und RTT (Objektorientiert) ausgewählt, welche sich in dem Programmierparadigma unterscheiden.

Evaluationsszenario

Das Anwendungsszenario konzentriert sich auf den Austausch eines Testfalls zwischen Airbus in der Rolle des OEM und dSPACE in der simulierten Rolle des Modul-Lieferanten. In dem Szenario ist das SUT das Hochauftriebssystem eines Flugzeugs. Das Vorflügelsystem steht im Fokus der Hochauftriebsvorrichtung (siehe Abbildung 1 für eine Übersicht), die normalerweise am Flügel montiert wird. Das Ziel des Vorflügels ist es, den Anstellwinkel des Flügels entsprechend der Flugphase anzupassen. Dazu kann die Klappe aus- oder eingefahren werden. Ein Computer, der so genannte SFCC, steuert seine Bewegung.

Evaluationsgegenstand

Der Evaluationsgegenstand ist ein Testfall in der Testskriptsprache CCDL von Airbus. Der Testfall dient der Überprüfung einer Monitorfunktion im Hochauftriebssystem. Zu diesem Zweck werden einige Klappenstellungen und Antriebssequenzen durch den Testfall gefahren. So wird für jede definierte Position über einen Klappenhebelbefehl eine Fahrsequenz ausgelöst und der Test das Hochauftriebsverhalten gemäß den Anweisungen des Typs *Verdikt* bewertet.

Evaluationsmethode

Die Evaluationsmethode überprüft, ob der übersetzte Testfall in RTT semantisch mit dem Testfall CCDL übereinstimmt. Hierfür werden beide Testfälle ausgeführt und anschließend werden die Signalverläufe und die Ergebnisse der *Verdikte* miteinander verglichen.

Die Vorbedingung für die Transkompilierung und Ausführung des Testfalls waren die Folgenden:

- Die *Testskriptsprachen* verfügen über kompatible Anweisungen, welche ineinander übersetzt werden können.
- Das Simulationsmodell für High-Lift auf dem Airbus Testsystem verhält sich ähnlich zu dem Simulationsmodell auf dem dSPACE Testsystem bezogen auf die Fahrsequenzen.

Ablauf

Zuerst wird der Testfall von CCDL über *Generic SCXML* nach RTT übersetzt. Hierfür wird der *Test Case Translator* eingesetzt. Anschließend wird der übersetzte Testfall auf dem Testsystem von dSPACE ausgeführt. Abschließend werden die Signalkurven und die Ergebnisse des *Verdikts* verglichen. Hierbei gilt das Evaluationskriterium, das die Signalverläufe und des *Verdikts* dasselbe Ergebnis zeigen müssen.

Evaluationsergebnis

Der Testfall konnte erfolgreich mithilfe der Fehlersuchmethode 1 und dem **Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche von CCDL nach RTT übersetzt werden. Es wurden für das Forschungsprojekt STEVE spezifische Bibliotheken für RTT von dSPACE implementiert, um die Lesbarkeit des übersetzten Testfalls zu verbessern und um die Flexibilität des Transformationsansatzes zu zeigen.

SCXML wurde bei der Querübersetzung als Informationsmodell angewendet. Die Ergebnisse wurden bereits veröffentlicht [Rasche et al. 2018]. Die Inhalte werden hier exakt wiedergegeben. Ein Ausschnitt des übersetzten Testfalls ist in Abbildung 55 gezeigt. Hierbei ist oben in der Abbildung der Ausschnitt des Testfalls in CCDL und unten der Ausschnitt des Testfalls in RTT zu sehen.

Ausschnitt des Testfalls in CCDL

```
/* *** ================================================================================
                                                    Test Step 2, Timeout 60 [sec]:
/* *** --------------------------------------------------------------------------------
{

wait until HLSF1.OD_POBOpen = 1

wait 2 [s]
   // expect that system drives with defined speed in RETRACT direction
   during 0 [s] .. 2000 [ms] : {
      // speed expected to be in between -396 [rpm] +/- 20 [rpm]
      expect HLSFA.OA_ShaftSpeedIB > (-FLAP_PCU_NOM / 2) - 20        (SRD:5207-3)
      expect HLSFA.OA_ShaftSpeedIB < (-FLAP_PCU_NOM / 2) + 20        (SRD:5207-3)
   }

wait until HLSF1.OD_POBOpen = 0
}
```

Ausschnitt des generierten Testfalls in RTT (Python)

```
## ================================================================================
    test_step("Test Step 2", timeout = 60)
## --------------------------------------------------------------------------------

    yield wait_until("HLSF1_OD_POBOpen == 1")

    yield wait(2)

    observe(0,
            2.0,
            expect_during("HLSFA_OA_ShaftSpeedIB > -396 - 20",  "during_2-1 SRD:5207-3"),
            expect_during("HLSFA_OA_ShaftSpeedIB < -396 + 20",  "during_2-2 SRD:5207-3"))

    yield wait_until("HLSF1_OD_POBOpen == 0")
```

Abbildung 55: Gleicher Ausschnitt aus den Testfällen in CCDL und RTT

Die anschließende Ausführung des CCDL-Testfalls bei Airbus und die anschließende Ausführung des Testfalls bei dSPACE zeigten **ähnliche** Signalkurven, was in Abbildung 56 gezeigt ist. Des Weiteren zeigten die Verdikte (*during* in CCDL und *expect_during* in RTT) die gleichen Ergebnisse. Hiermit konnte der *Indikator* VERDIKT-1 überprüft werden.

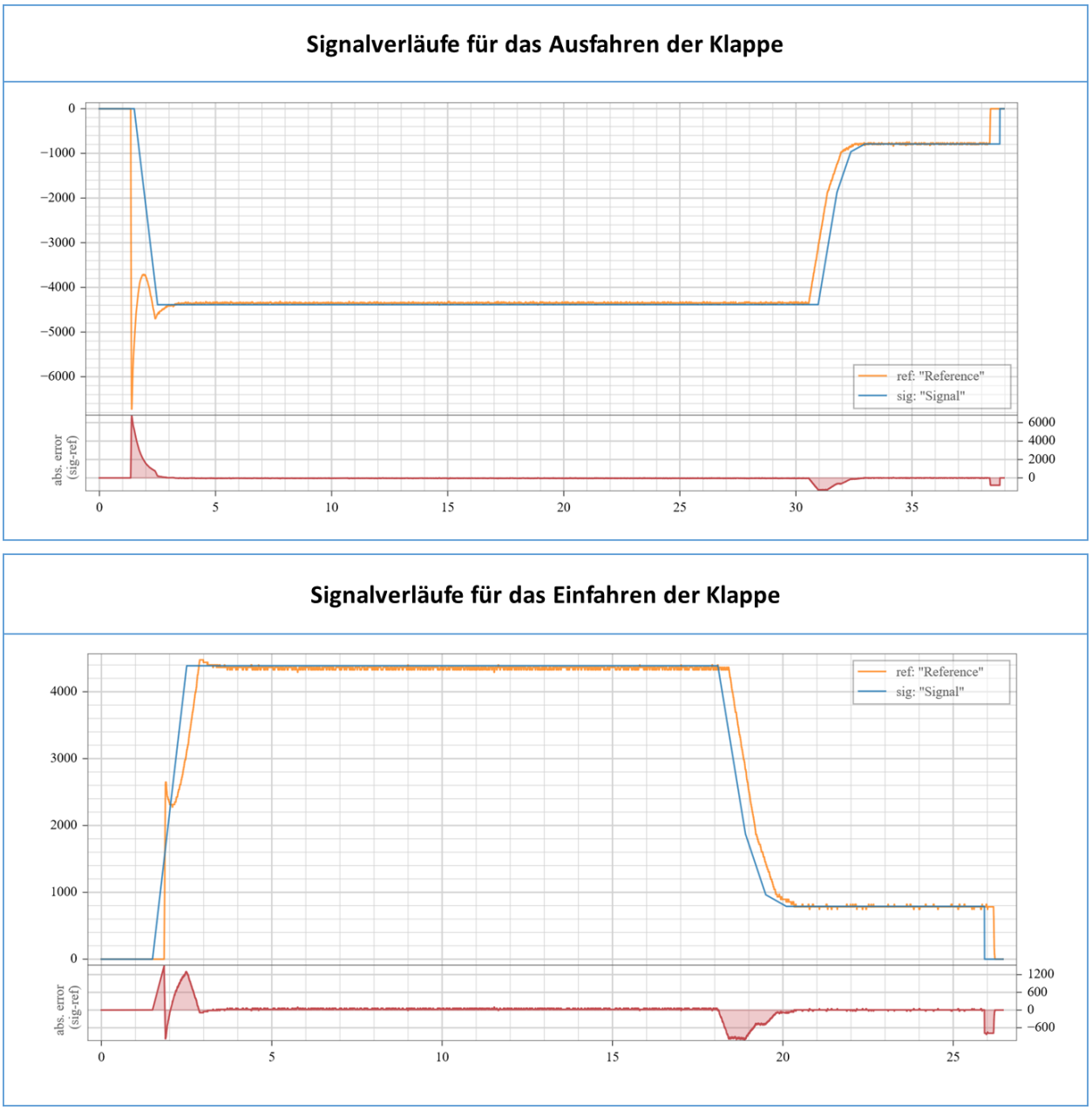

Abbildung 56: Entdeckte Unterschiede in den Signalkurven zwischen dem CCDL- und RTT-Testfall

Abbildung 56 zeigt einen Vergleich zwischen den Ergebnissen des Hardware-in-the-Loop-Tests mit CCDL und realem Flugzeugcontroller (orange Kurve) und dem Software-in-the Loop Test mit RTT und Simulationsmodell (blaue Wellenform). Hierbei wurde das Simulationsmodell für das Hochauftriebssystem gemäß der Spezifikation von dSPACE modelliert. Hierdurch konnten die *Indikatoren* STIMULUS-1, STIMULUS-2 und LAUF-1 überprüft werden. Die Analyse zeigt, dass die Simulationsergebnisse für das Ausfahren (links) und das Einfahren (rechts) geringfügig von dem ursprünglichen Test (CCDL) abweichen, bei dem der ursprüngliche Hochauftriebssteuercomputer beteiligt war. Eine manuelle Analyse, wie es im **Schritt 3: Erkenntnisse aus den Fehlersuchmethoden ziehen** der Fehlersuche motiviert ist,

wurde anschließend im Rahmen von STEVE durchgeführt. Als Grund für die Abweichung wurde festgestellt, dass das modellierte Hochauftriebssystem, das zur Ausführung des RTT-Testfalls verwendet wurde, auf High-Level-Anforderungen basiert, die die im Steuerrechner implementierte Luftlastkompensation nicht widerspiegeln. Die geringfügigen Unterschiede in der Fahrzeit sind auf den gleichen Effekt zurückzuführen. Daher entsprechen die Ergebnisse der erwarteten Ausgabe der Fehlersuchmethode 1 und ermöglichen die Identifizierung der *Indikatoren* STIMULUS-1, STIMULUS-2, VERDIKT-1 und LAUF-1.

6.2 Mustersuche

Die Evaluation der **Fehlersuchmethode 2: Mustersuche** der kollaborativen Fehlersuche wurde bereits im Rahmen des Forschungsprojektes AGILE-VT durchgeführt und veröffentlicht [Franke et al. 2020a]. Eine ergänzte Version der Evaluation wurde in [Franke et al. 2023] veröffentlicht.

Ausgangssituation

Die Ausgangssituation ist, dass eine kollaborative Fehlersuche notwendig ist, weil die Testprozesse des OEM und des Modul-Lieferanten zu unterschiedlichen Ergebnissen in ihren Testprozessen gekommen sind. Die Testingenieure aus beiden Testprozessen müssen verstehen, wie die Stimulierung und die anschließende Überprüfung der Reaktion des SUT in den Testfällen des OEM und des Modul-Lieferanten erfolgt sind. Diese Informationen sind in allen Testfällen enthalten. Die Herausforderung ist, dass die Testfälle in heterogenen *Testskriptsprachen* innerhalb der Zulieferpyramide repräsentiert sind. Das bedeutet, dass semantisch ähnliche Muster verteilt in Testfällen enthalten sind, diese aber für die Testingenieure versteckt sind.

