

Institut für Software Engineering

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

**Eine modellgetriebene
Automatisierung von
Code-Generation, -Integration und
-Deployment von autonomen
Fahrfunktionen**

Sebastian Baumfalk

Studiengang: Informatik

Prüfer/in: Prof. Dr.-Ing. Steffen Becker

Betreuer/in: Marcel Weller, M.Sc.

Beginn am: 2. Januar 2024

Beendet am: 24. September 2024

Danksagungen

Herzlichen Dank allen, die zum Gelingen meiner Masterarbeit beigetragen und mich auf so vielfältige Art und Weise unterstützt haben.

Danke an Herrn Prof. Dr. Becker für die Möglichkeit, meine Masterarbeit zum Thema „Eine modellgetriebene Automatisierung von Code-Generation, -Integration und -Deployment von autonomen Fahrfunktionen“ zu schreiben.

Ganz besonders bedanke ich mich bei Herrn M. Sc. Marcell Weller für seine Begleitung als Betreuer bei der Erstellung dieser Masterarbeit. Vielen Dank für alle wertvollen Hinweise, für alle investierte Zeit, für viele sehr gute Fachgespräche und für die gute Zusammenarbeit.

Ich möchte auch Herrn M. Sc. Sandro Speth für alle Hilfe und vielseitige Unterstützung während meines gesamten Studiums danken.

Danke für die Tipps und für alle schönen und wertvollen Gespräche.

Dankeschön an Frau Dr. Schneider und Herrn Dr. Zimmer für alle hilfreiche Beratung bezüglich Fragestellungen zu Organisation und Durchführung des Studiums.

Ich möchte mich auch sehr bei meinen Eltern für alle Unterstützung und Begleitung während meines gesamten Studiums bedanken.

Danke, dass ihr immer für mich da gewesen seid und dass ihr immer an mich geglaubt habt.

Abstract

Graphical models are used in different areas. In engineering they are used for e.g. the modeling of components and behaviours of cyberphysical systems (CPS). MechatronicUML (MUML) is a model driven software development method designed for such systems and its toolsuite can generate code from models.

An expansion for MUML is available, which enables the generation of arduino code, but it requires a complex post-processing process. The whole workflow for ready-to-use complete Arduino code based on MUML models involves many manual steps and therefore it previously couldn't be performed automatically.

Goal of this thesis is enabling an automatic build process for arduino code as well as the uploading of the produced code to the arduino micro controllers using an configurable pipeline, which supports the development practice „Continuous Integration/Continuous Deployment“ (CI/CD).

To support the version of the robot cars used at the date of this thesis, it was first investigated how the MUML tool suite and its expansions can be modified accordingly and one implementation has been selected. These adjustments at the MUML models were successfully carried out and the robot cars themselves were improved.

The integration of the work steps has been implemented as a versataile configurable pipeline. For that the exact sequences and calls of the manual work steps have been investigated.

The developed CI/CD pipeline can automatically carry out the workflow from the model transformations to the uploading of the code onto the robot cars' hardware. The evaluation has shown that this pipeline system works fast and reliable.

This way it assists development work to an high degree, because the users are relieved of the time-consuming build processes and the risks of error are avoided.

Kurzfassung

Grafische Modelle werden in verschiedenen Bereichen eingesetzt. In den Ingenieurwissenschaften werden sie beispielsweise für das Modellieren der Komponenten und Verhaltensweisen von Cyberphysischen Systemen (CPS) verwendet. MechatronicUML (MUML) ist eine für solche Systeme entworfene Methode für modellgetriebene Softwareentwicklung, deren Toolsuite aus Modellen Code generieren kann.

Es ist eine Erweiterung für MUML vorhanden, welche die Generation von Arduinocode ermöglicht, aber sie erfordert einen aufwändigen Post-Processing-Ablauf. Der gesamte Arbeitsprozess für nutzungsbereiten vollständigen Arduinocode ausgehend von den MUML-Modellen beinhaltet viele manuelle Schritte und war daher bisher nicht automatisch durchführbar.

Ziel dieser Ausarbeitung ist das Ermöglichen einer automatischen Ausführung des Build-Prozesses für Arduinocode sowie des Hochladens des hergestellten Codes auf die Arduino-Mikrocontroller mittels einer konfigurierbaren Pipeline, welche die Entwicklungspraxis „Continuous Integration/Continuous Deployment“ (CI/CD) unterstützt.

Für das Unterstützen der zum Zeitpunkt dieser Ausarbeitung verwendeten Version der Roboterautos wurde zunächst untersucht, wie die MUML-Toolsuite und ihre Erweiterungen entsprechend modifiziert werden können und eine Umsetzung ausgewählt. Diese Anpassungen an den MUML-Modellen wurden erfolgreich durchgeführt und es wurden an den Roboterautos selbst Verbesserungen vorgenommen.

Die Integration der Arbeitsschritte wurde als vielseitig konfigurierbare Pipeline umgesetzt. Hierfür wurden die genauen Abläufe und Aufrufe der manuellen Arbeitsschritte erforscht.

Die entwickelte CI/CD-Pipeline kann den Arbeitsablauf von den Modelltransformationen bis zum Hochladen des Codes auf die Hardware der Roboterautos automatisch durchführen. Die Evaluation hat gezeigt, dass dieses Pipelinesystem schnell und zuverlässig arbeitet.

Sie unterstützt dadurch Entwicklungsarbeiten in einem hohen Maß, weil der Nutzerschaft die zeitraubenden Build-Vorgänge abgenommen und Fehlerrisiken vermieden werden.

Inhaltsverzeichnis

1. Einführung	1
1.1. Problemstellung	1
1.2. Zielsetzung	2
1.3. Lösungsansatz	2
1.4. Ergebnis	3
2. Grundlagen	5
2.1. Die MechatronicUML und ihre Tool-Suite	5
2.2. Modell-Transformations-Ansätze	6
2.3. Continuous Integration/Continuous Deployment	7
2.4. Über die Arduino-CLI	10
2.5. Eclipse-Plug-Ins	10
3. Verwandte Arbeiten	13
3.1. Generation von Arduinocode mit MechatronicUML	13
3.2. Auswertung bestehender Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse	13
3.3. Taxonomie	17
3.4. Codequalität	17
3.5. CI/CD-Pipeline	20
4. Anwendungsszenario	21
4.1. Das „kooperatives Überholen“-Szenario	21
4.2. Die Roboterauto-Zielplattform	22
5. Analyse	25
6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite	29
6.1. Geplanter Ablauf für die Nutzung der CI/CD-Pipeline	30
6.2. Nicht oder unzureichend erfüllbare Anforderungen	36
6.3. Weitere Aspekte	37
6.4. Spezifische Pipelineschritte	38
6.5. Aufbau der Konfigurationsdatei	45
7. Umsetzungen	51
7.1. Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken	51
7.2. Integration der ArduinoCLI	56
7.3. Vollständige Automatisierung per Pipeline	67

8. Evaluation	75
8.1. Vorgehensweise	75
8.2. Ergebnisse	79
8.3. Diskussion	91
8.4. Gefahren für die Gültigkeit	93
9. Schlussfolgerung	95
9.1. Zusammenfassung	95
9.2. Vorteile	95
9.3. Limitierungen	96
9.4. Gelernte Lektionen	97
9.5. Zukünftige Arbeiten	97
Literaturverzeichnis	99
A. GitHub-Repository mit Sourcecode	103
B. Ergänzungsmaterial Übersicht und Details zu den Roboterautos	105
C. Ergänzungsmaterial Änderungen an MUML-Modellen	109
D. Ergänzungsmaterial Post-Processing-Ablauf	115
D.1. Änderungen am Post-Processing-Ablauf	115
D.2. Gegenüberstellung der Post-Processing-Schritte und der entsprechenden Pipelineschritte der generierbaren Beispielkonfiguration der Pipeline	123
E. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „ArduinoCLIUtilizer“	131
E.1. Verwendete Version	131
E.2. Die Konfigurationsdatei „arduinoCLIUtilizerConfig.yaml“	131
F. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „MUMLACGPPA“	133
F.1. Einschränkungen und Formate der Variablentypen	133
F.2. Beispiel der Reihenfolgeunterschiede	134
F.3. Typischer grober Ablauf der Pipelineschritte	136
G. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „PipelineExecution“	139
G.1. Überprüfung der Schritte nach einem Aspekt und gegebenenfalls Anzeigen einer entsprechenden Meldung	139
H. Ergänzungsmaterial Übersicht der Pipelineschritte	141
I. Verwendete Software und Projekte	147

Abbildungsverzeichnis

3.1. Übersicht des ISO-25010-Standards vom 23.2.2024 [Anw].	18
4.1. Überblick über das kooperative Überholmanöver mit zwei autonomen Fahrzeugen. Dieses Bild stammt aus Stürners Ausarbeitung [Stü22]. Das graue Auto stellt den Überholer dar und das weiße Auto den Partner.	22
4.2. Kamerabild der Unterseite eines für diese Ausarbeitung verwendeten Roboterautos. Die andere Konstruktion des Antriebes und der Lenkung sind erkennbar.	23
4.3. Kamerabilder eines für diese Ausarbeitung verwendeten Roboterautos. Die 9V-Blockbatterie und seine Halterung sind Teil der Energieversorgung des WiFi-Modules.	23
5.1. Der Prozess der Codegeneration wie abgebildet in der Anleitung für die Nutzerschaft [Coda], aber mit einer Änderung: Der Schritt T3.3 wurde blau markiert.	25
6.1. Übersicht des Nutzungsablaufes	31
6.2. Übersicht des Konfigurationsablauf, Teil 1	32
6.3. Übersicht des Konfigurationsablauf, Teil 2	33
6.4. Übersicht des Durchführungsablaufes	35
6.5. Container-Transformation, Teil 1	39
6.6. Container-Transformation, Teil 2	40
6.7. Container-Transformation, Teil 3	40
6.8. Container-Transformation, Teil 4	41
6.9. Übersicht über den Aufbau der Konfigurationsdatei der Pipeline „pipelinesettings.yaml“	46
6.10. Aufbau des Auftrages eines Pipelineschrittes	49
7.1. Konzeptgrafik des Aufbaus des Eclipse-Plug-Ins „ArduinoCLIUtilizer“. Die Nutzungsmöglichkeiten werden auch grob dargestellt.	62
7.2. Vereinfachte Übersicht über die *Action-Klassen aus dem Paket „de.ust.arduinocliutilizer.popup.actions“	65
B.1. Schaltplan der elektrischen und elektromechanischen Komponenten eines während dieser Ausarbeitung verwendeten Roboterautos.	106
B.2. Das modifizierte Roboterauto von links vorne	107
B.3. Das modifizierte Roboterauto von links hinten	107
B.4. Das modifizierte Roboterauto von rechts hinten	108
B.5. Das modifizierte Roboterauto von rechts vorne	108
F.1. Typischer grober Ablauf der Pipelineschritte. Pipelineschritte wie DialogMessage und OnlyContinueIfFulfilledElseAbort werden hierbei ignoriert.	137

Tabellenverzeichnis

7.1.	Namensänderungen	55
7.2.	Gegenüberstellung der manuellen Schritte und der Kommandozeilenbefehle, Teil 1	58
7.3.	Gegenüberstellung der manuellen Schritte und der Kommandozeilenbefehle, Teil 2	59
7.4.	Gegenüberstellung weiterer manueller Schritte und Kommandozeilenbefehle . . .	60
8.1.	Übersicht der Richtlinien und der Aspekte nach ISO-25010[Anw] sowie der jeweiligen Erfüllungsstufe in einer Tabelle	82
8.2.	Übersicht der Eigenschaften und Fähigkeiten nach Bigelow[Big] sowie der jeweiligen Erfüllungsstufe in einer Tabelle	89
D.1.	Vergleichstabelle der Änderungen beim Post-Processing-Ablauf	116
F.1.	Die Einschränkungen und Formate, die seitens der Variablenotypen gefordert werden.	134

Verzeichnis der Listings

7.1. Arduino-CLI-Befehle für die Installation der benötigten Arduino-Bibliotheken „Servo“, „WiFiEsp“ und „PubSubClient“.	61
G.1. Suchdurchlauf auf Schritte, die auf der Arduino-CLI basieren sowie das bedingte Anzeigen einer Fehlermeldung.	140

Verzeichnis der Algorithmen

Abkürzungen

- AMK** Arduino-Mikrocontroller. 3
- CD** Continuous Deployment. 7
- CI** Continuous Integration. 7
- CI/CD** Continuous Integration/Continuous Deployment. 1
- EASE** Eclipse Advanced Scripting Environment. 14
- ECU** Electric Control Unit. 22, 26
- HR** Hardwareressource. 6
- I2C** Inter-Integrated Circuit. 22, 80
- MDSE** Modellgetriebene Softwareentwicklung, auf Englisch „model driven software engineering“. 1
- MQTT** Message Queuing Telemetry Transport. 26, 91
- MUML** MechatronicUML. 1
- OTA** Over-the-Air. 3
- PDM** Plattformbeschreibungsmodell, auf Englisch „platform description model“. 6
- PIM** plattformunabhängiges Modell, auf Englisch „platform independent model“. 7
- PISM** Plattformunabhängiges Softwaremodell, auf Englisch „Platform Independent Software Model“. 6
- PSM** plattformspezifisches Modell, auf Englisch „platform specific model“. 6
- TEA** Task automation made easy. 15

1. Einführung

Die modellgetriebene Softwareentwicklung (MDSE) ist ein vielseitiger Ansatz mit verschiedenen Nutzungsmöglichkeiten und Vorteilen bei der Entwicklung von verschiedenen Systemen [OMG14]. So kann man die häufig zur Planung und Kommunikation genutzten formalen Modelle beispielsweise mit den entsprechenden Werkzeugen frühzeitig als komplette Systeme simulieren und verifizieren [OMG14]. Durch die Transformation zu Code wird Programmierzeit eingespart und das Resultat entspricht direkt der Planung [Mus]. Ein Anwendungsziel sind verteilte cyberphysische Systeme (auf Englisch *cyber physical systems*).

In diesen werden nicht nur informatische Systeme und Software sondern auch mechanische Komponenten miteinander verbunden. So werden Sensoren verwendet um Daten zu sammeln, Computersysteme werten diese aus und senden für eine Reaktion Befehle an die Aktoren, die wiederum diese ausführen und so auf die physische Welt Einfluss ausüben [Ben].

Im Fall von Autos sammeln verschiedene Sensoren Daten über die Umgebung und über das Auto selbst, die Boardelektronik wertet diese aus und je nach Resultat dieser Auswertungen werden z.B. Warnsignale an die Insassen gesendet.

Die Entwicklung von solchen Systemen kann durch verschiedene Entwicklungswerkzeuge erleichtert werden. Eines davon ist MechatronicUML (MUML). Es ist eine für cyberphysische Systeme entworfene MDSE-Methode und ihre Tool-Suite kann solche Systeme sowohl hinsichtlich ihrer Strukturen als auch hinsichtlich ihrer Verhaltensweisen modellieren, indem sie zustandsbasierte Teilsysteme und ihren Nachrichtenaustausch koordiniert [DPP+16]. Zusätzlich kann sie bei ausreichenden Modelldaten Code generieren [Stü22].

1.1. Problemstellung

Stürner hat im Rahmen seiner Arbeit „Generating Code for Distributed Deployments of Cyber-Physical Systems Using the MechatronicUML“ [Stü22] MUML analysiert und erweitert, um Code für arduino-basierte Roboterautos auf Basis von plattformunabhängigen Verhaltensmodellen generieren zu lassen. Tests haben jedoch gezeigt, dass der resultierende Arbeitsprozess auch außerhalb des Startens der Abläufe bzw. Plug-Ins und dem jeweiligen Auswählen der entsprechenden erforderlichen Einstellungen manuelle Handlungen außerhalb der grafischen Oberfläche erfordert. Dieser Prozess kann gegenwärtig nicht automatisch durchgeführt werden. Die Entwicklungspraxis Continuous Integration/Continuous Deployment (CI/CD) kann so auch nicht durchgeführt werden.