Ziel

Das Ziel der Evaluation ist der Nachweis, dass die Aggregation von Testfällen als ein Graph funktioniert und darauf aufbauend die enthaltenen Muster (invers die Unterschiede) gefunden werden können.

Anwendungsszenario

Das Anwendungsszenario konzentriert sich auf die Suche von Mustern in Testfällen in einem Testprozess. Der Testprozess adressiert das Testen der Flugzeugtüren, welche in dem ATA-Kapitel 52 definiert sind. Hierbei steuert das Cabin Intercommunication Data System (CIDS) die Kabinensysteme. Es steuert z.B. die Rutschen/Türen, Wasser/Abfall, Rauchanzeige, Klimaanlage usw. Das CIDS steuert die Türen und teilt Statusinformationen mit anderen Systemen. Die Zustandsinformation einer Tür wird durch überwachbare Sensoren realisiert. Die Informationen werden verwendet, um zu klassifizieren, ob die Tür verriegelt, gesichert oder geschlossen ist. Alle Türen

implementieren die gleichen Systemzustände, werden aber leicht unterschiedlich beschrieben.

Evaluationsgegenstand

Der Evaluationsgegenstand beinhaltet Testfälle von 8 Türen, welche unterschiedliche Türen im Flugzeug repräsentieren. Die Testfälle werden alle in *Generic SCXML* bereitgestellt. Alle Testfälle testen, ob sich die Türen über denselben Mechanismus verschließen und verriegeln lassen.

Evaluationsmethode

Die Evaluationsmethode überprüft, ob ein Muster für die Stimulation der Türen definiert und in den 16 Testfällen gefunden werden kann. Das Muster beinhaltet die Anweisungen vom Typ TASCXML:set- für die Positionssensoren und ist in den *Generic SCXML*-Testfällen im Start UP-Zustand enthalten. Der Inhalt des Musters beschreibt den Anfangszustand eines gewöhnlichen Türtestfalls.

Im zweiten Schritt wird überprüft, ob ein ganzer Testfall als Muster kodiert und anschließend in den 16 Testfällen gefunden werden kann. Dieser Schritt testet das beabsichtigte Einsatzszenario von der Fehlersuchmethode 2, welche strukturellen Unterschiede in den Testfällen identifizieren möchte.

Die Vorbedingung für die Mustersuche des Musters waren die Folgenden:

- Alle Testfälle der Kabinentüren werden in SCXML bereitgestellt. Zu diesem Zweck wurden die ursprünglichen Testfälle **mithilfe des Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche von Python in SCXML übersetzt.
- Der Testkorpus enthält 16 Testfälle für verschiedene Türen mit den gleichen Stimuli und der Überprüfung des gleichen Verhaltens. Sie unterscheiden sich nur in bestimmten Signalnamen. Für alle Türen wurde der Testfall für den Notbetrieb hinzugefügt.
- Die Testfälle für Normal- und Notbetrieb sind bis auf die notfallspezifischen Signalwerte ähnlich.
- Die Testfälle wurden in eine Neo4J Datenbank als Graph G importiert.
- Es wurde ein Muster bestehend aus 6 TASCXML:set Kommandos, welches mit einem Testfall (Erste linke Tür) übereinstimmt, und ein Muster bestehend aus einem kompletten Testfall für die Mustersuche bereitgestellt. Hierfür wurde der Testfall für die erste linke Tür im Notbetrieb als Muster übersetzt.

Anschließend wurden beide Muster in dem in Neo4J hinterlegten Knowledge Graph (Graph G) gesucht und die Ähnlichkeit zwischen dem Muster und dem Testfall berechnet.

Die Ähnlichkeit wird basierend auf der Übereinstimmung zwischen dem Testfall und dem ursprünglichen Muster definiert. Wenn das Muster exakt übereinstimmt, müssen keine Platzhalter für einen Treffer der Suche hinzugefügt werden und die Ähnlichkeit beträgt 100%. Wenn es keine Übereinstimmung mit dem ursprünglichen Muster gibt, wird das Muster manipuliert und ein Platzhalter hinzugefügt. Dabei ersetzt der Platzhalter den Wert einer Anweisung. Wenn es immer noch keine Übereinstimmung gibt, wird ein anderer Wert durch einen Platzhalter ersetzt. Wenn alle Werte iterativ durch Platzhalter ersetzt wurden und es immer noch keine Übereinstimmung gibt, dann werden die Variablen iterativ durch Platzhalter ersetzt. Wenn es immer noch keine Übereinstimmung gibt, wird der Typ der Anweisung iterativ durch Platzhalter ersetzt. Schließlich gibt es eine Übereinstimmung, wenn Platzhalter alle Tokens der Anweisungen ersetzt haben. Die Anzahl der hinzugefügten Platzhalter bestimmt die Ähnlichkeit des gefundenen Musters. Zu diesem Zweck wird die Anzahl an Tokens von den Platzhaltern subtrahiert und anschließend durch die Anzahl der Tokens dividiert, um einen prozentualen Wert zu erhalten.

Evaluationsergebnis

Das Ergebnis des oben genannten Testszenarios war, dass beide Muster in allen türrelevanten Testfällen mit dem Funktionsmuster *Test Advisor* gefunden wurde:

- Die Übereinstimmung war 100 % für den Testfall der ersten Tür mit dem ersten Muster
- Bei den anderen Testfällen war die Ähnlichkeit kleiner als 100 %. Der niedrigere Ähnlichkeitswert resultiert aus den unterschiedlichen Signalnamen für die anderen Türen.
- In jedem Testfall wurde der richtige Bereich des Musters gefunden.
- Die Übereinstimmung für beide Varianten des ersten Testfalls (Notfall, Normalbetrieb) war ebenfalls kleiner als 100 %, da sich beide Testfälle in den Signalen für den Notbetrieb unterscheiden.
- Das zweite Muster konnte in dem Testfall für die erste linke Tür im Normalbetrieb gefunden werden. Hierbei betrug die Länge des Musters 1001 Anweisungen und die Ähnlichkeit wurde mit 96,81% berechnet.

Das Ergebnis zeigt, dass die Mustersuche für Testfälle, die in *Generic SCXML* vorliegen müssen, funktioniert. Bei dem direkten Vergleich von Testfällen werden die

Indikatoren STIMULUS-3, VERDIKT-2, LAUF-2 und STIMULUS-4 direkt abgeprüft. Die Ergebnisse in dem Testfall für den normalen Betrieb und den Notfallbetrieb führten dazu das die *Indikatoren* STIMULUS-3 und STIMULUS-4 anschlugen. Dies ermöglicht eine statische Analyse von Testfällen aus heterogenen *Testskriptsprachen*. Es können Unterschiede in der Stimulierung und in der Systemreaktion der Testfälle des OEM und des Modul-Lieferanten gefunden werden. Daher entsprechen die Ergebnisse der erwarteten Ausgabe der Fehlersuchmethode 2 und können die *Indikatoren* STIMULUS-3, VERDIKT-2, LAUF-2 und STIMULUS-4 identifizieren.

6.3 Anpassung der Konfiguration und anschließende Ausführung

Die Evaluation der **Fehlersuchmethode 3: Anpassung der Konfiguration und anschließende Ausführung** der Fehlersuche wurde bereits im Rahmen von AGILE-VT durchgeführt und wurde in [Franke et al. 2020b] veröffentlicht.

Ausgangssituation

Die Ausgangssituation ist, dass eine kollaborative Fehlersuche notwendig ist, weil die Testprozesse des OEM und des Modul-Lieferanten in ihren Testprozessen zu unterschiedlichen Ergebnissen gekommen sind. Der Testingenieur aus dem Testprozess des OEM muss klären, ob der Grund für die fehlgeschlagene Testausführung in der Konfiguration liegt. Die Herausforderung hierbei ist, dass die Identifikation relevanter Konfigurationen, die Vorbereitung der Testausführung und auch die eigentliche Testausführung manuelle und zeitaufwändige Tätigkeiten sind.

Ziel

Ziel der Evaluation ist der Nachweis, dass der fehlgeschlagene Testfall mit einer anderen Konfiguration versehen, die Testausführung automatisiert vorbereitet, gestartet und die Testergebnisse ausgewertet werden können.

Anwendungsszenario

Das Anwendungsszenario konzentriert sich auf den Austausch von Konfigurationen im Rahmen der Testvorbereitung eines Testprozesses. Das adressierte SUT ist die Sauerstoffversorgung der Flugzeugkabine, die über das CIDS gesteuert werden kann. Zusätzlich verfügt das SUT über zwei simulierte Sauerstoffflaschen, die individuell gesteuert werden können.

Evaluationsgegenstand

Der Evaluationsgegenstand besteht aus einem Testfall und zwei anwendbaren Konfigurationen. Dabei besteht eine Konfiguration aus einer Abbildung von Signalen,

welche generische Signalnamen auf Signale des Testsystems abbildet, und einer Abbildung von Parameter auf Werte, welche verschiedene Werte für den Ausgangsdruck der Sauerstoffflaschen definiert. Dabei steuert die erste Abbildung von Signalen die Auswahl der ersten Sauerstoffflasche und die zweite Abbildung von Signalen die Auswahl der zweiten Sauerstoffflasche. Die verfügbaren Parameterwerte entsprechen den Werten einer vollen und einer leeren Flasche.

Evaluationsmethode

Die Evaluationsmethode überprüft, ob die Konfiguration bei einem fehlgeschlagenen Testfall automatisch durch einen neuen Vorschlag (*Proposal*) aktualisiert werden kann. Dazu wurde die Startkonfiguration so gewählt, dass während der Testausführung eine Warnung ausgegeben wird. Hierfür wurde der Testfall inklusive der fehlerhaften Konfiguration in ein *Test Execution Series* Objekt eingebettet und auf einem dSPACE Testsystem ausgeführt.

Die Vorbedingung für die Evaluation waren die Folgenden:

- Es stehen zwei Abbildungen von Signalen zur Verfügung, die unterschiedliche Sauerstoffflaschen abbilden.
- Es gibt bereits *Test Execution Plans* in der Datenbank, die die Abbildung von Signalen erfolgreich verwendet haben.
- Es gab bereits *Test Execution Plans* in der Datenbank, welche erfolgreich die Abbildung von Parametern eingesetzt haben.
- Es gab eine Konfiguration des Funktionsmusters *Test Advisor*, die den Austausch der Abbildung von Signalen und der Abbildung von Parametern bei einer Warnung des Typen *TrendWarning* vorsah.
- Es gab ein *Test Execution Series Objekt*, das den Testfall und die fehlerhafte Konfiguration enthielt, aber keine Vorschläge (*Proposals*) besitzt.

Nach der Ausführung des fehlerhaften *Test Execution Series* Objektes wird der Inhalt des Objektes überprüft.

Evaluationsergebnis

Die Ausführung des Testfalls führte wie erwartet zu einem Fehlschlag. Nach der Ausführung hat der *Test Advisor* das *Test Execution Series* Objektes angepasst. Die Anpassung ergänzte das erkannte Problem und Vorschläge (*Proposal*) für eine erneute Ausführung. Das generierte *Test Execution Series* Objekt ist in Abbildung 57 abgebildet.

Abbildung 57: Aktualisiertes *Test Execution Series* Objekt, welches Vorschläge enthält

Die Vorschläge beinhalten den Wechsel der Abbildung von Signalen (Proposal 1) und den Wechsel der Abbildung von Parametern (Proposal 2 &3), was in Abbildung 58 gezeigt ist.