„CI/CD automatisiert die manuellen menschlichen Interventionen, die traditionell erforderlich sind, um neuen Code vom Commit in die Produktion zu bringen.“ [Hata] Alle Phasen der Entwicklung werden automatisiert und dadurch werden u.a. Probleme bei der Code-Integration vermieden [Hata]. Schließlich kostet jeder Ablauf der Arbeitsschritte nicht nur Zeit und bereitet Aufwand, sondern stellt auch Gelegenheiten für Fehler dar (siehe [Fow20]). Es können beispielsweise beim

1. Einführung

Starten der Code-Generation Missgeschicke beim Einstellen der Konfigurationen passieren, wie Tests von Stürners Arbeit gezeigt haben. Zusätzlich können bei dem Entwicklungspersonal die Entwicklungsumgebungen variieren, was Integrationsprobleme verschlimmert. [Hata]. Eine CI/CD-Pipeline ist eine Methode, eine automatisierte Durchführung dieser Schritte umzusetzen [Hatb]. Die neuere Generation der verwendeten Arduino-Roboterautos wird in Folge einer neuen Hardware und deren Bibliothek [Reid][Reia] auch nicht vollständig unterstützt.

1.2. Zielsetzung

Im Rahmen dieser Ausarbeitung wurde die Automatisierung des Code-Generations-Prozesses und die Ermöglichung von CI/CD erforscht und umgesetzt. Der Integrationsteil von CI/CD ist als vielseitige konfigurierbare Pipeline implementiert, die den Prozess anhand einer Konfigurationsdatei automatisch und flexibel durchführt. Hierbei erfolgt ein automatisiertes Aufrufen der MUML-Werkzeuge auf die MUML-Modelle und beispielsweise das Kompilieren und Hochladen erfolgen auf Basis der Arduino-CLI, einer Kommandozeilensoftware für das Arbeiten mit Arduinocode. Für den Deploymentteil bzw. dessen Testphase ist das Programmieren der Roboterautos per USB-Kabel möglich.

Für das Unterstützen der neueren Generation der verwendeten Arduino-Roboterautos wurden die notwendigen Änderungen erforscht und durchgeführt. Zusätzlich wurde das Post-Processing ebenfalls angepasst.

Im Rahmen dieser Ausarbeitung wurde die volle Unterstützung der momentan an der Uni Stuttgart genutzten Roboterautos hergestellt, indem die Modelle und Post-Processing-Abläufe aktualisiert wurden.

Als ein Resultat dieser erneut erweiterten MUML-Tool-Suite wird für die Code-Generation nur noch das Bereitstellen aller Modelle und Konfigurationsmöglichkeiten benötigt.

1.3. Lösungsansatz

Wie bereits angeschnitten wurde, verfolgt diese Ausarbeitung eine Reihe an Zielen, um die oben genannten Schwächen zu beheben. So wurde für das Unterstützen der neueren Version der verwendeten Roboterautos zunächst untersucht, wie MUML entsprechend erweitert werden kann, um anschließend die vielversprechendste Umsetzung zu wählen.

Für das Modellieren der zu implementierenden Pipelineschritte wurden die genauen Abläufe und Aufrufe in den manuellen Schritten erforscht. Für die Basis der Pipeline wurden bestehende Automatisierungsmöglichkeiten für Eclipse Neon getestet und miteinander verglichen. Die als am besten bewertete Möglichkeit, also das Implementieren eines eigenen Pipelinesystems als Eclipse-Plug-In, wurde als Basis für die Implementierung der Pipeline verwendet.

Für das Herstellen der Unterstützung der momentan an der Uni Stuttgart genutzten Roboterautos wurden Vergleiche durchgeführt, um die Unterschiede zwischen der Version von Roboterautos, die Stürner konzipiert hat [Stü22], und der im Rahmen dieser Ausarbeitung verwendeten Version zu finden und aufzulisten. Diese Informationen wiederum wurden für das Auswerten der notwendigen Anpassungen von Stürners Modellen für die Roboterautos [Stü22] verwendet. Für die Stellen, an denen mehrere Änderungsansätze gefunden wurden, werden die jeweiligen Aspekte miteinander verglichen und dann für die Implementierung der passendste Ansatz ausgewählt.

Um die Qualität der Umsetzungen zu prüfen, wurden für die Themen jeweils eine Taxonomie aufgestellt und angewandt. Deren Kriterien sind die allgemein anerkannten Eigenschaften oder/und Funktionen zu dem jeweiligen Analyseziel.

Das Anwendungsszenario, das in Stürners Arbeit verwendet wurde [Stü22], wird in dieser Ausarbeitung ebenfalls zu Demonstrations- und Vergleichs-Zwecken verwendet, wenn auch mit leichten Anpassungen für die geänderte Hardware und Software.

1.4. Ergebnis

Die Anpassungen an die neuere Version der Roboterautos wurden erfolgreich durchgeführt. Die Automatisierung per Pipeline wurde auch erfolgreich umgesetzt und erfüllt ihre Anforderungen, d.h. außgehend von den MUML-Modellen wird zuverlässig und innerhalb einer Minute direkt verwendbarer Arduinocode erstellt und auch zielgerichtet auf die verschiedenen angeschlossenen Arduino-Mikrocontroller (AMK) verteilt. Das Pipelinesystem ist auch robust gegen verschiedene Nutzungsfehler wie fehlerhaft geschriebenen Konfigurationen und bietet eine hohe Flexibilität. Das Konfigurieren ihrer Schritte und Einstellungen fällt sowohl beim Erlernen als auch beim Nutzen leicht. Durch das Befolgen von Qualitätsrichtlinien nach ISO-25010 [Anw] kann sie auch später leicht und zuverlässig modifiziert werden.

Von der CI/CD-Praxis konnten alle Aspekte bis auf das Testen und das Interagieren mit einer Versionsverwaltung umgesetzt werden.

Zusammengefasst ist das Ziel dieser Ausarbeitung größtenteils erfüllt worden: Die oben genannten Schwächen wurden behoben, die volle Unterstützung der momentan an der Uni Stuttgart genutzten Roboterautos hergestellt, der Code-Generations-Prozess automatisiert und die Entwicklungspraxis CI/CD wurde größtenteils ermöglicht. Die geschriebenen Eclipse-Plug-In-Projekte können in zukünftigen Projekten erweitert werden, z.B. für die Interaktion mit verschiedenen Versionsverwaltungssystemen oder komplexeren Pipelineabläufen. Die Erweiterung um die kabellose Programmierung, also das Updaten Over-the-Air (OTA), ist auch ein Forschungsfeld mit Praxisrelevanz, weil in der Realität viele moderne Autos kabellos Updates erhalten.

Als ein Resultat oder Beitrag dieser so erneut erweiterten MUML-Tool-Suite erfordern die Code-generation und das automatische Hochladen auf die verschiedenen AMKs per USB nur noch das Bereitstellen aller Modelle und Konfigurationen.

Struktur der Ausarbeitung

Diese Ausarbeitung ist wie folgt aufgebaut:

Kapitel 2 – Grundlagen: Als erstes werden die in dieser Ausarbeitung verwendeten theoretischen Grundlagen erklärt. Dies betrifft MUML und ihre Tool-Suite, die Ansätze der Modelltransformation, die DevOps-Methode „Continuous Integration/Continuous Deployment“, die Arduino-CLI und die Eclipse-Plug-Ins.

Kapitel 3 – Verwandte Arbeiten: Als zweites werden bereits bestehende Ausarbeitungen und Ansätze, die für diese Ausarbeitung relevant sind, erörtert. Dies betrifft Ausarbeitungen und Artikel zu den Themen der Taxonomie, MUML und Continuous Integration/Continuous

1. Einführung

Deployment. Ansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE und für die Automatisierung von Abläufen mit oder innerhalb der Eclipse-IDE werden auch abgehandelt werden.

Kapitel 4 – Anwendungsszenario: Danach wird das in dieser Ausarbeitung verwendete Anwendungsszenario beschrieben. Dieses beschreibt eine Beispielanwendung, die verwendeten Roboterautos und die Versuchsumgebung.

Kapitel 5 – Analyse: In diesem Kapitel wird der bisherige Arbeitsablauf analysiert.

Kapitel 6 – Entwurf der CI/CD-Pipeline für die MUML-Toolsuite: In diesem Kapitel werden die Aspekte und Entscheidungen bei dem Entwurf der CI/CD-Pipeline beschrieben.

Kapitel 7 – Umsetzungen: Danach werden die Details der verschiedenen Umsetzungen beschrieben. Hierbei wird also erklärt, wie die volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken sowie die Integration der Arduino-CLI erreicht wurden. Dasselbe gilt für die Automatisierung des Post-Processings der generierten Code-Teile und die vollständige Automatisierung per Pipeline.

Kapitel 8 – Evaluation: Die verschiedenen Inhalte und Ergebnisse dieser Ausarbeitung werden noch einmal in ihrer Gesamtheit betrachtet und diskutiert. Die Forschungsziele werden zusammen mit ihren Antworten wiederholt. Die im Rahmen dieser Ausarbeitung entwickelte Software wird hinsichtlich ihrer Qualität bewertet. Das Anwendungsszenario wird als Praxistest und Leistungsvergleich mit der entwickelten Software durchgeführt. Anschließend wird auf die Gefahren für die Gültigkeit eingegangen.

Kapitel 9 – Schlussfolgerung: Als Abschluss erfolgt eine kurze Zusammenfassung der wichtigsten Errungenschaften und Limitierungen. Es werden auch Punkte und Themen für zukünftige Forschungen angeschnitten.

2. Grundlagen

In diesem Kapitel werden die Grundlagen, auf denen diese Ausarbeitung beruht, beschrieben. In Abschnitt 2.1 „Die MechatronicUML und ihre Tool-Suite“ werden die MechatronicUML und ihre Tool-Suite beschrieben. In 2.2 „Modell-Transformations-Ansätze“ werden die Modell-Transformations-Ansätze kurz erläutert. In 2.3 „Continuous Integration/Continuous Deployment“ wird die Entwicklungspraxis CI/CD erklärt. In 2.4 „Über die Arduino-CLI“ wird die Arduino-CLI in die Ausarbeitung eingeführt. In 2.5 „Eclipse-Plug-Ins“ werden die Grundlagen von Eclipse-Plug-ins beschrieben.

2.1. Die MechatronicUML und ihre Tool-Suite

Der Spezifikation von Pohlmann et al. zu MUML [DPP+16] zufolge ist es eine modellgetriebene Methode für die Entwicklung von eingebetteter Software auf mechatronischen Systemen [DPP+16, Kapitel 1, Abschnitt „Introduction“, Seite 2, Absatz 4]. Somit passt es auch zu cyberphysischen Systemen. Deshalb wird im Folgenden MUMLs Modelliersprache beschrieben werden.

Durch diese Modelliersprache unterstützt MUML die Darstellung von sowohl strukturellen Aspekten als auch die Verhaltensdarstellung. Es werden auch Echtzeit-Zustandsdiagramme verwendet, die eine Kombination aus UML-Zustandsmaschinen und zeitabhängiger Automatentheorie darstellen. Die Software-Entwicklung basiert auf dem Modellieren oder Definieren von Komponenten [DPP+16, Kapitel 1, Abschnitt „Introduction“, Seite 2, Absatz 4]. Das Verhalten von Komponenten hängt von deren Typ ab [DPP+16, Kapitel 2, Seite 8 unten]: Die Komponenten, deren Funktionsweise direkt implementiert ist, werden als atomare Komponenten bezeichnet. Diejenigen, die stattdessen aus anderen Komponenten bestehen, werden als strukturierte Komponenten kategorisiert. Die Echtzeit-Koordination zwischen den Komponenten erfolgt in Form von Nachrichten zwischen den Port-Instanzen[DPP+16, Kapitel 2, Seite 8] und wird jeweils über Echtzeitkoordinationsprotokolle definiert: Es wird für jeden Port einer Kommunikation eine Rolle mit den zu erfüllenden Eigenschaften und einem Echtzeit-Zustandsdiagramm für den Nachrichtenaustausch festgelegt.

Bei atomaren Komponenten wird das interne Verhalten jeweils als Echtzeit-Zustandsdiagramm modelliert [DPP+16, Kapitel 2, Seite 8]. Das externe Verhalten wird nicht bei ihnen definiert, sondern es werden nur Ports als externe Interaktionspunkte definiert, die jeweils den Verhaltenstyp bei einer Verbindung festlegen. Das externe Verhalten hängt also davon ab, mit welchen Protokollen die Ports verbunden werden.

Es wird in diskrete, kontinuierliche und hybride Komponenten unterschieden [DPP+16, Kapitel 3.7.2], aber dies ist momentan für die vorgeschlagene Masterarbeit nicht relevant.

Bei strukturierten Komponenten wird das Verhalten von den darin enthaltenen Komponenten festgelegt [DPP+16, Kapitel 3.7.2]. Es werden aber nur diskrete strukturierte Komponenten und hybride strukturierte Komponenten unterstützt.

Pohlman et al. [DPP+16] [DP14] zufolge ist der Entwicklungsprozess mit MUML in folgende

2. Grundlagen

Schritte aufgeteilt:

1. Spezifizierte formale Anforderungen [DPP+16]:

Im ersten Schritt werden formale Anforderungen in Form von Use-Cases modelliert. Dies erfolgt durch eine Variante der UML-Sequenzdiagramme, genannt „Modal Sequence Diagram“. Diese spezifizieren die Interaktionen zwischen Systemelementen und die Bedingungen bei den Interaktionen. Des Weiteren werden sie für Analysen auf Inkonsistenzen und Widersprüchen verwendet.

2. Designe plattformunabhängiges Softwaremodell (auf Englisch platform independent software model, PISM) [DPP+16]:

Im zweiten Schritt wird aus den Use-Cases aus Schritt 1 ein PISM abgeleitet. Hierfür werden zunächst aus den Use-Cases die zuvor erwähnten Komponenten abgeleitet. Dann werden die Echtzeitkoordinationsprotokolle spezifiziert und mittels Komponenteninstanzkonfigurationen werden die Komponenten und Echtzeitkoordinationsprotokolle instanziert. Danach wird das Verhalten der Komponenten festgelegt. Im letzten Unterschritt werden die Kontrollkomponenten mit der Umgebung bzw. den Verhaltensmodellen integriert. Die Verifikation erfolgt über das Simulieren des Systemmodells und bei Erfolg bilden alle Modelle ein PISM.

3. Designe Software/Hardware-Plattform [DP14]:

Im dritten Schritt werden die Hardware und die Software der genutzten Plattform modelliert. Mit der Hardware Platform Description Language der MUML-Tool-Suite kann die Hardware als strukturierte Hardware-Plattform aus Hardware-Ressourcen-Instanzen modelliert werden. Hierbei stehen Hardware-Ressourcen (HRs) für die Hardware-Teile, die analog zu den zuvor erwähnten Komponenten strukturiert oder atomar sein können. Die strukturierten bestehen aus anderen HRs und möglicherweise atomaren Ressourcen. Die atomaren Ressourcen stehen für die verwendeten Speicher- und Verarbeitungs-Technologien.

4. Designe plattformspezifische Software:

Der Projektgruppe Cybertron zufolge [BCD+14] wird im vierten Schritt mittels mehrerer Unterschritte die plattformspezifische Software modelliert. Mit der Allocation-Specification-Language werden die Optimierungsziele und Bedingungen für das Erstellen der Allocation bzw. Zuweisung aus dem PISM und dem Plattformbeschreibungsmodell (auf Englisch platform description model, PDM) festgelegt. Diese Allocation wird mittels einer von den Entwicklern definierten Platform-Mapping bzw. Plattform-Zuordnung zu einem plattformspezifischen Modell (auf Englisch platform specific model, PSM)[BCD+14]. Dieses wird dann von einem Code-Generator zu einem Codegen model, d.h. einem Code-Generations-Modell, verwertet. Als letzter Unterschritt werden Analysen durchgeführt, ob die Plattform ausreichend Ressourcen für das Programm aufweist. Die Analyseergebnisse werden ausgegeben und bei Erfüllung der Kriterien wird der Sourcecode bzw. werden die Ergebnisartefakte erstellt.

2.2. Modell-Transformations-Ansätze

Die Transformationen für Erstellung von Code anhand eines Modells sind Teil von MDSE und in Stürners Arbeit [Stü22] wurde MUML mit Transformationen verwendet. Ein Beispiel ist bei der Container-Transformation eine Model-zu-Model-Transformation.

Czarnecki und Helsen haben eine Übersicht über Modell-Transformations-Ansätze erstellt [CH03] und ihnen zufolge existieren auf dem oberen Level zwei Kategorien:

Modell-zu-Modell-Transformationen: Diese transformieren ein Modell aus dem Quell-Format zu einem Zielformat, wobei diese zu unterschiedlichen Metamodellen gehören können. In ihrem Bericht haben die Autoren erwähnt, dass dieser Typ für das Überbrücken von Lücken zwischen plattformunabhängigen Modellen (auf Englisch platform independent model, PIM) und PSMs und wegen seiner Vielseitigkeit verwendet wird.

Modell-zu-Code-Transformationen: Czarnecki und Helsen zufolge wird dieser Typ häufig für die Generation von compilierbarem Code für eine spezifische Zielplattform verwendet [CH03]. Stürner hat in seiner Arbeit für die Generierung von Arduinocode Acceleo verwendet [Stü22].

2.3. Continuous Integration/Continuous Deployment

Zunächst wird auf den englischen Begriff „build“ aus den englischen Quellen eingegangen: Es steht sowohl für den Prozess oder für das Durchführen eines Prozesses, der aus Quell-Daten ein Programm oder dessen Sourcecode erstellt, als auch für das Resultat von diesem Prozess. Deshalb wird für dieses Dokument folgende Vereinheitlichung verwendet: Der Prozess wird als „Build-Prozess“ bezeichnet und das fertige Resultat als „Build“.

CI/CD ist eine bewährte Methode zur Programmierung und besteht aus zwei Teilen: Continuous Integration (CI) und Continuous Deployment (CD).

Der CI-Prozess läuft wie folgt ab (siehe Redhats Artikel [Hata] und [Kab]): Die Änderungen der Entwickler werden jeweils regelmäßig auf einer gemeinsamen Datenbank zusammengeführt, um große Probleme durch ein zeitgleiches Zusammenführen von vielen Änderungen zu vermeiden. Nach jeder Zusammenführung bzw. Integration werden automatisch Build-Prozesse gestartet und Tests durchgeführt. So können neu entstandene Konflikte schneller bemerkt und leichter aufgelöst werden, als bei einer Integrationsphase am Ende der Entwicklung. Nach RedHat[Hata] ist der CD-Prozess hingegen das automatische Durchführen von Tests, die Bereitstellung für die Operationsteams in einem Repository, von dem aus der Entwicklungsstand in eine Live-Produktumgebung gebracht werden kann, die Freigabe des Entwicklungsstandes an die Kundschaft und das Automatisieren der Bereitstellung.

Wenn statt „Deployment“ der Begriff „Delivery“ verwendet wird, endet der CD-Prozess vor der Bereitstellung für die Nutzerschaft[Hata].

Die Automatisierung aller Schritte von CI und CD wird als Pipeline bezeichnet [Kab]). So wird auch das Risiko von menschlichen Fehlern minimiert und der Prozess für das Release von Software bleibt konsistent [Kab]).

Fowler beschreibt CI detaillierter als eine Reihe von Kernmethoden[Fow20]. Im Folgenden wird sein Artikel zusammengefasst beschrieben werden:

„Führe eine einzige Datenbank“[Fow20]:

Alle Code-Dateien, die für das Erstellen eines Builds des Programms benötigt werden, sollten in einem Repository bzw. in einer Datenbank für den Quellcode zur Verfügung stehen. Es ist auch möglich, die Konfigurationen für die genutzten Programme, wie etwa die Einstellungen der Entwicklungsumgebung, hinzuzufügen. Im Idealfall sollte ein Computer, der nur über die Grundvoraussetzungen für die Softwareentwicklung verfügt, nach dem Herunterladen des Standes

2. Grundlagen

der Datenbank in der Lage sein, um den Build-Prozess für zu entwickelnde System durchführen zu können. Des Weiteren sollte das Datenbanksystem auch verschiedene Entwicklungszweige beherrschen können.

„Automatisiere [den Build-Prozess]“[Fow20]:

Es kann passieren, dass der Build-Prozess eines Software-Projekts ein komplizierter Prozess wird. Dies erfordert Zeit seitens der Entwickler und es kann bei jedem Schritt zu einem Fehler kommen. Die Automatisierung durch ein Build-System kann die Ausführung aller Schritte übernehmen. Es sollte aber flexibel genug sein, um für verschiedene Ansprüche unterschiedliche Ziele für den Build-Prozess umsetzen zu können. Als Beispiel sollten Komponenten, die unabhängig sein können, auch als alternative Ziele umsetzbar sein. Das Festlegen von Tests und deren Auswahl sollte auch integriert werden können.

In der grundlegenden Ausführung wird von dem Build-System immer alles durchgeführt, was bei großen Systemen deutliche Bearbeitungszeiten verursachen kann. In fortgeschritteneren Ausführungen werden die Änderungen und deren Abhängigkeiten ausgewertet und nur die erforderlichen Teile werden neu erstellt.

„Mache die Builds selbsttestend“[Fow20]:

Die im vorherigen Abschnitt erwähnten automatisierten Tests sind wichtig bei der Software-Entwicklung, weil die Ausführbarkeit eines Programms nicht die Korrektheit seines Ablaufes bedeutet. Ein Quellcode gilt als selbsttestend, wenn dieser über automatisierte Tests verfügt und diese mit einem Befehl gestartet werden können. Das Resultat des Testablaufes sollte angeben, ob Tests fehlschlugen. Wenn ja, so sollte der Build-Prozess abgebrochen werden.

Dieser Aspekt hilft auch bei Vorgehensweisen, in denen Tests eine wichtige Rolle einnehmen.

„Jeder committet täglich auf den Hauptstrang [des Sourcecodes in der Versionsverwaltung]“[Fow20]: Dieser Aspekt beschreibt eine regelmäßige Kommunikation über die jeweiligen Änderungen von den Entwicklern. Jeder von diesen arbeitet genau genommen mit einer Arbeitskopie und die Arbeitsfortschritte auf diesen müssen in den Hauptstrang integriert werden, sofern die erforderlichen automatischen Tests erfüllt werden. Das Durchführen der Integrationen in kurzen Intervallen, z.B. stündlich, sorgt dafür, dass mögliche Konflikte schneller bemerkt und gelöst werden können, weil so die Anzahl der an diesen involvierten Entwicklern niedrig bleibt und währenddessen nur wenig passiert. Bei der Suche nach Fehlergründen sind kleinere Änderungsschritte bei den Commits leichter zu durchsuchen.

„Jeder Commit sollte den Hauptstrang auf der Integrationsmaschine bauen“[Fow20]:

Es kann trotz der bisher genannten Maßnahmen zu Fehlern kommen, deren Auswirkungen auf die lokalen Build-Prozesse und Testvorgänge unbemerkt bleiben, aber Schäden am Hauptstrang verursachen. Neben nicht durchgeföhrten lokalen Updates und Build-Prozessen können auch Unterschiede bei der Programmierumgebung bei den Entwicklern Probleme verursachen. Deshalb sollte eine separate Integrationsmaschine eingerichtet werden, die den aktuellen Stand vom Hauptstrang nur herunterlädt und den Build-Prozess ausführt. So kann der Hauptstrang überwacht werden und auftretende Fehler müssten von dem Entwickler, der seinen Stand hochgeladen hat, behoben werden.

2.3. Continuous Integration/Continuous Deployment

Die Arbeit auf der Integrationsmaschine kann nach einem Commit manuell oder, falls ein Server verwendet wird, automatisch gestartet werden.

„Behebe kaputte Builds sofort“[Fow20]:

Dies ist ein Schlüsselaspekt: Sobald der Build-Prozess des Hauptstranges fehlschlägt, sollte die Fehlerbehebung priorisiert und sofort durchgeführt werden. CI beruht auf einem stabilen Stand als Basis, mit der alle arbeiten. Wenn der Grund für den Fehlschlag nicht sofort auffällt, sollte der Hauptstrang auf den neuesten bekannten funktionierenden Zustand zurückgesetzt werden.

„Halte den Build-Prozess schnell“[Fow20]:

Die schnelle Rückmeldung über den Code ist integral für CI. Je nach Größe des Projektes kann der Build-Prozess viel Zeit beanspruchen und so Wartezeiten bei den Entwicklern verursachen. Das Aufteilen des Build-Prozesses und seine Tests in eine Deployment-Pipeline kann jedoch als Kompromiss zwischen Geschwindigkeit und dem Finden von Bugs helfen. Eine Deployment-Pipeline ist eine Reihe von Build-Prozessen, bei der die ersten Builds die zuerst benötigten Tests durchführen und die späteren Builds z.B. die langsameren Tests. Fowler hat in [Fow20] den ersten Build als den Commit-Build bezeichnet, weil dieser benötigt wird, wenn jemand die Freigabe für den Commit auf den Hauptstrang benötigt. So kann dann mit dem neuen Stand weiter gearbeitet werden, während die späteren Stufen im Laufe ihrer Ausführung weitere Fehler aufzeigen können.

„Teste in einem Klon der Produktionsumgebung“[Fow20]:

Hierbei soll eine Umgebung die Anwendungsumgebungen, soweit es praktikabel ist, nachstellen. Je genauer die Nachstellung und/oder die Vielfalt der Umgebungen, desto geringer sind die Risiken für die Kompatibilität. So sollten also nach Möglichkeit das Betriebssystem, die Hardware, die Netzwerkverbindungen, die genutzte Datenbank und nicht nur die erforderlichen Bibliotheken benutzt werden. Jedoch werden häufig Test-Doppelgänger verwendet, wenn etwas z.B. für die Tests zu langsam wäre.

Es ist auch möglich, nach dem selben Prinzip wie in „Halte den Build-Prozess schnell“ verschiedene Testumgebungen zu verwenden. Fowler zufolge können virtuelle Maschinen sehr helfen, weil diese mehrere Testumgebungen auf einem Gerät ermöglichen.

„Mache es für jeden einfach, an die neueste ausführbare Datei zu kommen“[Fow20]:

Für ein frühes Feedback z.B. seitens der Auftraggeber sollte eine Stelle bekannt sein, wo sie die neuesten ausführbaren Dateien finden können. Dies ist sehr hilfreich für z.B. Demonstrationen oder agile Entwicklung.

„Jeder kann sehen, was passiert“[Fow20]:

Der Zustand des Systems und alle Änderungen sollten leicht sichtbar sein. Neben Informationen hinsichtlich des Erfolgs oder Fehlschlags eines Builds sollten weitere Informationen leicht sichtbar sein, beispielsweise wer den Build-Prozess gestartet hat und ob er diesen noch beaufsichtigt oder wie das Ergebnis des neusten Builds ist.

„Automatisiere das Deployment“[Fow20]:

Das Deployment in die Produktionsumgebung sollte auch durch ein Skript automatisiert werden, weil dies an einem Tag mehrmals erforderlich werden kann. Neben der Zeitsparnis wird dadurch

2. Grundlagen

auch das Fehlerrisiko reduziert. Für mögliche unerwartete Probleme sollte hierbei auch das Zurücksetzen auf einen früheren Stand möglich sein.

Mit dem letzten Punkt hat Fowler auch das CD abgehandelt.

2.4. Über die Arduino-CLI

Die Arduino-CLI ist eine offizielle kommandozeilenbasierte Software für das Arbeiten mit Arduinocode und Boards, auf die mit der Software von Arduino zugegriffen werden kann. Sie wurde nach den Installationsproblemen von „Eclipse C++ Tools for Arduino 2.0“ (siehe Abschnitt 3.2.1 „Ansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE“) bei einer Suche nach der Verwendung von Arduinosoftware per Kommandozeile gefunden (Eingabe: „arduino command line“). Sie wurde statt des Erforschens der von der Arduino-IDE verwendeten Befehle gewählt, weil sich ihre Nutzung beim Lesen ihrer offiziellen Dokumentation als leichter und zuverlässiger nutzbar herausgestellt hat.

In vorläufigen Nutzbarkeitstests konnten über die von ihr bereitgestellten Kommandozeilenbefehle erfolgreich alle Schritte, die über die Arduino-IDE erfolgten, durchgeführt werden.

2.5. Eclipse-Plug-Ins

Eclipse wurde als modulares Softwaresystem konzipiert [Foug]. In diesem Abschnitt werden die für diese Ausarbeitung relevanten Aspekte der Entwicklung von Eclipse-Plug-Ins zusammengefasst. Hier stellen Plug-Ins einzelne modulare Einheiten dar, die jeweils etwas Code gruppieren und ihre Anforderungen wie Java-Packages und andere Plug-Ins sowie die bereitgestellten eigenen Java-Packages und Funktionen für Eclipse spezifizieren [Fouf]. Solche Informationen werden in den Manifest-Dateien wie plugin.xml und MANIFEST.MF festgehalten [Foua].

Im Sinne der Modularität sollen sie nur eine lose Kupplung aufweisen und dies wird mitunter durch Extension und Extension-Points erreicht [Foug].

Extensions sind der Ansatz für das Hinzufügen von Verhaltensmöglichkeiten und GUI-Elementen [Foug] zu Eclipse [Foue]. Ihre Definitionen geben Informationen wie die jeweils zugrunde liegende Klasse, aber auch Bedingungen wie Namensfilter an [Foue]. Export-Wizards werden ebenfalls über Extensions zu Eclipse hinzugefügt [Foui]. Extension-Points werden bei einem Bezug auf Code wie eine angegebene Klasse oder ein Paket verwendet und ihre Möglichkeiten reichen von nutzenden Zugriffen wie Funktionsaufrufen zum Hinzufügen oder Ersetzen von Code [Foue].

Die Ausführung einer Eclipse-Installation mit eigenen geschriebenen Plug-Ins kann auf zwei Wegen erfolgen [Fouh]:

1. Der Build-Vorgang des Plug-In-Projekts wird durchgeführt und danach werden die erstellte .jar-Datei und die zuvor erwähnten Manifest-Dateien in dem Ordner „plugins“ in den Eclipse-Installationsordner kopiert. Beim nächsten Start der Workbench bzw. Arbeitsumgebung wird es erkannt und geladen.

2.5. Eclipse-Plug-Ins

2. Durch das Starten des Plug-In-Projekts in der Runtime-Workbench. Dies erfolgt per [Rechtsklicken auf den Plug-In-Projekt-Ordner]/„Run As Eclipse Application“. Im Vorfeld erfolgte Tests haben gezeigt, dass zumindest unter der verwendeten Eclipse-Installation aus [Foub] hierbei automatisch alle offenen Plug-In-Projekte gebaut und geladen werden.

Für das leichtere Verteilen von Plug-Ins ist das Exportieren zu Ordnern und .jar-Dateien möglich [Fouf].

3. Verwandte Arbeiten

Dieses Kapitel umfasst die offiziellen Arbeiten und Dokumente, die in diese Ausarbeitung einfließen. In 3.1 „Generation von Arduinocode mit MechatronicUML“ wird auf Stürners Erweiterung zu MUML für die Generation von Arduinocode eingegangen. In 3.2 „Auswertung bestehender Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse“ werden bestehende Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse beschrieben und ausgewertet. In 3.3 „Taxonomie“ wird das Taxonomiekonzept kurz erklärt. In 3.4 „Codequalität“ wird der ISO-25010-Standard für die Qualität von Code aufgelistet. In 3.5 „CI/CD-Pipeline“ wird abschließend auf die Eigenschaften und Fähigkeiten von CI/CD-Pipelines eingegangen.

3.1. Generation von Arduinocode mit MechatronicUML

Die vorgeschlagene Arbeit baut auf Stürners Arbeit auf [Stü22]. Seine Erweiterung für MUML verwendet teilweise [Stü22] einen Ansatz für die Generierung von Sourcecode ausgehend von einem MUML-Modell wie ausgearbeitet von der Projektgruppe Cybertron [BCD+14] wieder. Dieser Ansatz ist im Groben ähnlich zu den generischen Abläufen für Generierung von Code ausgehend von einem Verhaltensmodell: Ein als PIM modelliertes MUML-Modell ist Quell-Modell und wird zusammen mit einem PDM durch eine Modell-zu-Modell-Transformation zu einem PSM transformiert. Dieses PSM wird dann durch eine Modell-zu-Code-Transformation zu Sourcecode für die entsprechende Plattform transformiert.

Stürner hat von Cybertron [BCD+14] die Plattform-Modellierung und das Allocation Engineering übernommen, aber bei der Software-Konstruktion mussten Teile für die Anwendung auf die genutzten arduinobasierten Roboterautos angepasst werden [Stü22]. In seinem Ergebnis müssen jedoch manche Schritte außerhalb der grafischen Oberfläche manuell durchgeführt werden.

3.2. Auswertung bestehender Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse

Um einen Überblick über mögliche bereits vorhandene Lösungsansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE und die Automatisierung von Abläufen mit Eclipse in dieser Ausarbeitung zu erhalten, wurden jeweils Recherchen durchgeführt. In diesem Kapitel wird beschrieben, welche Ansätze gefunden wurden und aus welchen Gründen stattdessen die in dem folgenden Kapitel 7.2 „Integration der ArduinoCLI“ durchgeführte Umsetzung ausgewählt wurde.