Abbildung 58: Aufschlüsselung der Vorschläge der *Test Execution Series*

Das Ergebnis zeigt das die **Fehlersuchmethode 3: Anpassung der Konfiguration und anschließende Ausführung für Testfälle** die automatische Änderung der Konfigurationen basierend auf historischen Testläufen korrekt vorgeschlagen hat. Hierfür wurden automatisiert die *Indikatoren* von STIMULUS-5 und STIMULUS-6 für die Anpassung der Konfiguration geprüft. Dementsprechend entsprechen die Ergebnisse der erwarteten Ausgabe der Fehlersuchmethode 3 und ermöglichen durch einen nachgeschalteten manuellen Vergleich der fehlgeschlagenen Konfiguration mit der erfolgreichen Konfiguration die Identifizierung der *Indikatoren* STIMULUS-5, STIMULUS-6, STIMULUS-7 und VERDIKT-3.

7 Diskussion der gewonnenen Erkenntnisse

In diesem Kapitel werden die Herausforderungen und Erkenntnisse für die kollaborative Fehlersuche aus verschiedenen Perspektiven diskutiert. Für jede Perspektive gibt es ein Unterkapitel, die folgend kurz zusammengefasst werden.

In Unterkapitel 7.1 werden zunächst die Herausforderungen für die kollaborative Fehlersuche in *lose gekoppelten* Testprozessen dargestellt. Hierbei wird auf das Problem der Heterogenität der Testprozesse eingegangen. Anschließend wird der Ansatz der Interoperabilität motiviert und die Ergebnisse der Transkompilierung und der Mustersuche vorgestellt. Hierbei entsprechen diese beiden Methoden den Lösungsansätzen der kollaborativen Fehlersuche. Anschließend wird auf Basis der Ergebnisse diskutiert, ob die Forschungsfrage beantwortet ist und ob abschließend die Anforderungen an eine kollaborative Fehlersuche erfüllt sind.

In Unterkapitel 7.2 werden die Ergebnisse in Bezug auf die Interoperabilität der Testfälle und den Stand der Technik diskutiert. Es wird zusammengefasst, ob die entwickelten Informationsmodelle und Transformationen für die Transkompilierung und der Mustersuche mit dem Stand der Technik kompatibel sind.

In Unterkapitel 7.3 werden die Auswirkungen der entwickelten Methoden auf den Stand der Technik bezogen auf die Transkompilierung und Mustersuche vorgestellt.

In Unterkapitel 7.4 wird der Mehrwert der Interoperabilität von Testfällen für die kollaborative Fehlersuche in der Praxis vorgestellt. Dafür wird die Anwendbarkeit und die sich daraus ergebenden Auswirkungen der Transkompilierung und Mustersuche auf die Testprozesse der Zulieferpyramide eingegangen.

In Unterkapitel 7.5 werden abschließend die Grenzen der Anwendbarkeit der Transkompilierung und Mustersuche vorgestellt. Basierend auf den aufgezeigten Grenzen werden offene Forschungspunkte vorgestellt.

7.1 Herausforderungen und Erkenntnisse

Die Entwicklung eines mechatronischen Systems erfolgt in einer Zulieferpyramide, in welcher der OEM und die Lieferanten jeweils einen eigenen Entwicklungsprozess durchlaufen. Der Testprozess ist Bestandteil des Entwicklungsprozesses und die Testprozesse sind in einer Zulieferpyramide voneinander abhängig.

Die Testprozesse innerhalb einer Zulieferpyramide sind heterogen bezogen auf *Testskriptsprachen*, Testsysteme, Teststufen und Anforderungen an den Prüfling. Die Heterogenität wird zu einer Herausforderung, sobald die Testprozesse zusam-

menarbeiten müssen. Bei der Zusammenarbeit, welche eine testprozessübergreifende Aktivität ist, müssen die Testingenieure die Ursache für das Fehlverhalten des Prüflings anhand des fehlgeschlagenen Testfalls finden. Hierfür hat die Dissertation das Konzept der kollaborativen Fehlersuche und die darin enthaltenen Fehlersuchmethoden entwickelt. Die Fehlersuchmethoden benötigen als Eingabe zwei Testfälle von unterschiedlichen Testprozessen. Der fehlgeschlagene Testfall und ein vergleichbarer Testfall, der den gleichen Sachverhalt erfolgreich getestet hat, sind die zentralen Informationsquellen. Die Fehlersuche ist in den Testfällen problematisch, weil diese in unterschiedlichen *Testskriptsprachen* implementiert sind und dadurch nicht einfach für die Fehlersuche nutzbar sind. Ein Ergebnis der Untersuchung ist, dass der Zugang zu den Informationen innerhalb der Testfälle die Fehlersuche in den Testprozessen deutlich beschleunigen kann. Dies gilt jedoch nur, wenn Testfälle interoperabel sind. Dadurch könnten die Testfälle automatisiert in die Zielsprachen übersetzt und dementsprechend in die Testprozesse integriert, ausgeführt und ihre Ergebnisse verglichen werden.

Um die Austauschbarkeit der Testfälle zu gewährleisten, muss die Interoperabilität der Testfälle sichergestellt werden. Die Interoperabilität wurde im Rahmen der Dissertation für zwei Zwecke genutzt, nämlich für die **Transkompilierung** und für die **Mustersuche**. Im Folgenden werden die Erkenntnisse für beide Szenarien präsentiert.

7.1.1 Transkompilierung von Testfällen

Die Transkompilierung von Testfällen ist das Ergebnis für die Fehlersuchmethode 1. Es ermöglicht Testfälle zwischen *Testskriptsprachen* zu übersetzen. Zu diesem Zweck verwendet der Ansatz eine selbst entwickelte Transkompilierung in Kombination mit einem Informationsmodell basierend auf SCXML.

Die Ergebnisse zeigen, dass *Testskriptsprachen* einen ähnlichen Sprachumfang für eine bestimmte Anwendungsdomäne, wie z.B. HIL-Tests, haben. So bieten die Sprachen ähnliche Möglichkeiten zur Stimulation, Überwachung und Zeitsteuerung des Prüflings an, während ihre Syntax unterschiedlich sein kann. Diese semantische Ähnlichkeit wurde genutzt, um ein Informationsmodell zur Abstraktion von *Testskriptsprachen* für HIL-Tests zu spezifizieren. Das Ergebnis ist *Generic SCXML*. Die Evaluierung zeigte, dass im Rahmen der Forschungsprojekte STEVE und AGILE-VT Testfälle aus verschiedenen Testabteilungen und *Testskriptsprachen* für unterschiedliche mechatronische Systeme (Türen, Sauerstoffversorgung, Klappen) eines Flugzeugs erfolgreich in *Generic SCXML* überführt werden können. Es wurde auch gezeigt, dass Testfälle von CCDL nach *Generic SCXML*, von RTT nach *Generic SCXML* und umgekehrt übersetzt werden können. Damit ist erstmals ein interoperabler Austausch von Testfällen zwischen den Testprozessen innerhalb der Zulieferpyramide möglich. Somit wird die Fehlersuchmethode 1 der Fehlersuche im industriellen Umfeld realisierbar.

7.1.2 Mustersuche in Testfällen

Testfälle testen spezifische Funktionen eines Prüflings. Die enthaltenen Informationen, wie der Prüfling stimuliert und getestet wurde, sind spezifisch für einen Testfall und beschreiben nicht den Prüfling als Ganzes. Es besteht der Bedarf, die in Testfällen enthaltenen Informationen zu aggregieren, um eine Gesamtsicht zu erhalten. Das Ergebnis der entwickelten Lösung ist die Mustersuche für die Fehlersuchmethode 2. Die Mustersuche ermöglicht die Extraktion und Aggregation von Informationen aus Testfällen (Datensicht/ Datenträger), welche in unterschiedlichen *Testskriptsprachen* implementiert sind und in einem Wissensgraphen (Informationsmodell/ Informationsträger) aggregiert werden. Die semantische Extraktion enthält Informationen aus den Informationsgruppen *Stimulus, Verdikt* und *Laufzeitverhalten*. Dieser Informationsgehalt deckt die funktionalen Teile eines Testfalls ab. Dazu wurde der Graph G des Wissensgraphen spezifiziert und die notwendigen Transformationsfähigkeiten implementiert. Damit ist die Aggregation von Informationen aus Testfällen realisierbar und erreicht das Interoperabilitätslevel Daten/ Informationen nach ATHENA. Damit der Wissensgraph nicht zu einer weiteren Datensenke im PLM wird, wurde eine Abfragesprache entwickelt. Diese Abfragesprache ist für die explorative Suche und für die Suche nach Unterschieden in Testfällen optimiert. Darüber hinaus wurde die Abfragesprache auf den Testingenieur zugeschnitten, so dass dieser sie direkt mit seinen Fähigkeiten nutzen kann.

Die Evaluation zeigte, dass bereitgestellte Testfälle in CCDL und Python transformiert und in die Graphdatenbank Neo4J importiert werden können, wobei die funktionalen Aspekte eines Testfalls erhalten bleiben. Die Anwendbarkeit von SCXML als Informationsmodell für Testfälle wurde im Rahmen der Dissertation bestätigt. Darüber hinaus zeigte die Evaluierung, dass Informationen aus Testfällen hochgeladen, aggregiert und durchsucht werden können. Damit wurde die Interoperabilität von Testfällen für eine Suche etabliert und ermöglicht semantische Unterschiede in Testfällen zu finden. Diese Fähigkeit übersteigt deutlich die Fähigkeiten einer syntaktischen Textsuche und ermöglicht damit die Fehlersuchmethode 2 für die kollaborative Fehlersuche.

7.1.3 Beantwortung der Forschungsfrage

In dem Unterkapitel 1.4 wurde die Forschungsfrage definiert, **ob mithilfe des Austausches von Informationen aus interoperablen Testfällen der Datenintegrationsaufwand reduziert und der Nutzen der ausgetauschten Informationen für die testprozessübergreifenden Aktivitäten erhöht werden kann**.

Zur Beantwortung der Forschungsfrage wurde die kollaborative Fehlersuche als testprozessübergreifende Aktivität ausgewählt. Der Nutzen der kollaborativen Fehlersuche wurde für drei Fehlersuchmethoden in dem Kapitel 6 evaluiert. Das Ergebnis

der Evaluation ist, dass die manuellen Datenintegrationsaufwände automatisiert werden konnten und dass die Fehlersuchmethoden für alle drei Testszenarien erfolgreich Indikatoren für die Fehlerursachen in den Testprozessen finden konnten. Somit kann die Forschungsfrage hinsichtlich der Reduzierung des Datenintegrationsaufwands und der Erhöhung des Nutzens von testprozessübergreifenden Aktivitäten mit **ja** beantwortet werden.

Die Tabelle 25 beantwortet die abgeleiteten Forschungsfragen tabellarisch.

Tabelle 25: Beantwortung der abgeleiteten Forschungsfragen

Bezeichner	**Forschungsfrage**	**Antwort**	**Erklärung**
1a	Sind die Testaktivitäten der Testprozesse aus der Zulieferpyramide vergleichbar, um testprozessübergreifende Aktivitäten zwischen den Testprozessen zu ermöglichen?	Ja	Der Ablauf der Testprozesse ist identisch. Zusätzlich sind die zu prüfenden Anforderungen und die Kommunikation innerhalb des mechatronischen Systems vergleichbar.
1b	Welche Fehlersuchmethoden sind notwendig, um den Grund für die unterschiedlichen Ergebnisse der Testprozesse zu finden?	Es sind drei Fehlersuchmethoden notwendig: Fehlersuchmethode 1: Ausführung des Testfalls und Analyse des Signalverlaufs, Fehlersuchmethode 2: Mustersuche und Fehlersuchmethode 3: Anpassung der Konfiguration und anschließende Ausführung	Eine Methode ermöglicht eine dynamische Analyse für Testfälle, die zweite eine statische für Testfälle und die dritte ermöglicht die dynamische Analyse für Konfigurationen.
2a	Welche Informationen beinhaltet ein Testfall in den Testprozessen der Zulieferpyramide	Stimulus, Verdikt und Laufzeitverhalten	Ein minimalistischer Testfall enthält Anweisungen aus den genannten Informationsgruppen, welche den Prüfling beeinflussen.
2b	Welche *Testskriptsprachen* werden für den Hardware in the Loop (HIL)-Test eingesetzt und inwiefern sind sie zueinander kompatibel?	Es gibt viele Sprachen. CCDL, C, Python, RTT, TTCN-3, ASAM-XIL wurden analysiert. Die Sprachen sind eingeschränkt kompatibel untereinander	*Testskriptsprachen* sind in allen sprachlichen Merkmalen heterogen und erlauben trotzdem die gleiche Semantik der Testfälle. Betrachtet man die Testfälle semantisch, so sind sie untereinander kompatibel. Dies gilt nur für die Schnittmenge

			der Sprachumfänge der *Testskriptsprachen*.
2c	Welches Schema muss ein Informationsmodell besitzen, welches auf die Semantik und nicht auf die Syntax von Testfällen fokussiert?	Das Schema muss auf einem Zustandsübergangsdiagramm basieren	Das Schema muss einem Zustandsübergangsdiagramm folgen und in der Lage sein, die genannten Informationsgruppen darzustellen. Außerdem müssen sequenzielle und parallele Zweige unterstützt werden.