3. Verwandte Arbeiten

3.2.1. Ansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE

Um das Verlassen der Eclipse-IDE und das Wechseln in die Arduino-IDE zu vermeiden, wurde im Vorfeld nach einer Möglichkeit für das Arbeiten mit Arduinocode und den passenden Mikrokontrollern gesucht. Wegen der in der Konsole der Arduino-IDE aufgelisteten Meldungen und Befehle wurde zunächst die Sucheingabe „arduino command line“ verwendet und so die Arduino-CLI gefunden. Diese ist eine offizielle Software von Arduino. Sie wurde wegen ihrer Automatisierbarkeit per programmatischer Nachstellung des Terminals bzw. Kommandozeile als leicht automatisierbar erkannt. In Kapitel 7.2 „Integration der ArduinoCLI“ wird sie näher beschrieben werden. Es wurde auch eine Suche nach bereits vorhandenen Möglichkeiten für das Arbeiten mit Arduinocode in Eclipse zum Gegenvergleich mittels der Eingaben „eclipse plug in arduino“ und „eclipse arduino“ durchgeführt. Hierbei wurde das Plug-In „Eclipse C++ Tools for Arduino 2.0“ gefunden. Aufgrund der leichten Testbarkeit und Interesse wurde dieses bereits während des Proposals zu dieser Ausarbeitung getestet. Bei diesen Vorabtests ist jedoch ihre Installation gescheitert, weil nicht mehr alle der erforderlichen Bibliotheken von dem Plug-in-Installer gefunden werden konnten.

3.2.2. Ansätze für die Automatisierung von Abläufen mit oder innerhalb der Eclipse-IDE

Für dieses Ziel wurde zunächst nach bereits vorhandenen Möglichkeiten für das Automatisieren von Abläufen mit Eclipse gesucht und dann auf ihre Durchführbarkeit getestet. Hierbei wurden offizielle Ansätze priorisiert, weil sie offiziell unterstützt werden und nicht von Drittanbietern stammen. Die Automatisierungsmöglichkeiten von Drittanbietern wurden ursprünglich nur als Reserve für den Fall, dass keine offizielle Lösung funktioniert, vorgesehen, aber wegen der schlechten Resultate zu EASE (siehe Abschnitt 3.2.2 „EASE“) getestet. An dieser Stelle sei auch erwähnt, dass eingestellte Software nicht als Ansatz zugelassen wurde. So soll weder eine weitere Hürde gegen einen möglichen Umstieg auf aktuelle Eclipse-Versionen geschaffen werden noch das Risiko von möglicherweise gelöschter oder aus sonstigen Gründen nicht mehr installierbaren Automatisierungswerkzeugen in das Ergebnis dieser Ausarbeitung eingebracht werden. Durch die in diesem Abschnitt bzw. im Folgenden beschriebenen Analysen und Auswertungen werden folgende Forschungsziele beantwortet:

FZ6.2 Welche Kriterien sind für die Auswertung der Build-Automatisierungsmöglichkeiten für Eclipse relevant?

FZ6.3 Welche der Build-Automatisierungsmöglichkeiten für Eclipse erfüllt die Kriterien am besten?

Offizielle Automatisierungsansätze

EASE

Die Eclipse Advanced Scripting Environment, kurz EASE, beruht auf der Interpretation von Skripten durch eine Java-Runtime innerhalb einer laufenden Eclipse-Instanz [Fouc]. Es stellt also praktisch gesehen nur eine Entwicklungsumgebung für ein leichteres Heraussuchen von zugreifbaren Teilen der Eclipse-IDE und das Aufrufen von diesen dar [Fouc]. Die Automatisierung von Vorgängen in Eclipse erfolgt genauer gesagt über die Skript-Module, die die Skripte enthalten [Fouj]. Verschiedene Sprachen wie Java werden mittels verschiedener Script-Engines unterstützt, wobei Entwickler

3.2. Auswertung bestehender Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse

eigene Engines schreiben können [Fouj]. EASE verfügt nicht über die Fähigkeiten einer per Konfigurationsdatei einstellbaren Pipeline. Es wird auf der Webseite von TEA [Foud] erwähnt und wurde deshalb zwecks Vollständigkeit auch überprüft. Aufgrund der verwendeten Neon-Edition der Eclipse-IDE wurde hierfür die Version 0.4.0 verwendet [Fouc].

Bei der Ausführung bzw. Interpretation von diesen kann es auch auf Java-Klassen und Teile der Eclipse-IDE zugreifen [Fouc]. Recherchen der Dokumentation und Tests haben gezeigt, dass es von sich aus keinen direkten Zugriff auf die Inhalte von Projekten und Plug-Ins erhalten kann, weil es bei diesen entweder deren Integration als Modul oder das Erstellen von Wrappern auf Instanzen einer Javaklasse mittels `wrap([Instanz der Java-Klasse])` erfordert [Fouj]. Die Nutzung ist jedoch in beiden Fällen auf Schnittstellen mit der Sichtbarkeit public beschränkt [Fouj]. Folglich müssen bei existierendem Java-Code entweder Module deklariert und Wrapper eingefügt werden oder bei jeder Instanz einer Klasse aus diesen `wrap([Instanz der Klasse])` verwendet werden [Fouj]. Dies wiederum bedeutet, dass bestehender Code hinsichtlich der Sichtbarkeiten und der von außen aufrufbaren Funktionen angepasst werden müsste.

Bei Nutzung wären jedoch in beiden Fällen Aufrufe und Programmierungen analog zu den Interaktionen der Plug-Ins untereinander erforderlich. Möglichkeiten für automatisierte Zugriffe auf GUI-Elemente wie bei den Exporten, die bei der Automatisierung von diesen eine große Zeitersparnis darstellen würden, konnten nicht gefunden werden. Die hierfür verwendeten Suchanfragen lauteten „`eclipse ease automate export`“, „`eclipse ease export`“ und „`eclipse ease access ui`“. Beim Arbeiten mit Eclipse-Plug-In-Projekten stellt es folglich praktisch gesehen nur einen Umweg für programmatische Aufrufe bereit.

Aufgrund dieser Problematik und der oben beschriebenen Schwächen stellt dieser Ansatz eine prinzipiell funktionierende, aber umständliche Basis für eine Automatisierung dar.

TEA

TEA, kurz für „Task automation made easy!“ [Foud], ist ein offizielles Set aus Erweiterungen für die Eclipse-IDE, deren Hauptaufgabe die Automatisierung von Aufgaben oder gar Aufgabenketten mit und ohne Nutzung der grafischen Oberfläche ist [Foud]. Im Gegensatz zu EASE soll es komplexere Anwendungen besser umsetzen können [Foud]. Hierfür werden die Aufgaben gewöhnlich als Eclipse-Plug-Ins geschrieben [Foud]. Es kann auf Teile der IDE zugreifen und auch ohne einem Workspace arbeiten [Foud]. Es wird auch eine Bibliothek bereitgestellt, die beispielsweise zu Jenkins kompatibel ist [Foud]. Bei diesem Ansatz hat das Alter von Eclipse Neon deutliche Auswirkungen auf die Installierbarkeit: Offizielle automatische Installationsmöglichkeiten, die eine Eclipse-Installation mit TEA vorbereiten konnten, funktionieren nicht mehr, weil nicht mehr alle von deren verwendeten Adressen die alten Dateien bereitstellen können. Beispielsweise muss man für das optionale Lcdsl nicht den Link „<https://mduft.github.io/lcdsl-latest/>“ aufrufen, sondern „<https://mduft.github.io/lcdsl-legacy/>“.

Wichtige Tauglichkeitstests von TEA schlugen fehl, z.B. weil die Erkennung von Bibliotheken nicht funktionierte. Deshalb wurde auch dieser Ansatz für diese Ausarbeitung verworfen, aber er könnte bei neueren Versionen bzw. Editionen von Eclipse korrekt funktionieren.

3. Verwandte Arbeiten

Ansätze von Drittanbietern

Apache Ant

Apache Ant, gewöhnlich kurz Ant genannt, ist gleichermaßen eine Bibliothek und Kommandozeilenwerkzeug, das anhand von Konfigurationsdateien Prozesse steuert[Foum]. Es wird hauptsächlich für Java-Anwendungen verwendet, kann aber auch außerhalb von Java jeglichen Typ von Prozess steuern, solange dieser in Form von Zielen und Tätigkeiten beschrieben werden kann[Foum]. Es ist auch möglich, Erweiterungen in Form von „antlibs“ zu programmieren[Foum]. Bei der Nutzung von Ant können .jar-Dateien und deren Methoden aufgerufen werden [Fouo, „Tutorial: Hello World with Apache Ant“].

In der Dokumentation[Fouo] konnte jedoch keine Möglichkeit für das Aufrufen von Inhalten von unkompliierten java-Dateien gefunden werden. Dies ist für die Ziele dieser Arbeit wichtig, weil die Plug-In-Projekte der MUML-Erweiterungen eine lange Liste an Errors und Duplikaten (durch das „alles importieren“-Vorgehen bei den MUML-Plug-Ins gibt es nicht nur die org.*-Projekte, sondern auch die Projekt-Ordner, die diese ebenfalls enthalten) haben. Alle Versuche, sie zu .jar-Dateien zu kompilieren, schlugen fehl.

Apache Maven

Apache Maven, gewöhnlich kurz Maven genannt, wird als „Softwareprojektmanagement- und Verständnis-Werkzeug“[Foun] beschrieben. Es kann Build-Prozesse, Reports und Dokumentationen managen[Foun]. Zusätzlich kann es Tätigkeiten von Ant verwenden, mit mehreren Projekten gleichzeitig arbeiten und verschiedene Versionsverwaltungssysteme für den Quellcode integrieren[Foul]. Benötigte Bibliotheken können auch automatisch heruntergeladen werden[Foul]. Eigene Erweiterungen werden als Maven-Plug-Ins geschrieben[Foun]. Zur Konfiguration werden mehrere Dateien verwendet [Fouk]. Jedoch wird auch hier kompilierter Javacode, also .jar-Dateien, benötigt, woran auch der vorherige Ansatz gescheitert ist[Foul].

Bei Versuchen mit m2e-extras konnten auf „<http://ifedorenko.github.com/m2e-extras/>“ keine Daten gefunden werden. Bei der Installation über „<https://download.eclipse.org/technology/m2e/releases/latest/>“ oder einer der dafür automatisch erfolgten Installation von benötigten Daten wurde etwas aus der gegebenen Eclipse-Installation für MUML entfernt oder ersetzt, weil Teile davon ihre Ausführung mit Nullpointer-Fehlern („Nullpointer exception“) abbrachen. Die Kommunikation mit den Downloadseiten und die Installationshistorie funktionierten nicht mehr und evtl. wurde auch mehr gestört. Die Fehlfunktionen konnten nicht behoben werden, also wurde die gestörte Eclipse-Installation durch ein Backup ersetzt.

Somit ist auch Maven keine mögliche Basis für die Automatisierung per Pipeline.

Gradle Build Tool

Gradle Build Tool, gewöhnlich kurz Gradle genannt, ist ein Abhängigkeitenverwaltungs- und Build-Prozess-Automatisierungs-Werkzeug, das für das Festlegen der Build-Prozesse in den Konfigurationsdateien entweder Groovy oder Kotlin verwendet [Bae]. So können manche Domänenprobleme gelöst werden [Bae]. Der Großteil der Funktionalität, wie z.B. die Verwendbarkeit für Java, erfolgt

durch Plug-Ins [Bae] und es ist auch möglich, eigene Plug-Ins zu entwickeln [Inca]. Gradle kann auch mit mehreren Projekten gleichzeitig arbeiten [Inca]. Laut Angaben in dem Benutzerhandbuch [Incb] soll es durch seine Schnelligkeit und Skalierbarkeit große und komplexe Projekte verarbeiten können.

In der Dokumentation [Incb] konnte keine Möglichkeit für das Nutzen von unkompliierten Eclipse-Plug-Ins bzw. unkompliiertem Java-Code gefunden werden. Es wurde nur auf kompliierte Dateien bzw. .jar-Dateien und deren Ausführung hingewiesen.

Somit ist auch dieser Ansatz verworfen worden.

Auswertung

Von den offiziellen und inoffiziellen Automatisierungsmöglichkeiten wurden alle bis auf EASE verworfen. EASE hat funktioniert, aber die Tests haben gezeigt, dass das Konzept der Eclipse-Plug-Ins eine Konkurrenz darstellt (siehe Abschnitt 2.5 „Eclipse-Plug-Ins“). Die reine Eclipse-Plug-In-Nutzung erfordert keinen Installationsaufwand an zusätzlicher Software und erreicht ohne Integrationsaufwand an anderen Eclipse-Plug-In-Projekten dasselbe Potential. Das Hinzufügen von GUI-Elementen und das Durchführen von Aktionen bei Verwendung von diesen ist in beiden Fällen möglich. Dies gilt ebenfalls im Hinblick auf das Ermöglichen von CI/CD per pipelinebasierter Automatisierung, weil die Entwicklung eines Systems für die Pipeline selbst bei beiden Ansätzen erforderlich wäre.

Die Antwort auf FZ6.2 „Welche Kriterien sind für die Auswertung der Build-Automatisierungsmöglichkeiten für Eclipse relevant?“ sind folglich die Unterschiede, also der Installations- und der Integrations-Aufwand. In beiden Punkten ist das Entwickeln von Eclipse-Plug-Ins überlegen, weil die Installation von zusätzlicher Software bei der vorbereiteten Eclipse-Installation für MUML über das Lubuntu-Image [Foub] nicht mehr durchgeführt werden muss und die Integration erfordert beispielsweise keine Wrapper. Dies beantwortet FZ6.3 „Welche der Build-Automatisierungsmöglichkeiten für Eclipse erfüllt die Kriterien am besten?“. So wurde auch EASE abgelehnt und die Entwicklung eines eigenen Plug-Ins, das auf andere Plug-Ins zugreift und so deren Nutzung automatisiert, als Basis für die Automatisierung per Pipeline gewählt.

3.3. Taxonomie

Laut Usman et al. [UBBM17] ist Taxonomie hauptsächlich als ein Klassifizierungssystem definiert. In diesem sollen beim Aufstellen einer Taxonomie zu einer Umsetzung jeweils die in dem Forschungsfeld oder in der Branche allgemein akzeptierten Aspekte, Attribute, Faktoren oder Funktionen unter Verwendung der jeweiligen üblichen Terminologie berücksichtigt werden [UBBM17].

3.4. Codequalität

Für die Code-Qualität wurde der ISO-25010-Standard [Anw] gewählt. Dieser beschreibt verschiedene Kriterien, von denen diejenigen, die für diese Ausarbeitung bzw. für den im Rahmen des von ihr erstellten Quellcodes relevant sind, im Folgenden zusammengefasst beschrieben werden.

3. Verwandte Arbeiten

Hinweis: Inzwischen wurde bemerkt, dass die Webseite und deren Punkte geändert wurden. Über „<https://web.archive.org/>“ können alte Versionen eingesehen werden, soweit diese und deren Unter-Seiten archiviert wurden. Für die Ausarbeitung wurde mit einer lokalen Kopie gearbeitet, d.h. es wurde alles am 23.2.2024 als Notiz in ein Textdokument kopiert. Diese alte Version wurde beibehalten, weil sie bei einer Besprechung über die Kriterien vorgelegt und für gut befunden wurde.



Abbildung 3.1.: Übersicht des ISO-25010-Standards vom 23.2.2024 [Anw].

3.4.1. Funktionale Eignung

Wie weit Funktionen, deren „angegebene und angedeutete Anforderungen unter den spezifizierten Bedingungen“ [Anw] erfüllt sind, bereitgestellt werden. Unter-Charakteristiken:

- Funktionale Vollständigkeit
- Funktionale Korrektheit
- Funktionale Angemessenheit

3.4.2. Performanceeffizienz

Weder bei dem Anwendungsszenario noch bei den Aufgaben ist die Performance relevant oder als Kriterium genannt. Deshalb wird zu diesem Aspekt keine Bewertung durchgeführt. Das Evaluieren des Zeitverhaltens der Pipeline wird von dem Attribut „Geschwindigkeit“ aus [Big] übernommen.

3.4.3. Kompatibilität

Innerhalb derselben Hardware oder Softwareumgebung können „Informationen mit anderen Produkten, Systemen oder Komponenten [ausgetauscht] und/oder die vorausgesetzten Funktionen [durchgeführt]“ [Anw] werden. Unter-Charakteristiken:

- Koexistenz
- Interoperabilität

3.4.4. Verwendbarkeit

„Das Erreichen von spezifizierten Zielen in einem spezifizierten Kontext mit Effektivität, Effizienz und Zufriedenstellung“ [Anw] Unter-Charakteristiken:

- Eignungserkennbarkeit
- Lernbarkeit
- Bedienbarkeit
- Nutzerfehlerschutz
- Benutzerinterface-Ästhetik
- Zugänglichkeit

3.4.5. Zuverlässigkeit

Für die spezifizierten Bedingungen und spezifizierte Zeit können die angegeben Funktionen ausgeführt werden. Unter-Charakteristiken:

- Ausgereiftheit
- Verfügbarkeit
- Fehlertoleranz
- Wiederherstellbarkeit

3.4.6. Sicherheit

Weder bei dem Anwendungsszenario noch bei den Aufgaben ist die Sicherheit relevant oder als Kriterium genannt. Deshalb wird zu diesem Aspekt keine Bewertung durchgeführt.

3.4.7. Wartbarkeit

„Die Effektivität und Effizienz, mit denen [das] Produkt oder System modifiziert werden kann, um es zu verbessern oder an Änderungen in der Umgebung und den Anforderungen anzupassen.“ [Anw] Unter-Charakteristiken:

- Modularität
- Wiederverwendbarkeit
- Analysierbarkeit
- Modifizierbarkeit
- Testbarkeit

3. Verwandte Arbeiten

3.4.8. Portabilität

Effektivität und Effizienz, mit der das Produkt „von einer Betriebs- oder Verwendungs-Umgebung zu einer anderen übertragen werden kann“ [Anw]. Unter-Charakteristiken:

- Adaptierbarkeit
- Installierbarkeit
- Ersetzbarkeit

3.5. CI/CD-Pipeline

Es wurde die Qualität der im Rahmen dieser Ausarbeitung geschriebenen Eclipse-Plug-Ins bewertet. Im Rahmen dieser Ausarbeitung wurde eine Implementierung einer Pipeline für die Eclipse-IDE umgesetzt, damit die Entwicklungspraxis CI/CD ermöglicht wird. Es wurde geprüft, wie gut sie die Eigenschaften und Funktionen von CI/CD-Pipelines nach Stephen J. Bigelow [Big] erfüllt.

Hier folgt eine Zusammenfassung aus dem Englischen von [Big]:

Geschwindigkeit: Schnelle Durchführung von vielen Schritten bzw. einer großen Pipeline.

Konsistenz: Der automatische Ablauf soll immer dieselben Ergebnisse liefern. Sowohl Variationen als auch manuelle Schritte „verlangsamen die Pipeline und laden Fehler und Ineffizienz ein.“ [Big]

Enge Versionskontrolle: Eine integrierte zuverlässige Versionskontrolle, um mögliche erforderliche Rückgängigmachungen jederzeit durchführen zu können.

Automatisierung: Extensive Automatisierung bringt den Code durch die Pipeline mit so wenig manueller Interaktion wie möglich.

Integrierte Rückmeldungsschleifen: Bei Problemen möglichst frühe Reaktionen und ein möglichst früher Abbruch, um diese (die Probleme) möglichst früh zu beheben.

Durchgängige Sicherheit: Weder bei dem Anwendungsszenario noch bei den Aufgaben ist die Sicherheit relevant oder als Kriterium genannt. Deshalb wird zu diesem Aspekt keine Bewertung durchgeführt.

4. Anwendungsszenario

Dieses Kapitel beschreibt das Anwendungsszenario, das größtenteils aus der Ausarbeitung von Stürner [Stü22] übernommen wurde. Hier dient es als gemeinsamer Vergleichspunkt, um die Ergebnisse der im Rahmen dieser Ausarbeitung entwickelten Plug-Ins und Abläufe mit den Ergebnissen der Software, die Stürner entwickelt hat, zu vergleichen.

In Abschnitt 4.1 „Das „kooperatives Überholen“-Szenario“ wird das „kooperative Überholen“-Szenario beschrieben. Dieses dient als Beispielverhalten für zwei Roboterautos. In Abschnitt 4.2 „Die Roboterauto-Zielplattform“ wird hinsichtlich der Roboterautos die Hardwareplattform, die im Rahmen dieser Ausarbeitung für diese Ausarbeitung verwendet wird, beschrieben.

4.1. Das „kooperatives Überholen“-Szenario

Dieses Szenario kommt häufig in verschiedenen Formen in Ausarbeitungen zu cyberphysischen Systemen und MUML [BCD+14] [BDG+] [Poh18] vor. In dieser Ausarbeitung wird jedoch die Variante aus Stürners Ausarbeitung [Stü22] wiederverwendet und er (Stürner) wiederum wurde von vorherigen Forschungsarbeiten zu dieser inspiriert.

Das Szenario selbst beschreibt den folgenden Kontext und die erwarteten Verhaltensweisen:

- Ein langsames und ein schnelles Auto fahren auf der gleichen Straße. Für die folgenden Beschreibungen werden beiden Autos zwei Namen zugewiesen: Dem schnellen Auto „Überholer“ und dem langsamem „Partner“.
- Die Straße besteht aus zwei Spuren:
 - Die rechte Spur, auf der beide Autos starten, und
 - die linke Spur, die zum Überholen dient.
- Der Überholer startet hinter dem Partner und nähert sich diesem aufgrund seiner höheren Geschwindigkeit.
- Der Überholer versucht, den Partner über die Überholspur zu überholen. Beim Überholen muss er eine höhere Geschwindigkeit als der Partner haben.
- In diesem Szenario wird angenommen, dass beide Autos ihre Zielgeschwindigkeiten einhalten, sodass beispielsweise der Partner nicht während des Überholmanövers beschleunigt.
- Die hierfür notwendige Kooperation zwischen den beiden Autos erfolgt, indem sie miteinander kabellos kommunizieren und so den Vorgang koordinieren. Der Überholer wird den Partner erst dann überholen, sobald dieser dem Überholvorgang zugestimmt hat. So kann sich der Überholer sicher sein, dass der Partner während des Überholmanövers nicht beschleunigt.

4. Anwendungsszenario

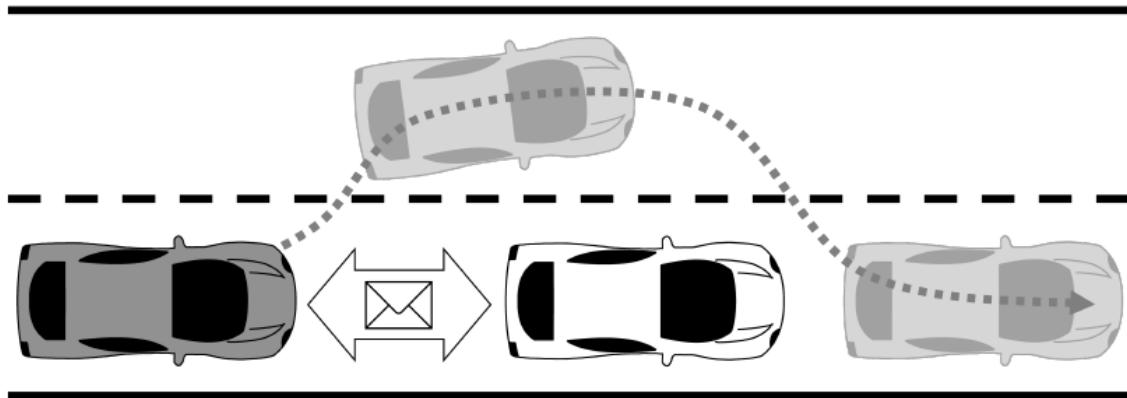


Abbildung 4.1.: Überblick über das kooperative Überholmanöver mit zwei autonomen Fahrzeugen.
Dieses Bild stammt aus Stürners Ausarbeitung [Stü22].
Das graue Auto stellt den Überholer dar und das weiße Auto den Partner.

- Das Szenario hat als vereinfachende Annahme, dass jedes gestartete Überholmanöver nicht abgebrochen wird, sondern erfolgreich durchgeführt wird.

Das Szenario wird in Abbildung 4.1 visualisiert

4.2. Die Roboterauto-Zielplattform

Für die Laborversuche des oben erklärten Szenarios wird in diesem Abschnitt die Roboterauto-Plattform bzw. ihr grober Hardwareaufbau beschrieben. Wie bereits in Abschnitt 1.1 „Problemstellung“ dargelegt, wurde nach Stürners Ausarbeitung eine neue Variante mit einem anderen Aufbau entwickelt.

Die Unterschiede liegen hauptsächlich in der Lenkung: Diese erfolgt in der neuen Variante über eine Lenkachse, die von einem Servo bewegt wird. Das Antreiben des Autos erfolgt durch zwei Motoren, an denen die hinteren Räder montiert sind und somit als Hinterradantrieb dienen. Ein Kamerabild hiervon ist Abbildung 4.2. So ist diese Variante der Roboterautos näher an den Autos der Realität. Bei der Abstandsmessung werden durch eine Modifikation weiterhin zwei Ultraschallsensoren verwendet, von denen einer vorne montiert ist und nach vorne misst. Der andere ist hinten montiert und misst in dieser Variante nach rechts. Auf diese Weise soll dieser beim Überholmanöver das Roboterauto mit der Partnerrolle zuverlässig erkennen können. Das Aufspüren der die Spuren symbolisierenden Linie erfolgt über drei vorne montierten Infrarotsensoren. Die Kommunikation erfolgt über ein WiFi-Modul.

Die Darstellung der verteilten Computersysteme erfolgt weiterhin durch zwei AMKs, die in Stürners Ausarbeitung als „Electric Control Units“ (ECUs) bezeichnet werden. Hierbei übernimmt der Koordinator-AMK die Kommunikation und der Fahrer-AMK das Fahren mittels der Sensorwerte und der Elektromechanik, d.h. der Elektromotoren und dem Servo.

Das Kommunikationsprotokoll zwischen den beiden AMKs wurde während dieser Ausarbeitung von „Inter-Integrated Circuit“ (I2C) nach „Seriell“ geändert.

Die Abbildung 4.3 enthält zwei Kamerabilder der für diese Ausarbeitung verwendeten Roboterautos. Für die Nutzung der Hardware durch einen AMK wird die Bibliothek „Sofdcar-HAL“ [Reid]

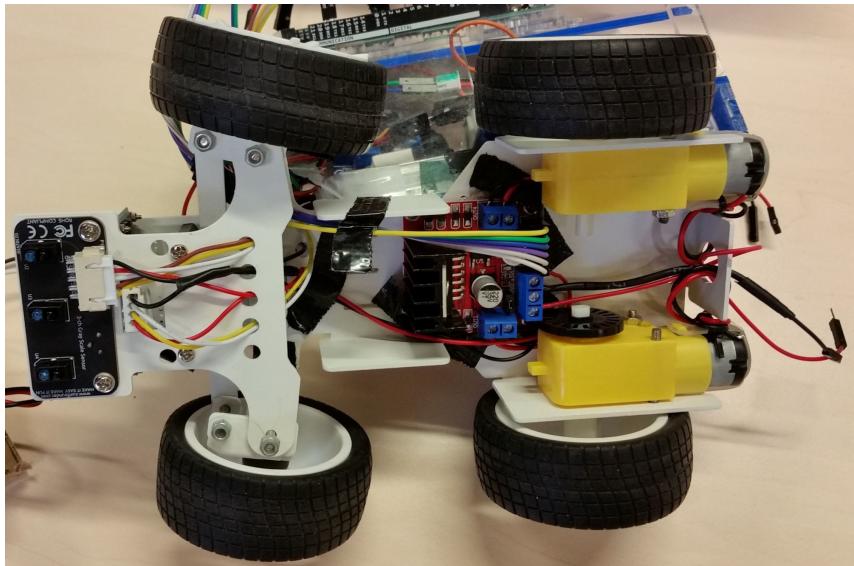
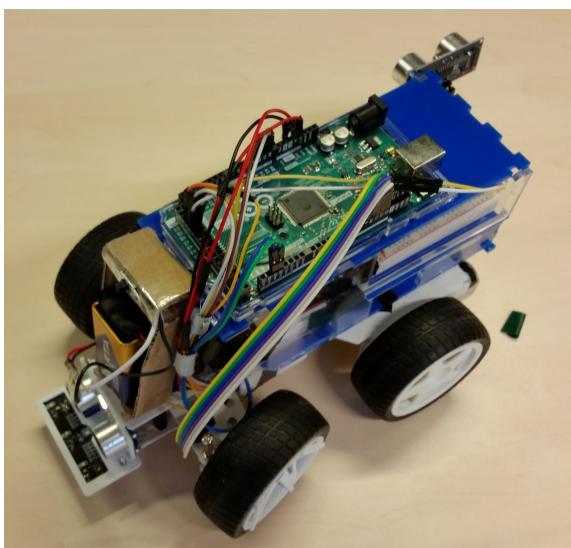
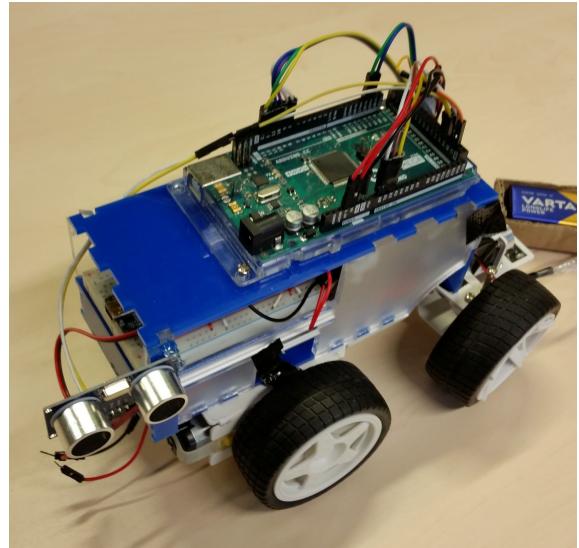


Abbildung 4.2.: Kamerabild der Unterseite eines für diese Ausarbeitung verwendeten Roboterautos.
Die andere Konstruktion des Antriebes und der Lenkung sind erkennbar.



(a) Ansicht von vorne links.



(b) Ansicht von hinten rechts.

Abbildung 4.3.: Kamerabilder eines für diese Ausarbeitung verwendeten Roboterautos.
Die 9V-Blockbatterie und seine Halterung sind Teil der Energieversorgung des WiFi-Modules.

4. Anwendungsszenario

verwendet. Die serielle Kommunikation zwischen den beiden AMKs erfolgt mittels einer von Georg Reißner geschriebenen internen Bibliothek. Sonst wird auf die von Stürner geschriebenen Bibliotheken und Implementierungen zurückgegriffen. Für weitere Details siehe Ergänzungsmaterial B „Ergänzungsmaterial Übersicht und Details zu den Roboterautos“.

5. Analyse

Die in der Einführung erwähnte manuelle Ausführung musste zunächst untersucht werden, um Änderungen oder Automatisierungen zu ermöglichen.

Hierfür wird in diesem Kapitel eine Übersicht an Arbeitsschritten mit einer jeweiligen Beschreibung der jeweils relevanten Informationen aufgestellt.

Hierbei wiederum werden zum besseren Nachschlagen oder Vergleichen die T-Nummerierungen der Schritte anhand des Schaubildes des Codegenerationsprozesses aus der Anleitung für die Nutzerschaft [Coda] mit angegeben (siehe Abbildung 5.1).