7.1.4 Abdeckung der Anforderungen durch die kollaborative Fehlersuche

Die Beantwortung der Forschungsfrage erforderte eine technische Umsetzung der kollaborativen Fehlersuche. Die für die Entwicklung benötigten funktionalen Anforderungen wurden im Kapitel 4.1 für die Interoperabilität von Testfällen und für die Ausgestaltung der drei Fehlersuchmethoden präsentiert. Die Übersicht der Anforderungen ist in der Tabelle 13 gegeben. Tabelle 26 überprüft, ob die Anforderungen durch die entwickelte kollaborative Fehlersuche erfüllt sind.

Tabelle 26: Übereinstimmung der kollaborativen Fehlersuche mit den Anforderungen

Anforderung	**Erfüllung**
Die Anweisungen sind in den Testfällen für die Anweisungstypen (*Stimulus, Laufzeitverhalten* und *Verdikt)* interoperabel austauschbar **(R1)**	Die Anforderung ist **erfüllt**, weil die Informationsmodelle SCXML und Graph G die Anweisungstypen repräsentieren können. Hierfür wurde der TASCXML Befehlssatz als Referenz genommen und implementiert.
Der Austausch der Informationen ist nicht hinreichend, sondern erst die Nutzung für die Fehlersuche **(R2)**	Die Anforderung ist **erfüllt**, weil die Fehlersuchmethode 1 einen Testfall in SCXML als Eingabe für eine Übersetzung in eine *Testskriptsprache* verwendet und die Fehlersuchmethode 2 Testfälle in SCXML in einen Graph G für eine Mustersuche importiert.
Die kollaborative Fehlersuche soll die Interoperabilität von Testfällen ermöglichen und soll darauf aufbauend drei semi-automatische Fehlersuchmethoden (Ausführung des Testfalls und Analyse des Signalverlaufs, Mustersuche und Anpassung der Konfiguration und anschließende Ausführung) implementieren **(R3)**	Die Anforderung ist **erfüllt** und drei Fehlersuchmethoden stehen über die Funktionsmuster *Test Case Translator* und *Test Advisor* zur Verfügung.

Die Eingabe für die Fehlersuchmethoden sind die Testfälle und die Konfiguration der höheren Integrationsstufe **(R4)**	Die Anforderung ist **erfüllt** und der Testfall des OEM wird mithilfe des **Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche in *Generic SCXML* transformiert und den Fehlersuchmethoden bereitgestellt.
Die Identifizierung aller Unterschiede zwischen zwei Testprozessen benötigt sowohl dynamische als auch statische Analysen **(R5)**	Die Anforderung ist **erfüllt**, weil die Fehlersuchmethode 1 eine dynamische Analyse der Testfälle, die Fehlersuchmethode 2 eine statische Analyse der Testfälle und Fehlersuchmethode 3 eine dynamische Analyse der Konfigurationen ermöglicht.
Testfälle der niedrigeren Integrationsstufe (Modul- oder Komponenten-Lieferant) werden als Eingabe interoperabel benötigt **(R6).**	Die Anforderung ist erfüllt und alle Testfälle können mithilfe des **Schritt 1: Interoperabilität der Testdaten herstellen** der Fehlersuche interoperabel dargestellt werden.

7.2 Abgleich mit bestehenden Studien

Die Inhalte des Unterkapitels wurden aus den Veröffentlichungen [Franke und Thoben 2022; Franke et al. 2023] direkt übernommen und angepasst.

Die vorgestellte Lösung konzentriert sich auf die Interoperabilität von Testfällen. Zu diesem Zweck wird *Generic SCXML* als Informationsmodell verwendet. Hiermit können Testfälle über die Grenzen einer bestimmten *Testskriptsprache* hinweg dargestellt werden. SCXML ist dabei ein vom W3C definiertes Zustandsübergangsdiagramm. Zustandsübergangsdiagramme sind ein gängiger Ansatz zur Modellierung von Testfällen als Testmodelle im Bereich des modellbasierten Testens [Franke et al. 2012; Peleska 2013]. Damit ist die Auswahl von *Generic SCXML* als Informationsmodell vollständig auf die von Testsystemherstellern bereitgestellte Werkzeuglandschaft abgestimmt. Darüber hinaus ist auch die Ausführung von Testfällen in SCXML möglich. Hierzu definiert der SCXML-Standard die Grundlage für ein Ausführungsframework. Die Integration von Testfällen in *Generic SCXML* in MBSE- und MBT-Modelle ist ebenfalls möglich, da MBSE- und MBT-Modelle Zustandsübergangsdiagramme einsetzen. Damit kann *Generic SCXML* vollständig in die Design-, Spezifikations-, Implementierungs- und Ausführungsphase eines Testprozesses integriert werden.

Die Interoperabilität von Testfällen mittels eines Wissensgraphen ist auch ausgerichtet am Stand der Technik. Das Anwendungsszenario eines Wissensgraphen dient nach [Hogan et al. 2021] dazu die Entitäten der realen Welt als ein Graph abzubilden. Der Beitrag befasst sich mit dem in Testfällen enthaltenen Informationen als real zu formalisierendes Wissen. Die realen Entitäten sind die Signale eines komplexen mechatronischen Systems, die stimuliert oder überprüft werden können. Darüber hinaus definieren die Kanten eines Graphs die zeitlichen Beziehungen zwischen den

Signalen. Damit kann das sequenzielle und parallele Verhalten von Signalen modelliert werden. Somit passt der Anwendungsbereich des Testens zu Hogans erwartetem Anwendungsszenario und seiner Definition.

Das benötigte Graphenmodell zur Formulierung der Informationen nutzt Labels von einem *directed edge-labelled graphs* [Hogan et al. 2021] und die Eigenschaften eines *property graphs* [Hogan et al. 2021]. Diese Graphtypen werden von gängigen Graphdatenbanken wie Neo4J unterstützt. Das Graphenmodell stellt die Übertragbarkeit der Forschungsergebnisse für die laufende Forschung und Kommerzialisierung sicher.

Die Integration von interoperablen Testfällen und deren Aggregation als Wissensgraph in das Supply Chain Management entspricht der Weiterentwicklung der Supply Chain hin zu digitalen Supply Chains. [Barykin et al. 2020; Wang et al. 2022]. Dies ermöglicht die gemeinsame Nutzung von Testfällen als digitale Ressourcen oder die Erweiterung des digitalen Zwillings um prozedurales Testwissen. Im letzteren Fall könnte eine Domänenontologie für Testfälle spezifiziert werden, basierend auf STOWS [Zhang und Zhu 2008], SWTO [Bezerra et al. 2009] oder OntoTest, um die Abfrage- und Integrationsfähigkeiten des Semantic Web zu ermöglichen. In beiden Fällen ist die Versionierung von gemeinsam genutzten Testfällen, die sich auch während eines Testprozesses weiterentwickeln, notwendig und wird durch das Konzept der Digital Threads abgedeckt. Dementsprechend kann das Informationsmodell einschließlich Transkompilierung in den aktuellen Stand der Technik für digitale Supply Chains integriert und um Methoden des Wissensmanagements erweitert werden.

7.3 Implikation für die Methode

Die Inhalte des Unterkapitels wurden aus den Veröffentlichungen [Franke und Thoben 2022; Franke et al. 2023] direkt übernommen und angepasst.

Die Auswirkungen der entwickelten Methoden auf die derzeit verwendeten Methoden der Transkompilierung und Mustersuche werden vorgestellt. Der Schwerpunkt liegt dabei auf der Transkompilierung und der Mustersuche, da diese als kollaborativer Suchansatz direkt die Interoperabilität von Testfällen widerspiegeln.

Transkompilierung

Die Heterogenität der *Testskriptsprachen* in Bezug auf Typsicherheit, angewandte Programmierparadigmen und Funktionalität stellt eine Herausforderung für die Transkompilierung als semantische Datenintegration dar. Aus theoretischer Sicht ist ein Testfall ein in einer bestimmten Programmiersprache geschriebenes Programm.

Es wird angenommen, dass die Übersetzung von Quellcode zwischen Programmiersprachen, die Turing-vollständig ist, möglich ist. [Lachaux et al. 2020]. Dies beweist, dass die Übersetzung zwischen *Testskriptsprachen* möglich ist. Der Nachweis, dass Programme zwischen ähnlichen Programmiersprachen korrekt übersetzt werden können, wird erbracht in [Lachaux et al. 2020]. Die verfügbaren Tools verwenden dazu Transcompiler, die eine Source-to-Source-Übersetzung ermöglichen. Die Fehlersuchmethode 1 zielt darauf ab, Transcompiler auch für *Testskriptsprachen* zu verwenden, die nicht ähnlich sind. Dabei gibt es zwei Unterschiede zwischen der entwickelten Methode und dem Stand der Technik.

Der erste Unterschied besteht darin, dass im Kontext der Arbeit nicht für jedes Tupel von *Testskriptsprachen* ein Transcompiler entwickelt wurde. Das heißt, es wurde zum Beispiel kein Transcompiler für (CCDL <-> RTT, Flatscript <-> CCDL usw.) entwickelt. Der Grund dafür ist, dass der Wartungsaufwand für eine spätere Integration neuer *Testskriptsprachen* oder neuer Versionen bereits unterstützter *Testskriptsprachen* zu hoch wäre. Anstatt für jede Sprache einen 1:1 Transcompiler zu entwickeln, fokussiert der Ansatz auf zwei Transcompiler für jede neue Sprache. Der erste Transcompiler übersetzt von der *Testskriptsprache* nach SCXML und der zweite Transcompiler übersetzt von SCXML in die Zielsprache. Dieser Ansatz gewährleistet eine kontinuierliche Integration neuer *Testskriptsprachen*. Dazu verfügen diese Compiler über einen Encoder und Decoder für die jeweiligen Sprachen sowie die Verwendung von *Generic SCXML* als Informationsmodell zur Darstellung der Testfallinhalte. Die Fehlersuchmethode 1 ist vollständig auf diesen Ansatz ausgerichtet, wobei der Decoder die Kombination aus *ParseTree/Timeline* und der Encoder der *SCXML-Generator* ist.

Der zweite Unterschied besteht darin, dass die Übersetzung eines Testfalls zwischen *Testskriptsprachen* nicht die gesamte Semantik abdeckt. Der Teil, der von der Struktur der Testskriptsprache benötigt wird, wird ignoriert. Das bedeutet, dass der Decoder den Informationsgehalt reduziert. Diese Lösung führt dementsprechend zu keiner 1:1-Übersetzung. Stattdessen zielt diese Lösung darauf ab, die funktionale Sicht des Testfalls zu übersetzen, aber nicht die notwendige Syntax und Syntaxerweiterungen. Der Fokus lag auf einer Teilmenge von Anweisungen, die sich auf die Anweisungstypen konzentrieren. Die Reduzierung stellt jedoch kein Problem dar, da die Testskript-spezifische Syntax vom Encoder hinzugefügt wird, wenn der Testfall für eine bestimmte Testfallsprache rückübersetzt wird.