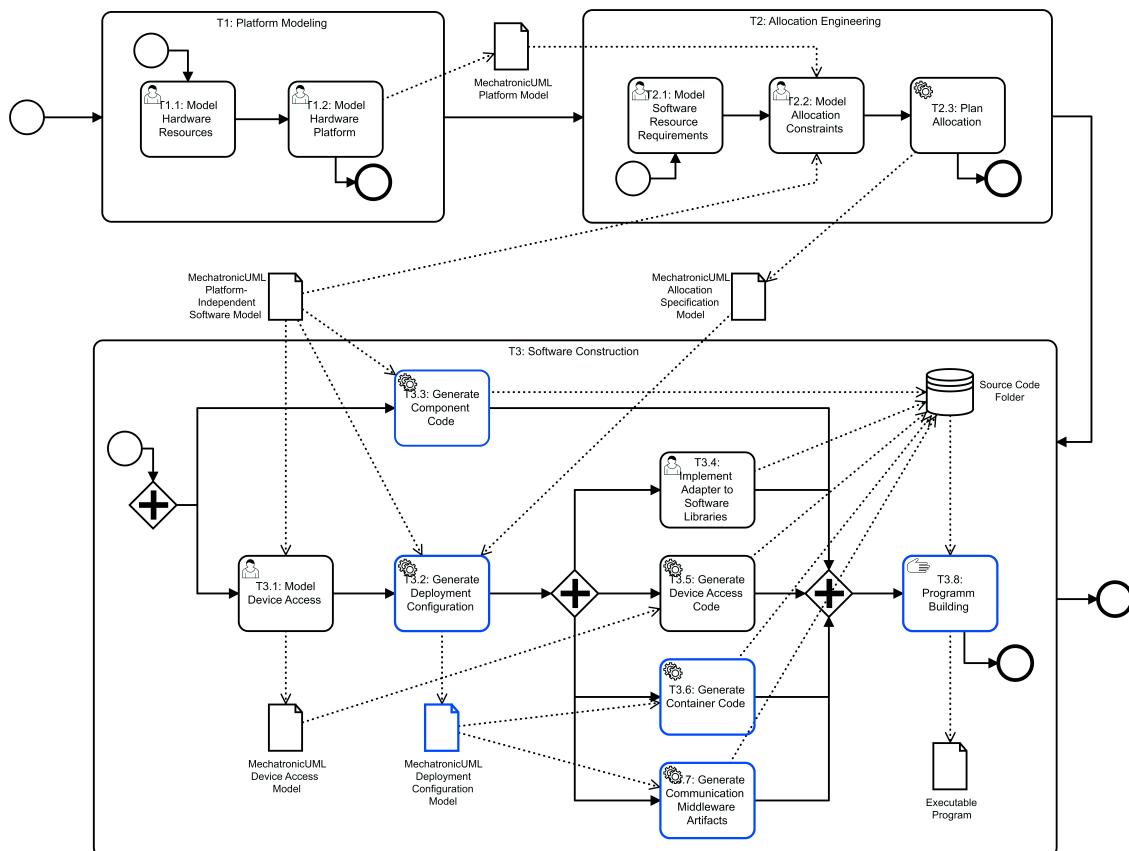


Abbildung 5.1.: Der Prozess der Codegeneration wie abgebildet in der Anleitung für die Nutzerschaft [Coda], aber mit einer Änderung: Der Schritt T3.3 wurde blau markiert.

Im Rahmen dieses Kapitels werden also die folgenden Forschungsziele beantwortet:

- FZ1.1 Welche Schritte des Arbeitsablaufes werden von welchen Komponenten und mit welchen Daten ausgeführt?

5. Analyse

FZ1.2 Welche Schritte des Arbeitsablaufes müssen manuell durchgeführt werden?

Davon wird FZ1.1 durch die folgenden Beschreibungen der Arbeitsschritte nach der Anleitung für die Nutzerschaft [Coda] beantwortet.

Arbeitsschritt 1: T3.2: „Deployment Configuration aka Container Transformation“[Coda], also Container-Transformation

Hier wird per Export-Operation „MechatronicUML“ /„Container Model and Middleware Configuration“ mit den Einstellungen für die Middleware (hier „MQTT and I2C Middleware Configuration“) und einem Zielort aus der Datei „roboCar.muml“ die Datei „MUML_Container.muml_container“ mit den verschiedenen Containern für die verschiedenen Komponenten und deren Konfigurationen generiert. MQTT steht hierbei für „Message Queuing Telemetry Transport“[Stü22]. Hierbei wird intern anhand der Systemallokationen-Ressource in „roboCar.muml“ gearbeitet.

Für das interne Starten der Transformation ist der Export-Wizard ContainerWizard aus dem MUML-Plug-In-Projekt „PlatformSpecificModeling“ bzw. seinem Unterprojekt „org.muml.psm.container.transformation“ verantwortlich.

Arbeitsschritt 2: T3.6 und T3.7: „Container Code Generation“[Coda], also Generation des Containercodes

Dies wird per [Rechtsklick auf die in dem vorherigen Schritt generierte „MUML_Container.muml_container“-Datei]/„mumlContainer“/„Generate Arduino Container Code“ gestartet und es erstellt einen Ordner namens „arduino-containers“. In diesem befinden sich der Ordner „APIMappings“ für die verschiedenen APIs und die Ordner mit der Namensendung „ECU“ für die verschiedenen Electric Control Unit (ECU)s bzw. AMKs. In dem letzteren befinden sich Code-Dateien wie einige interne Bibliotheken, z.B. die Dateien „I2CCustomLib.cpp“ und „I2CCustomLib.hpp“ aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“, Package „org.mumlarduino.adapter.container“ und Ordnerverzeichnis „resources/container_lib“. Jedoch fehlen dem generierten Arduinocode, der offiziell als Sketch bezeichnet wird [Arda], noch verschiedene andere Codedateien, beispielsweise für das Verhalten in den verschiedenen Fahrzuständen.

Intern wird GenerateAll aus dem Package „org.mumlarduino.adapter.container.ui.common“ aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“ bzw. dessen Unterprojekt „org.muml.codegen.componenttype.export.ui“ verwendet, um die Generierung zu starten.

Arbeitsschritt 3: T3.3: „Component Code Generation“[Coda], also Generation der Komponentencodes

Hier werden per Export-Operation „MechatronicUML“/„Source Code_workspace“ mit den Einstellungen für die Zielplattform (hier „Component Type ANSI C99“) und einem Zielort aus der Datei „roboCar.muml“ die Komponentencodes generiert und in dem Eclipse-Projektordner in dem Ordner „fastAndSlowCar_v2“ platziert.

Das interne Starten der Codegeneration erfolgt über die Klasse C99SourceCodeExport und deren Methode generateSourceCode(EObject element, IContainer targetFolder, IProgressMonitor monitor) aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“ bzw. dessen Unterprojekt „org.muml.c.adapter.componenttype.ui“.

Arbeitsschritt 4: T3.8: „Program Building“[Coda]

Diese Phase besteht genau genommen aus dem Post-Processing und dem Deployment bzw. Hochladen auf die verschiedenen AMKs.

Beim Post-Processing wird zunächst der generierte, aber noch nicht direkt verwendbare, sondern unvollständige Code in einen direkt verwendbaren und vollständigen Zustand gebracht. So kann die Arduino-IDE die verschiedenen Sketches korrekt kompilieren.

Stürner hat für seine Modelle und Roboterautos einen passenden Post-Processing-Ablauf in seiner Ausarbeitung [Stü22] beschrieben. Hierbei erklärt er aber auch, dass das Post-Processing infolge eines nicht korrekt kompilierbaren Codegenerator-Quellcodes notwendig ist. Für die während dieser Ausarbeitung verwendete Version der Roboterautos und der Bibliothek „Sofdcar-HAL“ kann der hinsichtlich „Sofdcar-HAL“ bereits angepasste Verlauf des Post-Processings unter [Reib] nachgeschlagen werden.

Das Deployment besteht darin, die verschiedenen Sketches mit der Arduino-IDE zu öffnen und jeweils auf den entsprechenden AMK hochzuladen [Stü22].

FZ1.2 kann mit der folgenden Zusammenfassung zu den Beschreibungen von oben beantwortet werden:

In den Arbeitsschritten 1 und 3 wird jeweils von der Nutzerschaft ein Export-Wizard aufgerufen, die Einstellungen festgelegt und der Export-Vorgang durch das Drücken auf den „Finish“-Knopf gestartet. Danach läuft intern der jeweilige Prozess automatisch ab.

Für Schritt 2 wird von Hand ein Rechtsklick auf die „MUML_Container.muml_container“-Datei durchgeführt und in dem Kontextmenü wird der Eintrag „Generate Arduino Container Code“ ausgewählt. So wird die Generierung der ersten Codedateien gestartet.

Von Schritt 4 ist das Post-Processing ein langer und gänzlich manueller Ablauf, der von der Nutzerschaft befolgt werden muss. Dies bedeutet, dass bisher von Hand viele Dateien kopiert oder verschoben werden mussten. In manchen mussten die Pfade in den #include-Anweisungen und das Einbinden von „clock.h“ korrigiert werden. In anderen Fällen musste ein entsprechender Befehl nachgetragen werden. Werte, wie z.B. die von dem jeweiligen Roboterauto zu verwendende Geschwindigkeit, mussten passend eingetragen werden. Dasselbe gilt für die Anmeldeinformationen des zu verwendenden WLAN-Netzwerks und für die Verbindungsinformationen mit dem zu nutzenden MQTT-Server. Bei der neueren Version der Roboterautos musste die Nutzung der „Sofdcar-HAL“-Bibliothek nachgetragen werden.

Nach dem Post-Processing werden beim Deployment die verschiedenen Sketches über das Ordnersystem geöffnet. Bei jedem Sketch werden entsprechend Boardtyp und Verbindungsport eingestellt sowie der Uploadvorgang gestartet.

Hierbei ist wichtig, dass alle geschilderten Schritte von der Nutzerschaft durchgeführt werden müssen, aber innerhalb von Eclipse keine Anleitung über die einzuhaltende Reihenfolge der auszuführenden Wizards und zu nutzenden Menüelemente gegeben ist bzw. angezeigt wird.

Stattdessen müssen die Anleitungen von zwei Git-Seiten und deren Einstellungen exakt befolgt werden:

1. Die Anleitung für die Nutzerschaft [Coda] beschreibt die Verwendung der MUML-Werkzeuge von den Modellen bis zu unvollständigen Sketches.
2. Die Seite [Reib] beschreibt die vielen Nachbearbeitungsschritte, durch die der Code der Sketches vervollständigt wird und so für die Arduino-Software kompilierbar wird.

5. Analyse

Beim Hochladen muss die Nutzerschaft darauf achten, dass hinsichtlich der Roboterautos bzw. deren AMKs und dem jeweiligen Sketch keine Verwechslungen passieren oder unbemerkt bleiben.

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

Dieses Kapitel behandelt den Entwurf der CI/CD-Pipeline, durch die alle Arbeitsschritte nicht mehr von dem Nutzer oder der Nutzerin durchgeführt werden müssen. Stattdessen ist nach dem Modellieren mit MUML und dem Konfigurieren der Abläufe der Automatisierung nur noch das Starten der Pipeline erforderlich.

Die Anforderungen an die Pipeline selbst folgen hauptsächlich aus den Artikeln von Fowler[Fow20], RedHat[Hata][Hatb][Kab] und Tuli[Tul]. Die Konfigurationen werden im YAML-Format [Tea] niedergeschrieben, damit das Lesen und Schreiben von diesen leichter fällt.

Zur erhöhten Flexibilität bei der Nutzung der einzelnen Anwendungen der MUML-Werkzeuge für die Container-Transformation, die Generation des Container-Codes und die Generation des Komponenten-Codes wurde beschlossen, dass für diese auch eigene Konfigurationsmöglichkeiten für deren einzelne automatische Ausführung integriert werden sollten. Dasselbe gilt auch für das Post-Processing in Form einer eigenen Liste.

Aufgrund des Arbeitens mit der MUML-Tool-Suite und somit innerhalb der Eclipse-IDE muss beachtet werden, dass Build-Prozesse nur lokal auf einem der Arbeitscomputer erfolgen können. Möglichkeiten für das Ausführen von Eclipse online konnten nicht gefunden werden.

Hier folgt eine Auflistung der Anforderungen:

ANF 6.1 Konfigurationen erfolgen im YAML-Format

ANF 6.2 Frühes Feedback bei Fehlern

ANF 6.3 Automatische Tests der lokalen Arbeitskopie

ANF 6.4 Automatischer Build-Prozess

ANF 6.5 Schneller Build-Prozess

ANF 6.6 Automatisches Hochladen von der lokalen Arbeitskopie auf die Versionsverwaltung

ANF 6.7 Automatisches Herunterladen des aktuellen Codestandes von der Versionsverwaltung

ANF 6.8 Automatisches Deployment

ANF 6.9 Konfigurationsmöglichkeit für die einzelne automatische Ausführung von den MUML-Werkzeugen für die Container-Transformation, die Generation des Container-Codes und die Generation des Komponenten-Codes sowie dem Post-Processing

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

Die verworfenen Entwürfe werden nicht beschrieben, sondern nur der umgesetzte Ansatz.

Die Beschreibung zu diesem ist wie folgt aufgeteilt:

In 6.1 „Geplanter Ablauf für die Nutzung der CI/CD-Pipeline“ wird der geplante Ablauf für die Nutzung der CI/CD-Pipeline beschrieben. Hierbei werden beiläufig an den entsprechenden relevanten Stellen zu den ausreichend erfüllbaren Anforderungen die Lösungen auf einem hohen Detaillevel bzw. auf einem konzeptionellen Level beschrieben.

In 6.2 „Nicht oder unzureichend erfüllbare Anforderungen“ werden die nicht oder unzureichend erfüllbaren Anforderungen erläutert.

In 6.3 „Weitere Aspekte“ werden weitere Aspekte und deren zugrunde liegenden Entscheidungen erklärt.

In 6.4 „Spezifische Pipelineschritte“ werden beispielhaft sechs Pipelineschritte, die sich aus den Anforderungen ergeben, beschrieben.

In 6.5 „Aufbau der Konfigurationsdatei“ wird der Aufbau der Konfigurationsdatei detaillierter erläutert.

6.1. Geplanter Ablauf für die Nutzung der CI/CD-Pipeline

In diesem Unterkapitel wird der Ablauf, der auf Basis der Anforderungen entworfen wurde, beschrieben. Wie bereits oben erwähnt, beschränken sich die Erklärungen zu den Lösungen auf einem hohen Detaillevel bzw. auf einem konzeptionellen Level. Für eine Ansicht auf einem niedrigeren Level mit Namen und Beispielkonfigurationen siehe Abschnitt 6.4 „Spezifische Pipelineschritte“. Für den Arbeitsablauf mit der Vorbereitung und Nutzung der CI/CD-Pipeline sind folgende Arbeitsphasen und deren Schritte vorgesehen (siehe auch Abbildungen 6.1, 6.2 und 6.3, wobei diese auf die Pipeline selbst begrenzt sind):