Mustersuche

Eine Mustersuche für Testfälle nutzt heute die Suchmöglichkeiten für Programmiersprachen, die in Entwicklungsumgebungen integriert sind. Derzeit sind eine Schlagwortsuche und eine syntaxbasierte Suche integriert. Bei der Schlagwortsuche wird nach dem Wort oder dem regulären Ausdruck im Quellcode gesucht. Im Gegensatz

dazu wird bei der syntaxbasierten Suche die Deklaration, Definition oder Verwendung einer Anweisung durchsuchbar gemacht. Hierbei ist zu beachten, dass eine Anweisung aus mehreren Zeilen bestehen kann. Bei diesen Arten der Suche liegt der Fokus der Suche auf einer Anweisung. Die entwickelte Mustersuche erweitert den Umfang von einer Anweisung auf ein Muster, welches aus mehreren Anweisungen bestehen kann. Außerdem arbeitet die Suche nicht auf der Syntax der Programmiersprache, sondern auf dem Graphen G, der die Eigenschaften der Programmiersprache abstrahiert. Dadurch wird eine Suche ermöglicht, die sich auf die Bedeutung einer Anweisung oder auf die Bedeutung einer Folge von Anweisungen für den Testfall konzentriert. Damit werden die Suchmöglichkeiten des Standes der Technik um semantische Suchmöglichkeiten erweitert.

7.4 Implikationen für die Praxis

Die Inhalte des Unterkapitels wurden aus den Veröffentlichungen [Franke und Thoben 2022; Franke et al. 2023] direkt übernommen und angepasst.

Ziel war es, die Zusammenarbeit zwischen den Testprozessen innerhalb der Zulieferpyramide mittels einer kollaborativen Fehlersuche zu verbessern. Die zu lösende Herausforderung war die fehlende Interoperabilität von Testfällen zwischen Testprozessen. Bisher gab es keine interoperablen Testfälle innerhalb der Zulieferpyramide und deswegen war eine automatische Integration von Testfällen verschiedener Lieferanten nicht möglich. Daher war die Integration von Testfällen aus verschiedenen *Testskriptsprachen* innerhalb der Zulieferpyramide eine manuelle Aufgabe und erforderte bisher einen Experten in der jeweiligen *Testskriptsprache*. Dieses Fachwissen ist notwendig, da die Testingenieure verstehen müssen, wie der Testfall in einer unbekannten *Testskriptsprache* den Prüfling getestet hat. Dazu muss er durch das Lesen des Testfalls (statische Analyse) verstehen, wie der Prüfling stimuliert wurde und was als korrektes Verhalten definiert wurde.

Zusätzlich zur statischen Analyse, basierend auf Textsuchen, müssen die „fremden“ Testfälle aus nicht unterstützten *Testskriptsprachen* auf dem Prüfstand ausgeführt werden, um mögliche Verzögerungen oder Zeitverschiebungen in der Stimulation oder der Reaktion des Prüflings zu identifizieren. Dazu musste der Testfall manuell in die Testsprache des Lieferanten übersetzt, ausgeführt und mit den Testergebnissen des Lieferanten verglichen werden, was einen hohen manuellen Aufwand durch Experten erforderte. Die Notwendigkeit, diesen manuellen Aufwand durch einen automatisierten Ansatz zu ersetzen, hängt von der Häufigkeit ab, mit der Testfälle zwischen Anbietern ausgetauscht werden müssen. Der Entwurf von Autos und Flugzeugen wird immer komplexer und hierbei steigt der Anteil von Elektronik und Soft-

ware. Hierdurch werden Testprozessen ebenfalls komplexer und damit fehleranfälliger. Daher sind die Auswirkungen einer möglichen Verbesserung der Testautomatisierung in der Praxis sehr hoch.

Der Lösungsvorschlag ermöglicht in der Praxis die Interoperabilität von Testfällen in den Testprozessen der Zulieferpyramide. Dies bedeutet, dass ein Testfall in einer *Testskriptsprache* A ohne manuellen Aufwand und ohne Experten in eine *Testskriptsprache* B übersetzt werden kann. Dies erleichtert die kollaborative Fehlersuche im Rahmen der Zusammenarbeit von Testprozessen, da der Austausch des Testfalls, die Übersetzung des Testfalls, die anschließende Ausführung und der Vergleich der Testberichte automatisiert werden können. Im Rahmen dieser Dissertation wurde der Schritt der Testfallübersetzung gelöst. Die anderen Schritte für die Fehlersuchmethode 1 sind bereits vorhanden, da die Testmanagementwerkzeuge diese bereits bereitstellen können.

Übersetzungsfähigkeiten können überall dort bereitgestellt werden, wo ein komplexes mechatronisches Produkt innerhalb einer Supply Chain entwickelt wird. Hierfür kann die Testfallübersetzung in der Praxis bei der Entwicklung von Flugzeugen eingesetzt werden, da diese auf HIL-Tests angewiesen sind. Die Interoperabilität von Testfällen ermöglicht nicht nur die Transkompilierung für die Fehlersuchmethode 1, sondern auch die Aggregation von Testfällen zu prozeduralem Testwissen. Dadurch wird ein breites Spektrum von informationsgesteuerten Diensten ermöglicht. Die existierenden Datenquellen der Testprozesse enthalten bereits unzählige Testfälle unterschiedlicher Flugzeugtypen vom A220 bis A380 und entsprechenden Varianten. Während sich die verwendeten *Testskriptsprachen* im Laufe der Zeit geändert haben, haben sich die Grundlagen der Flugdynamik und der Steuerung eines Flugzeugs nicht geändert. Das heißt, Testwissen für gemeinsame übliche Flugmanöver inklusive des richtigen Stimulus und des erwarteten Verhaltens ist bereits vorhanden und kann durch die entwickelte Mustersuche als Bestandteil der Fehlersuchmethode 2 oder eines zukünftigen informationsgesteuerten Dienstes durchsuchbar & anwendbar gestaltet werden. Dabei ermöglicht die entwickelte Methode trotz der Heterogenität von *Testskriptsprachen* die Aggregation von Testfällen und speist Testwissen für informationsgetriebene Dienste ein.

7.5 Einschränkungen und zukünftige Forschung

Die Inhalte des Unterkapitels wurden aus den Veröffentlichungen [Franke und Thoben 2022; Franke et al. 2023] direkt übernommen und angepasst.

Der vorgeschlagene Ansatz ermöglicht die Interoperabilität von Testfällen für HIL-Tests mechatronischer Systeme. So werden Testfälle unterstützt, bei denen der Prüfling durch Signale stimuliert und anschließend auf korrektes Verhalten überprüft wird. Dabei gibt es keine Einschränkungen hinsichtlich des Typs (sicher, unsicher)

und des gewählten Programmierparadigmas der *Testskriptsprache*. Theoretisch kann jede verfügbare und zukünftige *Testskriptsprache* unterstützt werden. Aus praktischer Sicht verfügt die vorgeschlagene Lösung über einen *Testskriptsprachen*-spezifischen Decoder und Encoder, der die Übersetzung von einem Parsebaum in eine Zeitachse und die Übersetzung von einem SCXML-Testfall in eine spezifische *Testskriptsprache* implementiert. Das bedeutet, dass für jede neu unterstützte Sprache diese sprachspezifischen Softwaremodule implementiert werden müssen. Die Integration neuer Sprachen erfordert keine weiteren Forschungsaktivitäten, sondern Softwareentwicklungsprojekte können die Integration auf einfache Weise durchführen. Die derzeitige Einschränkung der Funktionsmuster ist die Unterstützung der *Testskriptsprachen* CCDL und Python, wobei Python als RTT durch testsystemspezifische Bibliotheken spezialisiert wurde. Der vorgestellte Ansatz und die Funktionsmuster reflektieren nicht die tatsächliche Bereitstellung übersetzter Testfälle im operativen Betrieb der Supply Chain. Hierfür müssen neue digitale Prozesse, Rechte- und Rollenkonzepte, Urheberrechte und andere Überlegungen entwickelt werden. Die folgenden Forschungsfragen beziehen sich auf digitalisierte Supply Chains und befinden sich in der Entwicklung [Barykin et al. 2020; Wang et al. 2022] [Zhu et al. 2022].

Das Übersetzen von Testfällen aus verschiedenen *Testskriptsprachen* in dasselbe Informationsmodell ermöglicht die Erstellung eines Wissensgraphs, der von Testfällen gespeist wird. Dieser Wissensgraph könnte die heterogenen Silos von Testfällen ersetzen, die über die Testprozesse der Supply Chain verteilt sind. Die Erstellung eines solchen Wissensgraphs würde das implizite Wissen innerhalb des Testfalls sammeln, wie ein bestimmtes System aus verschiedenen Perspektiven stimuliert und überprüft werden kann. Die Grenze des aktuell unterstützten Wissensgraphs ergibt sich aus der Abdeckung der übersetzbaren funktionalen Aspekte des Testfalls durch *Generic SCXML*. Ändert sich die Definition der TASCXML-Befehle als Bestandteil von *Generic SCXML*, muss auch der Graph G inklusive der Transformationsfunktionen f & f2 angepasst werden. Testfälle, die in Excel-Tabellen oder in natürlicher Sprache geschrieben sind, sind manuelle Tests und können nicht mit den Fehlersuchmethoden in Graph G importiert werden. Zu diesem Zweck ist ein vorbereitender Schritt 0 mit Natural Language Processing (NLP) erforderlich, um die Testfälle in eine *Testskriptsprache* zu übersetzen. Abgesehen von der technischen Einschränkung müssen die Rechte am geistigen Eigentum (IPR) und das Nutzungsrecht berücksichtigt werden. Der OEM und alle Lieferanten entwickeln Testfälle in unterschiedlichen *Testskriptsprachen*, und erst die Aggregation aller Testfälle als Wissensgraph würde das vollständige Bild des mechatronischen Systems als Produkt ergeben. Die Herausforderung besteht darin, dass das geistige Eigentum jedes hochgeladenen Testfalls unabhängig von seiner Darstellungsform berücksichtigt werden muss. Das bedeutet, dass der Wissensgraph eine Reihe von IPRs berücksichtigen und ein entsprechendes Rechte- und Rollenkonzept implementieren muss. Ein weiterer Aspekt sind die europäischen Regelungen zum Urheberecht. Grundsätzlich gibt es kein Dateneigentumsrecht in der Europäischen Union [Duch-Brown et al. 2017].

Hierbei wäre zu klären ob die Testfälle in SCXML oder im Graph G, welche verarbeitet wurden als Daten im Sinne der Europäischen Union zu klassifizieren wären.

Die oben beschriebenen Umstände bestimmen maßgeblich, inwieweit die entwickelte kollaborative Fehlersuche im operativen Geschäft des Supply Chain Managements eingesetzt werden kann. Der Bedarf für informationsgetriebene Dienste ist im Tagesgeschäft der Testingenieure gegeben. Beispielsweise könnte ein Testingenieur nach einem gegebenen Signal in dem Wissensgraph suchen und seine Stimulierung explorieren. Ein Testingenieur könnte ebenfalls den Wissensgraphen fragen, wie ein bestimmtes Signal zu prüfen ist. Die Schaffung dieses Wissensgraphs und die anschließende Entwicklung von informationsgetriebenen Diensten sind als zukünftige Forschung geplant. Der Wissensgraph ist auch auf andere Phasen des PLM anwendbar. Im MOL könnte die Bereitstellung von Testwissen auch die vorausschauenden Wartungsaktivitäten verbessern [Thoben et al. 2018]. Auch das Forschungsfeld der digitalen Zwillinge könnte von der Integration von Testwissen zur Bestimmung von erwartetem und auffälligem Verhalten profitieren. Bei digitalen Zwillingen von Flugzeugen lassen sich bereits erste Schritte zur Verwertung von Konstruktionswissen erkennen [Aydemir et al. 2020].

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel der Dissertation war die Entwicklung einer kollaborativen Fehlersuche als testprozessübergreifende Aktivität. Die Arbeit konzentrierte sich auf die Ausgangssituation, dass zwei Testprozesse beim Testen der gleichen Produktfunktion zu unterschiedlichen Ergebnissen kommen und die Ursache dafür identifiziert werden muss. Ziel der kollaborativen Fehlersuche war es, Indikatoren zu identifizieren, warum die Testprozesse zu unterschiedlichen Ergebnissen gekommen sind.