6.1. Geplanter Ablauf für die Nutzung der CI/CD-Pipeline

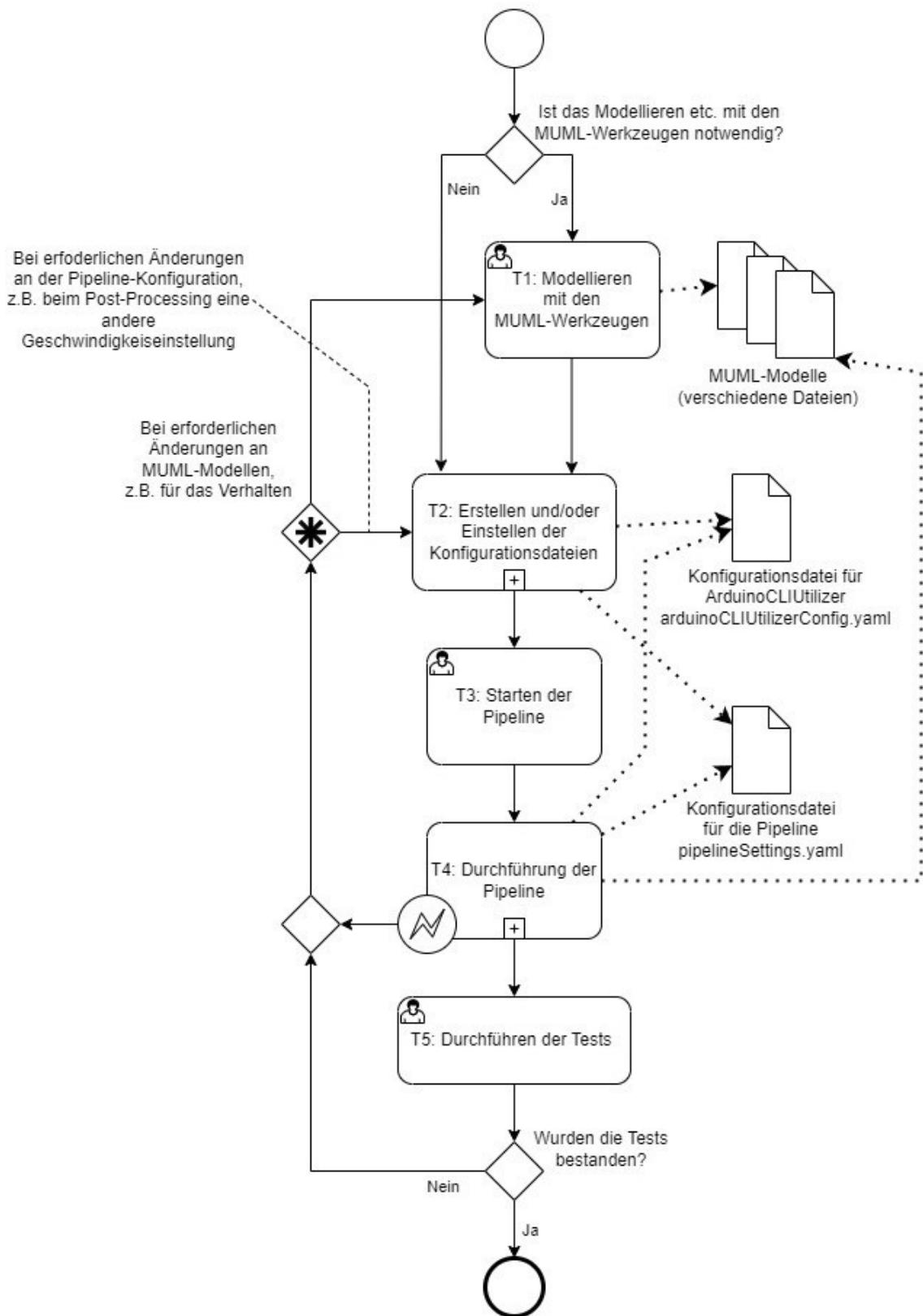


Abbildung 6.1.: Übersicht des Nutzungsablaufes

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

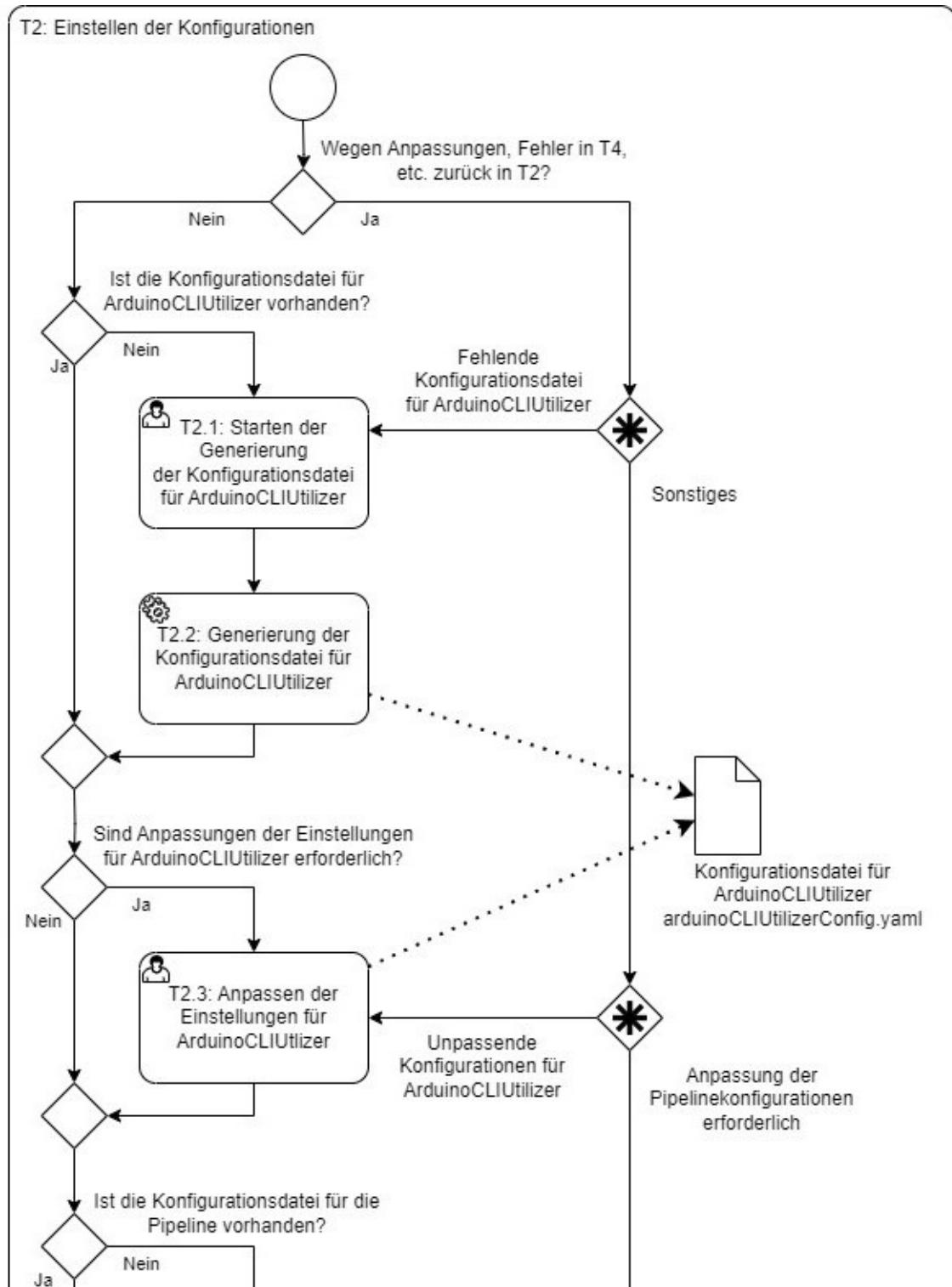


Abbildung 6.2.: Übersicht des Konfigurationsablauf, Teil 1

6.1. Geplanter Ablauf für die Nutzung der CI/CD-Pipeline

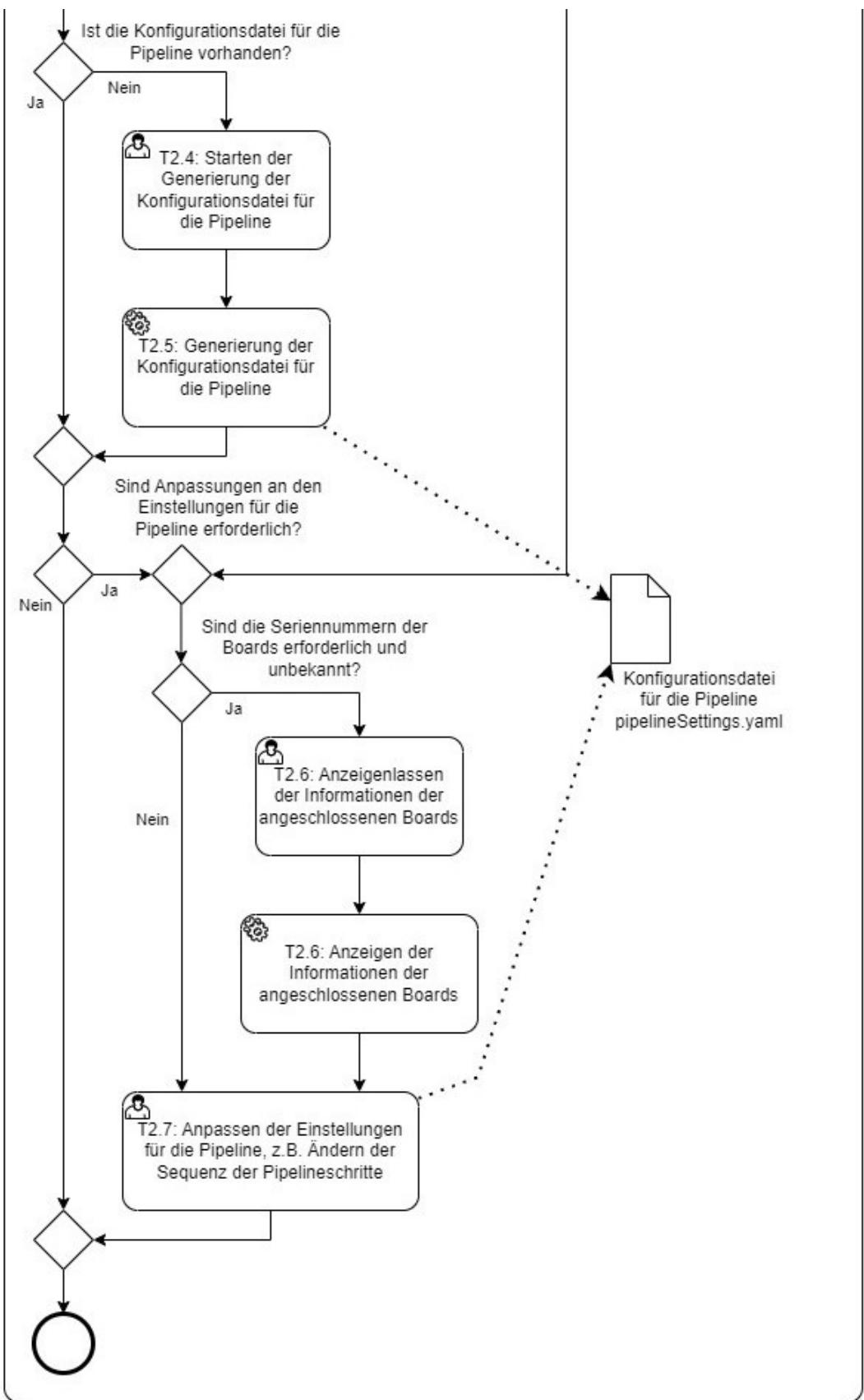


Abbildung 6.3.: Übersicht des Konfigurationsablauf, Teil 2

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

Phase 1 Sofern notwendig, wird zuerst mit den MUML-Werkzeugen modelliert oder andere Arbeiten vorgenommen (siehe T1 in Abbildung 6.1).

Phase 2 Anschließend wird mit den Konfigurationsdateien gearbeitet (Siehe T2 in Abbildungen 6.1, 6.2 und 6.3), die im YAML-Format geschrieben sind. In Anbetracht der einfachen Schreibweise des YAML-Formats als Auswahlgrund werden nur die leicht und schnell verstehbaren und erlernbaren Aspekte von YAML, also Skalare bzw. Zeichenketten, Mappings, Sequenzen und Kommentare verwendet. Folglich werden Konzepte wie das Integrieren von mehreren Dokumenten in einer Datei oder Tags nicht verwendet. So wird ANF6.1 erfüllt.

Phase 2.1 Falls die Konfigurationsdateien noch nicht vorhanden sind, so kann deren Generierung gestartet werden (siehe T2.1 und T2.2 in Abbildung 6.2 sowie T2.4 und T2.5 in 6.3). Hierbei entspricht die Beispielkonfiguration in „arduinoCLUtilizerConfig.yaml“ der vorgesehenen Installation der ArduinoCLI. Falls die Arduino-CLI an einem anderen Pfad installiert ist oder der Zugriffspfad auf sie bereits in den Systemeinstellungen vorhanden ist oder beides, können auch schon zu diesem Zeitpunkt die Einstellungen für sie (die Arduino-CLI) eingetragen werden (siehe T2.3 in Abbildung 6.2). Im Fall der Datei „pipelineSettings.yaml“ sind die Beispieleinstellungen auf das in Kapitel 4 „Anwendungsszenario“ beschriebene Anwendungsszenario des Überholvorganges und dessen MUML-Modellen für die Roboterautos sowie den im Rahmen dieser Ausarbeitung vorgenommenen Änderungen (siehe Abschnitt 4.2 „Die Roboterauto-Zielplattform“) orientiert. Es kann aber auch eine Datei „Each_pipeline_step_by_an_example.txt“ generiert werden lassen, die eine Sammlung von allen Piepline-Schritten mit einer jeweiligen Beispielkonfiguration darstellt.

Phase 2.2 Vornehmen der Anpassungen an den Konfigurationsdateien: Sofern die Konfigurationen für die ArduinoCLI noch nicht geschehen sind oder noch nicht korrekt passen, sollten sie vorgenommen werden (siehe T2.3 in Abbildung 6.2). Die Einstellungen der Pipeline (siehe T2.7 in Abbildung 6.3) wie das Hinzufügen der gewünschten Schritte sind für diesen Zeitpunkt vorgesehen. Falls die Seriennummern für die automatischen Auswahlen der angeschlossenen Boards benötigt werden, so können diese auch angezeigt werden.

Der erste Teil von ANF 6.9 wird über das Konzipieren eines eigenen Bereiches für die Einstellungen der einzelnen automatischen Ausführungen von den MUML-Werkzeugen (die Container-Transformation, die Generation des Container-Codes und die Generation des Komponenten-Codes) erfüllt. So kann auch beispielsweise nach größeren Änderungen an den verschiedenen MUML-Modellen nur der rohe Code generiert werden lassen, um diesen für die Anpassungen an den Post-Processing-Schritten zu untersuchen, ohne Änderungen an der eigentlichen Pipeline durchzuführen zu müssen. Der zweite Teil der Erfüllung von ANF6.9, also die Konfigurierbarkeit des Post-Processings, wird über eine eigene konfigurierbare Liste bzw. Sequenz umgesetzt. Für weitere Details hinsichtlich des Aufbaus und der Nutzung siehe Abschnitt 6.3.2 „Eigene Sequenz für das Post-Processing“.

6.1. Geplanter Ablauf für die Nutzung der CI/CD-Pipeline

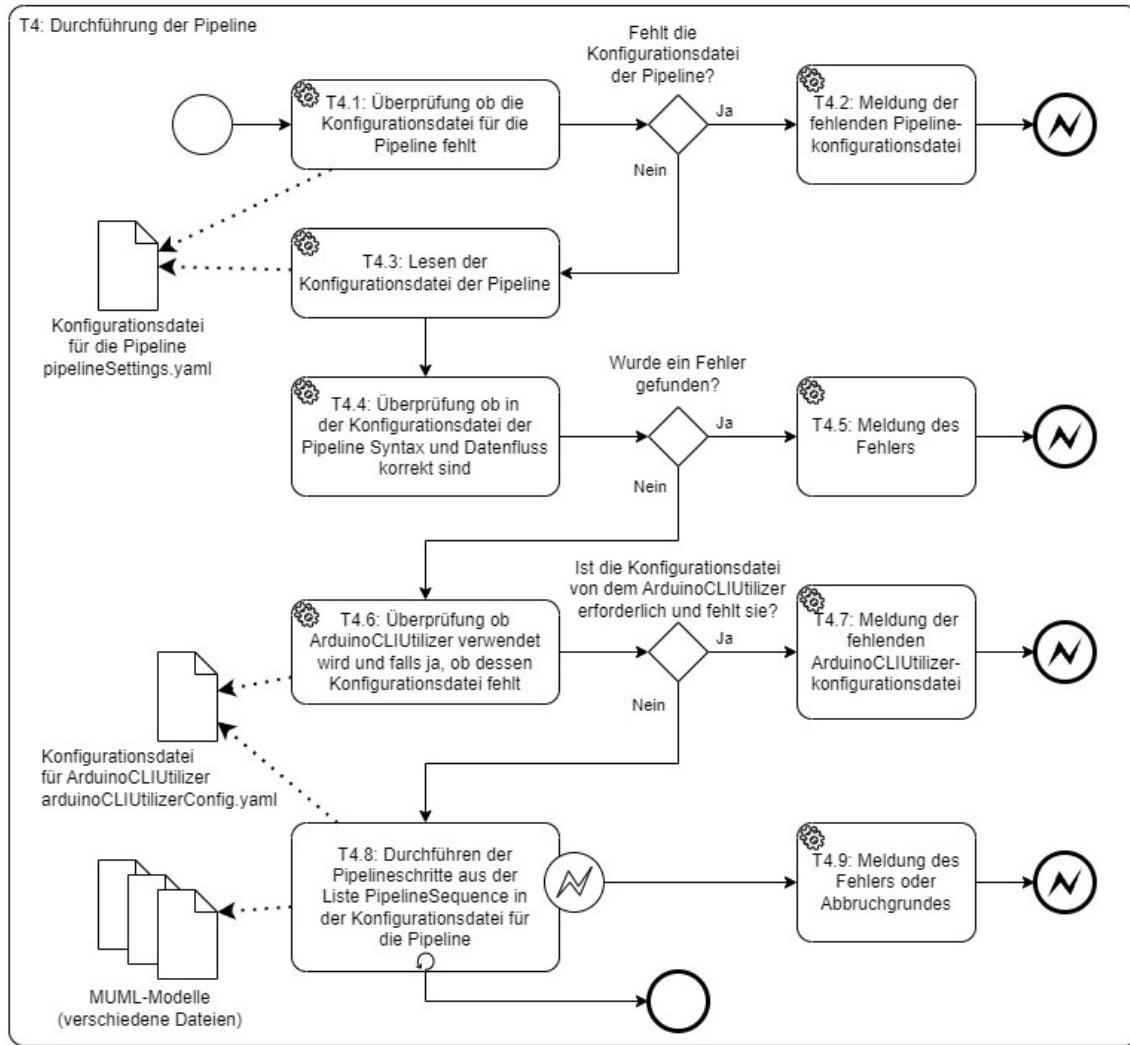


Abbildung 6.4.: Übersicht des Durchführungsablaufes

Phase 3 Dann wird die Ausführung der Pipeline gestartet (siehe T3 in Abbildung 6.1). Dieser Teil der Software läuft wie folgt ab:

Phase 3.1 Es wird dabei zunächst nacheinander geprüft, dass die Konfigurationsdatei für die Pipeline vorhanden ist (siehe T4.1 in Abbildung 6.4). Falls sie fehlt, wird der Vorgang abgebrochen und eine entsprechende Fehlermeldung angezeigt (siehe T4.2 in Abbildung 6.4).