Die Identifikation der Indikatoren erforderte ein detailliertes Verständnis der Testprozesse in der Zulieferpyramide. Dazu wurde **im Kapitel 2** der Entwicklungsprozess eines komplexen mechatronischen Produktes dargestellt und die Einflüsse der Anforderungsanalyse und der Eigenschaften des mechatronischen Systems auf die Testprozesse untersucht. Das Ergebnis der Untersuchung ist, dass die Testprozesse in der Zulieferpyramide nicht unabhängig voneinander sind. Vielmehr führt die Anforderungsanalyse innerhalb der Zulieferpyramide dazu, dass sich die Anforderungen vom OEM über die Modul-Lieferanten bis hin zu den Komponenten-Lieferanten immer weiter spezialisieren und somit eine globale Baumstruktur der Anforderungen entsteht. Innerhalb eines Stranges der Baumstruktur testen Testprozesse ähnliche Anforderungen, was auch zu ähnlichen Testzielen und entsprechenden Testfällen als Nachweismittel führt. Weiterhin wurde der Einfluss der Struktur eines mechatronischen Systems auf die Testprozesse untersucht. Das Ergebnis war, dass mechatronische Systeme diskrete, miteinander kommunizierende Systeme sind und diese Eigenschaft zu ähnlichen Testfällen für das Testen der Kommunikationsschnittstellen führt. Die Abhängigkeit der Testprozesse wurde als lose Kopplung eingeführt und diese Eigenschaft unterstützt die Fehlersuche bei unterschiedlichen Testergebnissen. Für die Suche wurden fehlgeschlagene Testfälle und ähnliche, erfolgreiche Testfälle als wichtige Informationsträger identifiziert und deren Austausch und Integration als obligatorisch eingestuft.

Der Austausch und Integration von Testfällen zwischen den *lose gekoppelten* Testprozessen ist aufgrund der Heterogenität der *Testskriptsprachen* nicht möglich. **Im Kapitel 3** wurde zunächst herausgearbeitet, welche Inhalte eines Testfalls als Informationen relevant sind und wofür diese in einem minimalistischen Testfall benötigt werden. Anschließend wurde untersucht, wie heterogen diese Informationen in den *Testskriptsprachen* kodiert sind. Dazu wurde zunächst die Heterogenität in der Struktur der *Testskriptsprachen* untersucht. Darauf aufbauend wurde untersucht, welche Funktionen sie für die Implementierung von Testfällen bereitstellen. Das Ergebnis der Literaturrecherche ist, dass sich die Funktionen in den *Testskriptsprachen*

bezüglich der lexikalischen Merkmale, der syntaktischen Merkmale, der morphologischen Merkmale und der semantischen Merkmale unterscheiden. Dementsprechend wird für den Informationsaustausch eine semantische Datenintegration benötigt. Anschließend wurde die Eignung von generischen Ansätzen zur Datenintegration und von Transcompilern untersucht. Beide Lösungsansätze können die Informationen aus den Testfällen nicht vollständig extrahieren.

Im Kapitel 4 der Arbeit wurde das Konzept der kollaborativen Fehlersuche unter der Annahme spezifiziert, dass die Interoperabilität mit dem in der Dissertation entwickelten Lösungsansatz möglich ist. Die entwickelte kollaborative Fehlersuche ermöglicht im ersten Schritt die Interoperabilität von Testfällen. Im zweiten Schritt der Fehlersuche wurden drei Fehlersuchmethoden entwickelt, die eine dynamische und statische Analyse der Testfälle ermöglichen. Dabei wurden die Indikatoren herausgearbeitet, welche Ursachen der fehlgeschlagene Testprozess haben kann und welche Fehlersuchmethode welche Indikatoren erkennen kann.

Im Kapitel 5 der Arbeit wurde die softwaretechnische Umsetzung für die Interoperabilität der Testfälle und für die Fehlersuchmethoden spezifiziert und prototypisch als Funktionsmuster (*Test Case Translator*, *Test Advisor*) entwickelt. Dabei wurde *Generic SCXML* als Informationsmodell für Testfälle und der Graph G für eine Mustersuche spezifiziert. Basierend auf *Generic SCXML* und Graph G wurden Transformationsmethoden für die Übersetzung entwickelt. Weiterhin wurde im Kontext der Fehlersuchmethode 3 ein Informationsmodell für das Testmanagement entwickelt, mit der die Testausführung für einen Testplan und seine generierten Variationen automatisiert verwaltet und ausgeführt werden kann**.**

Im Kapitel 6 der Arbeit wurden die drei Fehlersuchmethoden an drei Anwendungsfällen aus den Testprozessen eines Flugzeugs evaluiert. Hierbei stand die Evaluierung der entwickelten Informationsmodelle und Algorithmen im Vordergrund. Das Ergebnis ist, dass die Informationsmodelle, die Transformationsmethoden und die Such- und Generierungsmethoden auf den durch die Forschungsprojekte STEVE und AGILE-VT zur Verfügung gestellten Testfällen und Konfigurationen funktionieren. Damit sind die Funktionsmuster für die Anwendung in **Schritt 1: Interoperabilität der Testdaten herstellen** und die drei Fehlersuchmethoden in **Schritt 2: Adaptive Fehlersuche auf Testdaten ausführen** der kollaborativen Fehlersuche einsatzbereit.

Im Kapitel 7 der Arbeit erfolgte eine kritische Diskussion der Forschungsergebnisse und der Anwendbarkeit in der Praxis. Dabei wurde diskutiert, dass die entwickelten Funktionsmuster zwar funktionieren, aber nicht direkt in die Praxis übernommen werden können. Zum einen müssen zur Unterstützung zukünftiger Testprozesse sprachspezifische Softwaremodule für die jeweiligen *Testskriptsprachen* entwickelt werden. Zum anderen ist der TASCXML-Befehlssatz auf die in den Forschungspro-

jekten STEVE und AGILE-VT betrachteten Testprozesse zugeschnitten. In der Praxis werden aufgrund der beschriebenen Heterogenität der Testprozesse weitere TASCXML-Befehle benötigt. Dementsprechend müsste SCXML als Informationsmodell auf Testfälle fallbasiert erweitert und die Funktionsmuster entsprechend angepasst werden. Eine weitere Herausforderung ergibt sich aus dem IPR und den abgeleiteten Nutzungsrechten für die Aggregation und Nutzung von Testfällen. Diese müssten entwickelt und entsprechend in die Funktionsmuster integriert werden. Abgesehen von den zu lösenden Herausforderungen für die Umsetzung in die Praxis erweitern die entwickelten Informationsmodelle und Transformationsanwendungen den Stand der Technik.

8.2 Ausblick

Der Ausblick der Dissertation besteht aus drei Teilen. Zunächst werden potenzielle Verbesserungen in den Methoden der Informationsmodellierung und den entsprechenden Transformationsfähigkeiten aufgeführt. Ziel ist es, die Wartbarkeit und Weiterentwicklung domänenspezifisch zu ermöglichen. Anschließend werden potenzielle Verbesserungen für die Generierung eines direkten Mehrwerts in den Funktionsmustern aufgeführt, die für eine Umsetzung in der Praxis relevant sind. Abschließend werden ergänzende Forschungsthemen aufgeführt, die im Rahmen einer Industrialisierung noch zu untersuchen sind.

8.2.1 Potenzielle Verbesserungen der Methode

Die Komponenten der kollaborativen Fehlersuche wurden einzeln erfolgreich evaluiert. Somit ist der Nachweis für die Algorithmen und die darin eingebetteten Informationsmodelle erbracht. Die Evaluation der Fehlersuchmethoden im Verbund als kollaborative Fehlersuche ist offen. Dementsprechend müsste die Ausführung der Fehlersuchmethoden nacheinander und die anschließende Auswertung aller gesammelten Indikatoren im **Schritt 3: Erkenntnisse aus den Fehlersuchmethoden ziehen** erfolgen. Hierfür müsste im nächsten Schritt ein geeignetes Szenario ausgesucht und die kollaborative Fehlersuche ausgeführt werden. Basierend auf den Evaluationsergebnissen werden dann mögliche Verbesserungen abgeleitet.

Die Anpassung der Informationsmodelle und Algorithmen ist für die Fehlersuchmethoden 1 & 2 notwendig, sobald neue Domänen oder neue Sprachmerkmale der *Testskriptsprachen* erschlossen werden. Eine Anpassung erfordert umfangreiche manuelle Entwicklungsaktivitäten, die geradlinig, aber aufwendig sind. Die Annahme ist, dass sich die manuellen Entwicklungsaufwände durch die Automatisierung deutlich reduzieren lassen.

Der entstehende Entwicklungsaufwand müsste durch ein übergeordnetes Framework reduziert werden können, welches die notwendigen Ergänzungen im Quellcode der Funktionsmuster automatisch generieren kann. Vereinzelt ist dies bereits Bestandteil der Funktionsmuster, indem Lexer, Parser für die *Testskriptsprachen* und die Mustersprache auf Basis von Grammatiken automatisch generiert werden können. Das übergeordnete Framework würde für diese Aufgabe ein Metamodell und entsprechende Abbildungen benötigen. Erfolgreiche Frameworks für die Generierung von Modellierungs- und Ausführungsumgebungen mit ähnlicher Funktionalität sind verfügbar. Ein Beispiel ist das Graphical Modeling Framework (GMF) [projects.eclipse.org 2013], mit dem Modellierungswerkzeuge spezifiziert und anschließende die Software als Quellcode generiert werden kann. Dementsprechend ist die Entwicklung eines solchen Framework möglich und würde die Wartbarkeit der kollaborativen Fehlersuche bezogen auf die entstehenden Aufwände verbessern.

8.2.2 Potenzielle Verbesserung der Funktionsmuster

Die Komponenten der kollaborativen Fehlersuche enthalten sowohl generische als auch spezifische Lösungskomponenten, welche die *Testskriptsprache* und deren Sprachumfang reflektieren. Die Unterstützung weiterer *Testskriptsprachen* ermöglicht die Integration und Evaluation der Funktionsmuster in weiteren Testprozessen der Zulieferpyramide. Hierdurch können beispielsweise weitere obligatorische TASCXML-Befehle entdeckt werden und die Interoperabilität der Testprozesse mittelfristig in der gesamten Zulieferpyramide erreicht werden. Basierend auf der Interoperabilität ergeben sich weitere Nutzungsszenarien aus der Gruppe der datengetriebenen Dienste. Ein Beispiel hierfür ist das Befüllen des Wissensgraphen mit Testfällen von weiteren Testprozessen der Zulieferpyramide. Hierdurch würde sich über die Zeit eine ganzheitliche Sicht auf den Prüfling ergeben, was weitere Analysen für die Optimierung ermöglicht. Beispielsweise könnte die Mustersuche mittelfristig Muster nicht nur zwischen Testfällen, sondern innerhalb von Testfällen der gesamten Zulieferpyramide finden. Solche Muster würden sich für die Modularisierung und anschließende Wiederverwendung als Makros oder TASCXML-Befehle eignen. Die Suche nach solchen Mustern ist zurzeit nicht über die Grenzen einer *Testskriptsprache* möglich, wäre aber eine sinnvolle Erweiterung. Hierfür könnten Suchen für die Identifizierung von isomorphen Graphen integriert werden.

8.2.3 Komplementäre Forschungsbereiche

Die entwickelte kollaborative Fehlersuche nutzt die Interoperabilität von Testfallen als Ansatz für die drei Fehlersuchmethoden. Hierbei erfolgt die Öffnung der Datensilos von Testfällen durch die Erstellung und Befüllung von verschiedenen Informationsmodellen. Das Konzept der kollaborativen Fehlersuche sieht die Aggregation und Integration der Testdaten als Informationen ohne weitere Einschränkungen in

der Zulieferpyramide vor. Natürlich ließe sich der Mehrwert der Fehlersuchmethoden auch monetarisieren. Die Bereitstellung solcher Informationen eröffnet neue datengetriebene Geschäftsfelder und hierdurch werden Daten zur Schlüsselressource. [Arbeitsgruppe für Supply Chain Services des Fraunhofer IIS 2023].

Dazu wäre die Teilnahme der Testprozesse an Marktplätzen für Daten notwendig, die ein Ökosystem für den Handel mit Daten darstellen. Die Angebote reichen hierbei von der Bereitstellung von Daten bis hin zum Angebot von sogenannten *Data Products*, die bereits Daten analysieren und die gewonnenen Erkenntnisse als Informationen verkaufen. Sowohl die erste Option mit Testfällen in Form von SCXML Modellen als auch die zweite Option mithilfe der *Indikatoren* als Ergebnis der Fehlersuchmethoden wäre mit dem Konzept der kollaborativen Fehlersuche möglich. Für die Teilnahme an Data Marketplace sind sowohl technische als auch konzeptionelle Hürden zu überwinden. Im Folgenden sind Beispiele für technischen Hürden (Industrial Data Space (IDSA)) und konzeptionellen Hürden (IPR von testrelevanten Informationen) aufgeführt.

8.2.3.1 Integration von Testprozessen in IDSA

IDSA definiert eine Referenzarchitektur (IDS-RAM), welche den Austausch von Daten ermöglicht und dabei die Souveränität der Daten gewährleistest [IDSA 2018]. Im Rahmen der Architektur werden Schnittstellen definiert, wie der Datenaustausch mithilfe von Informationsmodellen erfolgen kann. Hierbei ist eine Voraussetzung, dass nur zertifizierte Partner Daten austauschen dürfen. Diese Voraussetzung kann nur durch die Integration eines IDS Connector [International Data Spaces 2022] in die Testsysteme oder in die Werkzeuge des Testmanagements erfüllt werden. Der Austausch der Informationen könnte hierbei weiterhin über die entwickelten Informationsmodelle SCXML oder den Graph G erfolgen, weil IDSA domänenspezifische Informationsmodelle unterstützt.

8.2.3.2 Berücksichtigung der Rechte am geistigen Eigentum

Die Testfälle enthalten die Informationen wie ein mechatronisches Produkt, wie ein Flugzeug oder Auto, zu stimulieren ist und außerdem, wie das korrekte Verhalten anschließend auch überprüft werden kann. Hierüber lassen sich beispielsweise die Anforderungen und auch Teile der Spezifikation des Flugzeugs ableiten. Dementsprechend sind Testfälle für den Testprozess sehr wichtig, aber auf der anderen Seite auch sehr schützenswert. Die entwickelte kollaborative Fehlersuche vernachlässigt den Schutz der Testfälle, indem die Inhalte der Testfälle ohne weitere Schutzmechanismen in die Informationsmodelle transformiert und entsprechend als Informationsquelle genutzt werden können. Hierbei stellt sich aus juristischer Sicht anschlie-

ßend die Frage, wie die Rechte am geistigen Eigentum (IPR) und ein darauf aufbauendes Nutzungsrecht berücksichtigt werden können. Das Nutzungsrecht für einen Nutzer ist hierbei nicht eine binäre Entscheidung. Vielmehr muss das Nutzungsrecht über den Datensatz und die darauf aufbauenden Analysefunktionen definiert werden. Beispielsweise könnte der Modul-Lieferant seine Testfälle für den Komponenten-Lieferanten anbieten, um Unterschiede in den Stimuli zu suchen. Der Modul-Lieferant hätte aber kein Interesse daran, dass die Exploration der Wertebereiche eines Signals möglich ist. Beide Funktionen könnten über denselben Datensatz ermittelt werden. Dementsprechend muss ein komplexes Eigentums- und Nutzungskonzept für den Umgang mit Informationen aus Testfällen entwickelt werden. Die Grundlagen hierfür werden bereits im Rahmen von Gaia-X [BMWi und BMBF 2019] erarbeitet.

9 Literaturverzeichnis

Arbeitsgruppe für Supply Chain Services des Fraunhofer IIS: *Datengetriebene Geschäftsmodelle*. WWW-Seite. https://internationaldataspaces.org/wp-content/uploads/Whitepaper-2018.pdf. Besucht am: 30 Januar 2023.

ASAM: *ASAM XIL*. WWW-Seite. https://www.asam.net/standards/detail/xil/. Besucht am: 3 Februar 2023.

Aydemir, H., Zengin, U., Durak, U.: The Digital Twin Paradigm for Aircraft Review and Outlook. In: *AIAA Scitech 2020* 2020 (2020), S.553.

Barbosa, E.F., Nakagawa, E.Y., Maldonado, J.C.: Towards the Establishment of an Ontology of Software Testing, In: *Seke* (2006), S. 522–525.

Barykin, S.Y., Bochkarev, A.A., Kalinina, O.V., Yadykin, V.K.: Concept for a Supply Chain Digital Twin. In: *Int J Math, Eng, Manag Sci* 5 (2020), S.1498–1515.

Bellinger, G., Castro, D., Mills, A.: *Data, information, knowledge, and wisdom*. WWW-Seite. https://homepages.dcc.ufmg.br/~amendes/SistemasInformacaoTP/TextosBasicos/Data-Information-Knowledge.pdf. Besucht am: 14 März 2023.

Berner, S.: *Informationsmodellierung: durch Verstehen zu besserer Software.* vdf Hochschulverlag AG 2016.

Bezerra, D., Costa, A., Okada, K.: SWTOI (software test ontology integrated) and its application in Linux test, In: *International Workshop on Ontology, Conceptualization and Epistemology for Information Systems, Software Engineering and Service Science* (2009), S. 25–36.

BIBA - Bremer Institut für Produktion und Logistik 2018. STEVE : Schlussbericht für den Zeitraum 15.08.2014-30.09.2017, 10th ed., [Bremen]. In: https://doi.org/10.2314/GBV:102515116X.

Bolton, W.: *Mechatronics*: *Electronic control systems in mechanical engineering,* 3rd ed. Addison Wesley Longman, Harlow 1997, 380 Seiten.

Bourns: *Bourns - Commercial Vehicle Sensors*. WWW-Seite. https://www.bourns.com/products/automotive/commercial-vehicle-sensors. Besucht am: 17 Januar 2023.

Broy, M. (Ed.): *Model-based testing of reactive systems*: *Advanced lectures.* Springer, Berlin 2005, 659 Seiten.

Busch, A., Dangelmaier, W. (Eds.): *Integriertes Supply Chain Management: Theorie und Praxis effektiver unternehmensübergreifender Geschäftsprozesse.* Gabler Verlag, Wiesbaden 2002.

Campetelli, A., Broy, M.: Modelling Logical Architecture of Mechatronic Systems and Its Quality Control. In: *Winner, H., Prokop, G., Maurer, M.* (Eds.) *Automotive Systems Engineering II*. Springer International Publishing, Cham (2018), S. 73–91.

Deng, Q., Franke, M., Lejardi, E.S., Rial, R.M., Thoben, K.-D.: Development of a Digital Thread Tool for Extending the Useful Life of Capital Items in Manufacturing Companies - an Example Applied for the Refurbishment Protocol, In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE (2021), S. 1–8.

Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review, In: *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (2007), S. 31–36.

Dong, X.L., Naumann, F.: Data fusion: resolving data conflicts for integration. In: *Proceedings of the VLDB Endowment* 2 (2009), S.1654–1655.

dSPACE: *Real-Time Testing*. WWW-Seite. https://www.dspace.com/de/gmb/home/products/sw/test_automation_software/automated_test_execution_in_re.cfm. Besucht am: 3 Februar 2023.

dSPACE GmbH: 2021. AGILE-VT-dSPACE - agiles virtuelles Testen der nächsten Generation für die Luftfahrtindustrie von dSPACE im Verbund AGILE-VT: *Schlussbericht - öffentlich : BMWi-Verbundprojekt im Rahmen des Luftfahrtforschungsprogramms LuFo V-3 : Berichtszeitraum: 1. Oktober 2017-30. Juni 2021,* Paderborn.

Duch-Brown, N., Martens, B., Mueller-Langer, F.: The Economics of Ownership, Access and Trade in Digital Data. In: *JRC Digital Economy Working Paper 2017-01* (2017). https://doi.org/10.2139/ssrn.2914144.

DUDEN: *transformieren*. WWW-Seite. https://www.duden.de/rechtschreibung/transformieren. Besucht am: 31 Januar 2023.

Endres, A.: Die Wissensgesellschaft und ihr Bezug zur Informatik. In: *Informatik-Spektrum* 26 (2003), S.195–200.

Fette, M., Herrmann, A.: Zivile Luftfahrtindustrie im Wandel der Digitalisierung. In: *Sonderprojekte ATZ/MTZ* 22 (2017), S.38–41.

Focus Online: *Auto: Aus wie vielen Einzelteilen besteht ein Auto?* WWW-Seite. https://www.focus.de/auto/ratgeber/auto-abc/auto-aus-wie-vielen-einzelteilen-besteht-ein-auto_id_3514353.html. Besucht am: 7 Februar 2023.

Franke, M., Gerke, D., Hans, C., Thoben, K.: Functional System Verification, In: *Air Transport and Operations: Proceedings of the Third International Air Transport and Operations Symposium 2012* (2012), p. 36.

Franke, M., Hribernik, K., Thoben, K.-D.: Semantic Interoperability for Logistics and Beyond. In: *Freitag, M.* (Ed.) *Dynamics in Logistics*. *Twenty-Five Years of Interdisciplinary Logistics Research in Bremen, Germany*. Springer International Publishing AG, Cham (2021), S. 109–128.

Franke, M., Hribernik, K.A., Thoben, K.-D.: An approach to support reliable test processes between suppliers and OEM. In: *Procedia Manufacturing* 16 (2018), S.83–90.

Franke, M., Hribernik, K.A., Thoben, K.-D.: Interoperable Access to Heterogeneous Test Knowledge. In: *TESConf 2020 - 9th International Conference on Through-life Engineering Services* 2020 (2020a), https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3717723.

Franke, M., Hribernik, K.A., Thoben, K.-D., Hans, C.: Improving Maintenance Activities by the Usage of BOL Data. In: *Procedia CIRP* 11 (2013), S.62–67.

Franke, M., Klein, K., Hribernik, K.A., Thoben, K.-D.: Semantic Data Integration Approach for the Vision of a Digital Factory. In: *Mertins, K., Jardim-Gonçalves, R., Popplewell, K., Mendonça, J.P.* (Eds.) *Enterprise Interoperability VII*. *Enterprise Interoperability in the Digitized and Networked Factory of the Future*, vol. 8. Springer International Publishing, Cham (2016), S. 77–86.

Franke, M., Krause, S., Thoben, K.-D., Himmler, A., Hribernik, K.A.: Adaptive Test Feedback Loop: A Modeling Approach for Checking Side Effects during Test Execution in Advised Explorative Testing, In: *SAE Technical Paper Series*. AeroTech. MAR. 17, 2020. SAE International400 Commonwealth Drive, Warrendale, PA, United States (2020b), S. 1182–1188.

Franke, M., Meyer, V., Rasche, R., Himmler. Andreas, Thoben, K.-D.: Interoperability of Test Procedures Between Enterprises: Intermediate Representation for

Test Procedure Exchange. In: *Enterprise Interoperability Viii*. *Smart services and business impact of*. Springer Nature, [S.l.] (2019), S. 177–188.

Franke, M., Thoben, K.-D.: Interoperable Test Cases to Mediate between Supply Chain's Test Processes. In: *Information* 13 (2022), S.498.

Franke, M., Thoben, K.-D., Ehrhardt, B.: The Faceted and Exploratory Search for Test Knowledge. In: *Information* 14 (2023), S.45.

Geraci, A., Katki, F., MCMonegal, L., Meyer, B., Lane, J.: *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries.* IEEE Press 1991.

Giese, H., Henkler, S., Hirsch, M., Priesterjahn, C.: Model-based testing of mechatronic systems. In: *Volume Editors* (2007), S.12.

Goh, C.H.: *Representing and reasoning about semantic conflicts in heterogeneous information systems*. Dissertation, Massachusetts 1997.

Guarino, N.: Formal Ontology in Information Systems. In: *Proceedings of the first international conference (FOIS'98)* 1998 (1998).

Guido Tebes, Luis Olsina, Denis Peppino, Pablo Becker: TestTDO: A Top-Domain Software Testing Ontology, In: . XXIII Conferencia Iberoamericana en Software Engineering (CIbSE'20) (2020).

Harel, D.: Statecharts: a visual formalism for complex systems. In: *Science of Computer Programming* 8 (1987), S.231–274.

HEICON Global Engineering GmbH: *Statische Analyse und Dynamischer Test: Wo liegen die Stärken und Schwächen? - Heicon Ulm*. WWW-Seite. https://heicon-ulm.de/statische-analyse-und-dynamischer-test-wo-liegen-die-staerken-und-schwaechen/. Besucht am: 8 Februar 2023.624Z.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G.d., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., others: Knowledge graphs. In: *ACM Computing Surveys (CSUR)* 54 (2021), S.1–37.

Hribernik, K.A., Kramer, C., Hans, C.: *Spezifikation eines Semantischen Mediators zur Unterstützung der Selbststeuerung logistischer Prozesse*. WWW-Seite. http://www.sfb637.uni-bremen.de/pubdb/repository/SFB637-C2-11-009-TR.pdf. Besucht am: 31 Januar 2023.

IDSA: *IDSA whitepaper 2018*. WWW-Seite. https://internationaldataspaces.org/publications/white-papers/. Besucht am: 30 Januar 2023.

International Data Spaces: *IDS Components - International Data Spaces*. WWW-Seite. https://www.bmwk.de/Redaktion/DE/Publikationen/Digitale-Welt/das-projekt-gaia-x.pdf?__blob=publicationFile&v=22. Besucht am: 30 Januar 2023.808Z.

ISO/IEC 15288:2008: *ISO/IEC 15288:2008(en), Systems and software engineering — System life cycle processes*. WWW-Seite. https://www.iso.org/obp/ui/#iso:std:iso-iec:15288:ed-2:v1:en. Besucht am: 10 August 2020.

ISO/IEC 9899:2018: *ISO/IEC 9899:2018*. WWW-Seite. https://www.iso.org/standard/74528.html. Besucht am: 2023-02-03T.

ISO/IEC/IEEE 12207:2017: *ISO/IEC/IEEE 12207:2017*. WWW-Seite. https://www.iso.org/standard/63712.html. Besucht am: 10 August 2020.

ISO/IEC/IEEE 29119-1:2013: *ISO/IEC/IEEE 29119-1:2013(en), Software and systems engineering — Software testing — Part 1: Concepts and definitions*. WWW-Seite. https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29119:-1:ed-1:v1:en:sec:4.39. Besucht am: 10 August 2022.

ISTQB Glossary: *ISTQB Glossary*. WWW-Seite. https://istqb-glossary.page/de/. Besucht am: 16 Juni 2021.

Jastram, M.: *Was ist eigentlich das V-Modell?* WWW-Seite. https://www.se-trends.de/was-ist-eigentlich-das-v-modell/. Besucht am: 17 Januar 2023.689Z.

Jorgensen, P.: *Software testing*: *A craftman's approach.* Auerbach, Boca Raton 2018, 1 online resource.

Kiritsis, D., Bufardi, A., Xirouchakis, P.: Research issues on product lifecycle management and information tracking using smart embedded systems. In: *Advanced Engineering Informatics* 17 (2003), S.189–202.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., others: Moses: Open source toolkit for statistical machine translation, In: *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions* (2007), S. 177–180.

Krueger, M., Lewis, J., Jacoby, C., Rantowich, N., Ice: *Systems Engineering Guidebook for Intelligent Transportation Systems*. WWW-Seite. https://www.fhwa.dot.gov/cadiv/segb/files/segbversion3.pdf. Besucht am: 14.02-2023.

Lachaux, M.-A., Roziere, B., Chanussot, L., Lample, G.: *Unsupervised Translation of Programming Languages* 2020.

Lenzerini, M.: Data integration, In: *Proceedings of the Tweny-first ACM SIGMOD SIGACT SIGART Symposium on Principles of Database Systems*. *Madison, Wisconsin, June 3-5, 2002*. the twenty-first ACM SIGMOD-SIGACT-SIGART symposium, Madison, Wisconsin. 6/3/2002 - 6/5/2002. ACM Press, New York, NY (2002), p. 233.

Lübbert, T.: *Integrierte Entwurfs-und Verifikationsmethode für verteilte eingebettete Steuerungssysteme am Beispiel eines Flugzeugkabinensystems*. Dissertation, Bochum 2010.

Mari, S.I., Lee, Y.H., Memon, M.S., Park, Y.S., Kim, M.: Adaptivity of Complex Network Topologies for Designing Resilient Supply Chain Networks. In: *International Journal of Industrial Engineering* 22 (2015).

Mascardi, V., Cord, V., Rosso, P.: A Comparison of Upper Ontologies, In: *Woa* (2007), S. 55–64.

Matyokurehwa, K., Mavetera, N., Jokonya, O.: Requirements engineering techniques: A systematic literature review. In: *International Journal of Soft Computing and Engineering* 7 (2017), S.14–20.

Munoz-Castaner, J., Asorey-Cacheda, R., Gil-Castineira, F.J., Gonzalez-Castano, F.J., Rodriguez-Hernandez, P.S.: A Review of Aeronautical Electronics and Its Parallelism With Automotive Electronics. In: *IEEE Trans. Ind. Electron.* 58 (2011), S.3090–3100.

Ören, T.I., Ghasem-Aghaee, N., Yilmaz, L.: An ontology-based dictionary of understanding as a basis for software agents with understanding abilities, In: *SpringSim (2)* (2007), S. 19–27.

Pan, S., Trentesaux, D., McFarlane, D., Montreuil, B., Ballot, E., Huang, G.Q.: Digital interoperability in logistics and supply chain management: state-of-the-art and research avenues towards Physical Internet. In: *Computers in Industry* 128 (2021), S.103435.

Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. In: *Journal of Management Information Systems* 24 (2007), S.45–77.

Peleska, J.: Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In: *Electron. Proc. Theor. Comput. Sci.* 111 (2013), S.3–28.

Pierce, B.C.: *Types and programming languages.* MIT Press, Cambridge, Mass, London 2002, 623 Seiten.

Pilorget, L. (Ed.): *Testen von Informationssystemen*: *Integriertes und prozessorientiertes Testen.* Vieweg+Teubner Verlag, Wiesbaden 2012.

projects.eclipse.org: *Eclipse GMF Tooling*. WWW-Seite. https://projects.eclipse.org/projects/modeling.gmf-tooling. Besucht am: 29 Januar 2023.565Z.

Python.org: *Welcome to Python.org*. WWW-Seite. https://www.python.org/. Besucht am: 3 Februar 2023.

Rasche, R., Himmler, A., Franke, M., Meyer, V., Klaus-Dieter, T. (Eds.): *Interfacing & Interchanging – Reusing Real-Time Tests for Safety-Critical Systems* 2018.

Reeve, A.: *Managing Data in Motion*: *Data Integration Best Practice Techniques and Technologies.* Elsevier Science, Burlington 2013, 203 Seiten.

Rowley, J.: The wisdom hierarchy: representations of the DIKW hierarchy. In: *Journal of Information Science* 33 (2007), S.163–180.

Schindler, M.: *Wissensmanagement in der Projektabwicklung*: *Grundlagen, Determinanten und Gestaltungskonzepte eines ganzheitlichen Projektwissensmanagements.* BoD – Books on Demand 2002.

Scholz, P.: *Softwareentwicklung eingebetteter systeme: grundlagen, modellierung, qualitätssicherung.* Springer-Verlag 2006.

Scholz-Reiter, B., Jakobza, J.: Supply Chain Management - Ueberblick und Konzeption. In: *HMD - Praxis der Wirtschaftsinformatik* 207 (1999), S.7–15.

Schulte, C.: *Logistik*: *Wege zur Optimierung der Supply Chain,* 6th ed. Vahlen, München 2013, 750 Seiten.

Shani, U., Franke, M., Hribernik, K.A., Thoben, K.-D.: Ontology mediation to rule them all: Managing the plurality in product service systems, In: *2017 Annual IEEE International Systems Conference (SysCon)* (2017), S. 1–7.

Sima, L.M.: Testing Mechatronic Systems and their Integration into the Digital Enterprise. In: *IOP Conf. Ser.: Mater. Sci. Eng.* 1268 (2022), S.12011.

Sima, L.M., Zapciu, M.: Testing the Interfaces for Mechatronic Systems. In: *Journal of Mechatronics and Robotics* 6 (2022), S.22–27.

Singhal, A.: Introducing the Knowledge Graph: things, not strings. In: *Google* (2012).

Smart, P.D., Jones, C.B., Twaroch, F.A.: Multi-source Toponym Data Integration and Mediation for a Meta-Gazetteer Service, In: . International Conference on Geographic Information Science. Springer, Berlin, Heidelberg (2010), S. 234–248.

Sudarsan, R., Fenves, S.J., Sriram, R.D., Wang, F.: A product information modeling framework for product lifecycle management. In: *Computer-aided design* 37 (2005), S.1399–1411.

Thaler, K.: *Supply chain management: Prozessoptimierung in der logistischen Kette.* Fortis-Verlag 1999.

Thoben, K.-D., Ait-Alla, A., Franke, M., Hribernik, K., Lütjen, M., Freitag, M.: Real-time Predictive Maintenance Based on Complex Event Processing. In: *Zelm, M., Jaekel, F.-W., Doumeingts, G., Wollschlaeger, M.* (Eds.) *Enterprise interoperability*. *Smart services and business impact of enterprise interoperability*, vol. 49. ISTE Ltd; Wiley, London, UK, Hoboken, NJ (2018), S. 291–296.

Thoben, K.-D., Hans, C., Gerke, D., Strahmann, J., Geiwiz, W., Neuhaus, M.: Effizientes testen komplexer mechatronischer Systeme. In: *Deutscher Luft- und Raumfahrtkongress, Tagungsband - Manuskripte, 2011* (2011), S.555–562.

Tubbs, D.J.: Lion: Listen online. Using GraphQL as a mediator for data integration and ingestion. In: (2018).

Uschold, M., Gruninger, M.: Ontologies: principles, methods and applications. In: *The Knowledge Engineering Review* 11 (1996), S.93–136.

Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. In: *Software testing, verification and reliability* 22 (2012), S.297–312.

van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. In: *ACM Sigplan Notices* 35 (2000), S.26–36.

W3C: *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. WWW-Seite. https://www.w3.org/TR/scxml/. Besucht am: 17 Oktober 2022.

Wache, H.: *Semantische Mediation für heterogene informationsquellen*. Dissertation 2003.

Wang, L., Deng, T., Shen, Z.-J.M., Hu, H., Qi, Y.: Digital twin-driven smart supply chain. In: *Front. Eng. Manag.* 9 (2022), S.56–70.

Wiederhold, G.: Mediators in the architecture of future information systems. In: *Computer* 25 (1992), S.38–49.

Witte, F.: *Strategie, Planung und Organisation von Testprozessen*: *Basis für erfolgreiche Projektabwicklung im Softwaretest,* 1st ed. Springer Fachmedien Wiesbaden; Imprint: Springer Vieweg, Wiesbaden 2020, 257).

Wittner, M.: *CCDL Whitepaper*. WWW-Seite. https://www.razorcat.com/files/de/produkte/ccdl/Razorcat_Technical_Report_CCDL_Whitepaper_02.pdf. Besucht am: 3 Februar 2023.

Zhang, Y., Zhu, H.: Ontology for Service Oriented Testing of Web Services. In: *2008 IEEE International Symposium on Service-Oriented System Engineering* (2008), S.129–134.

Zhu, C., Guo, X., Zou, S.: Impact of information and communications technology alignment on supply chain performance in the Industry 4.0 era: mediation effect of supply chain integration. In: *Journal of Industrial and Production Engineering* 39 (2022), S.505–520.