Phase 3.2 Anschließend wird sie gelesen (siehe T4.3 in Abbildung 6.4). Falls Syntaxfehler und Datenflussfehler gefunden werden (siehe T4.4 in Abbildung 6.4), so erfolgt auch hier ein Abbruch mit entsprechender Fehlermeldung (siehe T4.5 in Abbildung 6.4).

Phase 3.3 Falls der ArduinoCLIUtilizer durch mindestens einen auf ihn basierenden Schritt verwendet wird, wird das Vorhandensein der Konfigurationsdatei für ihn überprüft (siehe T4.6 in Abbildung 6.4). Falls diese Konfigurationsdatei fehlt, erfolgt ebenfalls ein Abbruch mit Fehlermeldung (siehe T4.7 in Abbildung 6.4).

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

Phase 3.4 Danach findet die eigentliche Ausführung der Pipelineschritte aus der Pipelinesequenz bzw. der eigentlichen auszuführenden Pipeline in PipelineSequence statt (siehe T4.8 in Abbildung 6.4). Falls ein Fehler auftritt oder ein OnlyContinueIfFulfilledElseAbort-Schritt ausgelöst wird, so wird die Ausführung abgebrochen und eine Fehlermeldung angezeigt. In diesem Fall geht die Ausführung je nach Fehlergrund zu einem früheren Schritt zurück, um von diesem aus den Fehlergrund zu beheben.

Dieser Ablauf stellt einen automatisierten Build-Prozess von .hex-Dateien für die Roboterautos ausgehend von MUML-Modellen und somit die Erfüllung von ANF 6.4 dar. Durch die in der vorhergegangenen Ablaufbeschreibung erwähnten Abbruchbedingungen und deren Feedback sowie der beschriebenen Verwendung von OnlyContinueIfFulfilledElseAbort ist zwischen jedem Pipelineschritt frühes Feedback bei Fehlern möglich und somit ist ANF 6.2 erfüllt. ANF 6.8 wird durch den Pipelineschritt Upload erfüllt, der das Hochladen auf einen unter dem angegebenen Port angeschlossenen Board repräsentiert. Zwecks Modularisierung soll die dafür verantwortliche Klasse in ArduinoCLIUtilizer dabei zwischen kompilierten Dateien bzw. .hex-Dateien und unkompliiertem Arduinocode bzw. .ino-Dateien unterscheiden und entsprechend einen internen Kompilervorgang einschieben.

Beim Planen und Überprüfen der einzelnen Abläufe für die Pipelineschritte wurde bemerkt, dass im Fall des Anwendungsszenarios das Heraussuchen eines Boards mehrmals benötigt wird und dass dabei von der ArduinoCLI das Suchen und Auflisten der angeschlossenen Boards bzw. deren Daten und Anschlüssen immer ein paar Sekunden benötigt. Deshalb werden in der Implementierung (siehe Kapitel 7.3 „Vollständige Automatisierung per Pipeline“) von diesem Schritt die Suchergebnisse intern gespeichert und wiederverwendet, damit beim n-maligen Heraussuchen von dem jeweils gewünschten Board aus m angeschlossenen Boards die zeitliche Komplexität nur noch in $O(1 * „angeschlossene Boards suchen“ + n * „gewünschtes Board aus der Ergebnisliste heraussuchen“)$ bzw. $O(1 * „angeschlossene Boards suchen“ + n * m)$ liegt.

Weitere Optimierungsmöglichkeiten konnten nicht gefunden werden, aber später haben Tests gezeigt, dass für das Überholmanöverszenario (siehe Kapitel 4 „Anwendungsszenario“) der komplette Ablauf deutlich schneller erfolgt (siehe Abschnitt „8.2.4.1 Geschwindigkeit: Erfüllungsstufe: 3“). Auf diese Weise wird ANF 6.5 erfüllt.

Phase 4 Falls alles ausgeführt werden konnte, kann der Nutzer oder die Nutzerin Tests oder andere Handlungen durchführen. Je nach Ergebnis der Tests ist die Durchführung entweder abgeschlossen oder sie geht auch hier zu einem früheren Schritt zurück.

Auf diese Weise wäre in der Praxis ein Arbeitszyklus mit der Pipeline abgeschlossen.

6.2. Nicht oder unzureichend erfüllbare Anforderungen

Es konnte nicht zu allen Anforderungen eine Lösung gefunden oder ausreichend umgesetzt werden. In diesem Abschnitt werden zu diesen die jeweiligen Gründe beschrieben.

6.2.1. ANF 6.3: Automatische Tests der lokalen Arbeitskopie

Diese Anforderung konnte nicht umgesetzt werden, weil keine Möglichkeiten für das Testen von Arduinocode mit den verwendeten Roboterautos gefunden oder dafür angepasst werden konnten. Dies kann also bei der aktuellen Umsetzung nur manuell nach einem erfolgreichen Build-Prozess erfolgen.

6.2.2. ANF 6: Automatisches Hochladen von der lokalen Arbeitskopie auf die Versionsverwaltung

Hierfür wurde ein Schritt erstellt, der für das automatische Hinzufügen von neuen Dateien, das Committen der Änderungen und das Hochladen auf Git zuständig ist. Es wird aber wegen möglichen Sicherheitsvorkehrungen und anderen Versionsverwaltungssystemen empfohlen, hierfür ein Skript oder Programm zu schreiben oder zu verwenden, das per Pipelineschritt `TerminalCommand` (siehe Abschnitt 6.3.1 „Langlebige Flexibilität über mögliche Terminalaufrufe“) aufgerufen werden kann.

6.2.3. ANF 6.7: Automatisches Herunterladen des aktuellen Codestandes von der Versionsverwaltung

Diese Anforderung wurde nicht umgesetzt, weil unklar ist, ob bei Unterschieden bei den MUML-Modellen zwischen der lokalen Arbeitskopie und dem Entwicklungsstand von der Versionsverwaltung diese (die unterschiedlichen Versionen der MUML-Modelle) korrekt automatisch zusammengeführt werden können oder ob es z.B. bei den IDs zu Fehlern kommen kann.

6.3. Weitere Aspekte

Dieser Abschnitt und seine Unterabschnitte beschreiben weitere Aspekte und deren gewählte Umsetzungen. Diese weiteren Aspekte sind nicht aus den genannten Quellen hervorgegangen, sondern kamen bei einer Suche nach möglichen Schwächen und Verbesserungsmöglichkeiten zum Vorschein. Die im Folgenden beschriebenen Ergebnisse werden ebenfalls für den Aufbau des Systems der CI/CD-Pipeline verwendet.

6.3.1. Langlebige Flexibilität über mögliche Terminalaufrufe

Es ist zum Zeitpunkt dieser Ausarbeitung unklar, welche Anforderungen später an die zu implementierende Pipeline auftreten könnten, also wurde ein eigener Schritt namens `TerminalCommand` für das Nutzen des Terminals bzw. der Befehlszeile beschlossen. So können spätere Nutzer und Nutzerinnen fehlende Pipelinefähigkeiten z.B. durch eigene geschriebene Skripte und deren Aufrufe kompensieren.

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

6.3.2. Eigene Sequenz für das Post-Processing

Zwecks Langlebigkeit und Flexibilität wird der Ablauf des Post-Processings nicht als nur ein Schritt repräsentiert, sondern es wurden hierfür verschiedene Pipelineschritte konzipiert, die als Sequenz verwendet werden sollen. Durch die Verwendbarkeit als eigene Sequenz ist die Aufbereitung des generierten unvollständigen Codes zu einem kompilierfähigen Zustand für z.B. die Arduino-IDE möglich, ohne Änderungen an der eigentlichen Pipeline durchführen zu müssen. Zusätzlich kann dies beim Debuggen von diesem Arbeitsschritt helfen, weil so die anderen Teile der Automatisierung nicht zusätzlich ausgeführt werden und folglich nicht geändert werden müssen. Für weitere Details siehe Abschnitt 6.5.3 „PostProcessingSequence - Einstellungen für die automatische Ausführung von den Post-Processing-Schritten“.

6.3.3. Datenfluss zwischen den eingetragenen Pipelineschritten

Um die Übertragung von Daten von einem eingetragenen Pipelineschritt zu einem anderen zu ermöglichen, wurde beschlossen, dass die verschiedenen Pipelineschrittypen über ihre jeweiligen Ausgabeparameter verfügen sollen. So kann beispielsweise nach einem Suchvorgang nach dem Port eines gewünschten Boards per `LookupBoardBySerialNumber` der gefundene Port für das Hochladen auf das gewünschte Board per `Upload` verwendet werden.

6.3.4. Vermeidung von Redundanz

Um die Konfigurationen für die einzelne automatische Ausführung von den MUML-Werkzeugen für die Container-Transformation, die Generation des Container-Codes und die Generation des Komponenten-Codes sowie dem Post-Processing jeweils nicht redundant auftreten zu lassen, was bei Änderungen Fehlerpotential bedeuten würde, sind Referenzen bzw. Verweise vorgesehen. So wird an der Stelle der Referenz jeweils der entsprechende gewünschte Pipelineschritt eingefügt. Im Fall des Post-Processings werden alle Schritte an der Stelle der Referenz eingefügt.

6.3.5. Leichtes Anfangen des Arbeitens mit der Pipeline

Um das anfängliche Arbeiten mit der Pipeline auch ohne Erfahrung mit dem im Rahmen dieser Ausarbeitung geschriebenen System zu ermöglichen, ist vorgesehen, dass alle Konfigurationsdateien, also die für `ArduinoCLUtilizer` und für die Pipeline selbst, generiert werden lassen können. Für die verschiedenen Schritte der Pipeline ist dasselbe auch in Form einer Textdatei mit allen Schritten und einer jeweiligen Beispielkonfiguration möglich.

6.4. Spezifische Pipelineschritte

In diesem Unterkapitel werden aus dem Arbeitsablauf beispielhaft die sechs Pipelineschritte zu der Container-Transformation, dem Eintragen der WLAN-Einstellungen, dem Kompilieren sowie den Hilfsschritten für das Löschen von Ordnern, dem bedingten Abbrechen der Pipelinedurchführung und für das Anzeigen von Nachrichten beschrieben. Für die Schritte außerhalb des Arbeitsablaufes

erfolgt danach dasselbe für den Hilfsschritt für das Löschen von Ordnern, weil so das Arbeitsverzeichnis zuverlässig von alten Dateien bereinigt wird. Zum besseren Vergleichen und Verstehen wird hierbei jeweils die manuelle Durchführung und der entsprechende Pipelineschritt in Form von den entsprechenden Beispieleinträgen gegenübergestellt. Für eine detaillierte Auflistung und Beschreibung aller Pipelineschritte siehe Ergänzungsmaterial Übersicht der Pipelineschritte.

Als Sonderfall unter den Pipelineschritten wird zusätzlich für das Abbrechen der Pipeline OnlyContinueIfFulfilledElseAbort beschrieben. So erfolgen zugleich die Lösungsbeschreibungen zu ANF 6.2, 6.3, 6.4, 6.8 und 6.9 aus der Perspektive der Nutzungsweise.

Zum Abschluss dieses Kapitels wird kurz auf die Pipelineschritte, die der Flexibilität und Langzeitigkeit des Pipelinesystems dienen, eingegangen.

6.4.1. Modelltransformationen und Codegeneration: Beispiel: Container-Transformation

Zunächst kommt ein Beispiel aus den Arbeitsschritten und Pipelineschritten für die Codegeneration ausgehend von MUML-Modellen nach der Anleitung für die Nutzerschaft [Coda].

Die Transformation von MUML-Modellen zu einer Modelldatei, die für einen Teil der Generation des unvollständigen Codes verwendet wird, erfolgt über die Container-Transformation. Diese erfolgt nach der Beschreibung aus der Anleitung für die Nutzerschaft [Coda, T3.2: Deployment Configuration aka Container Transformation].

Als erstes wird ein Rechtsklick auf die „roboCar.muml“-Datei durchgeführt und in dem Kontext-Menü „Export“ ausgewählt. In dem Fenster mit den Export-Wizards wird in dem Ordner „MechatronicUML“ der Eintrag „Container Model and Middleware Configuration“ ausgewählt.

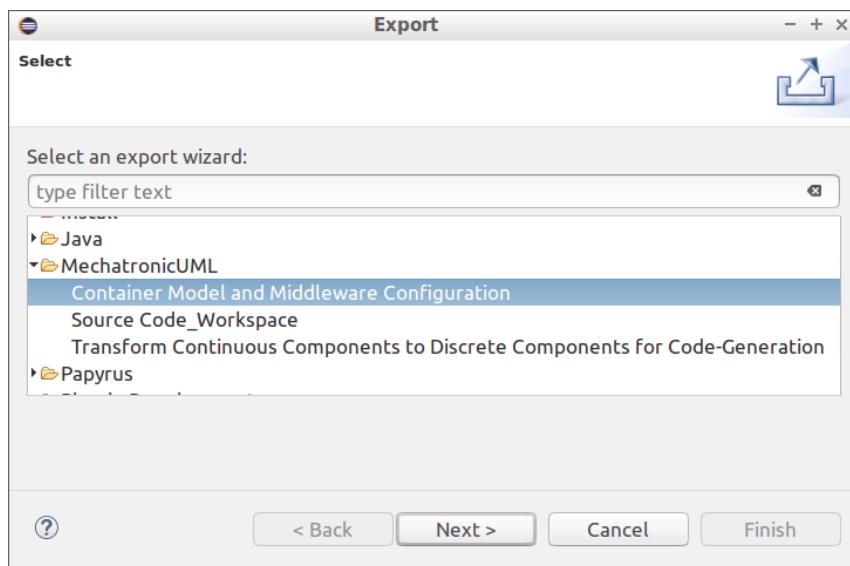


Abbildung 6.5.: Container-Transformation, Teil 1

Es sollte automatisch die Datei „roboCar.muml“ als Modell-URI eingetragen sein und unter „Select source elements“ sollte in dem Baum das Element „System Allocation“ eingetragen sein.

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

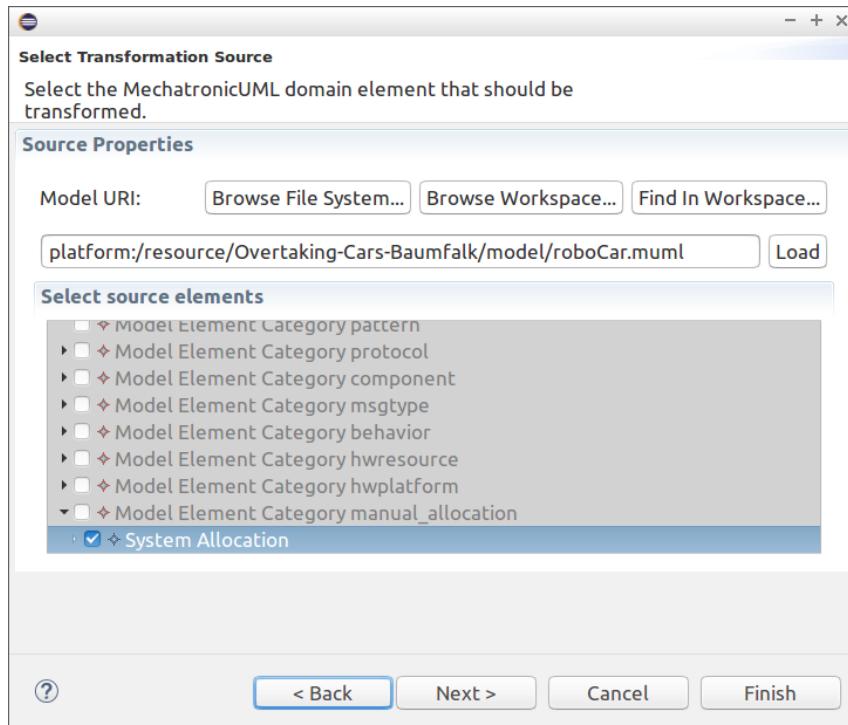


Abbildung 6.6.: Container-Transformation, Teil 2

Nach dem Klicken auf „Next >“ wechselt das Fenster in die Auswahl der Middleware-Konfigurationen. An dieser Stelle wird für das Anwendungsszenario die Option „MQTT and I2C Middleware Configuration“ ausgewählt.

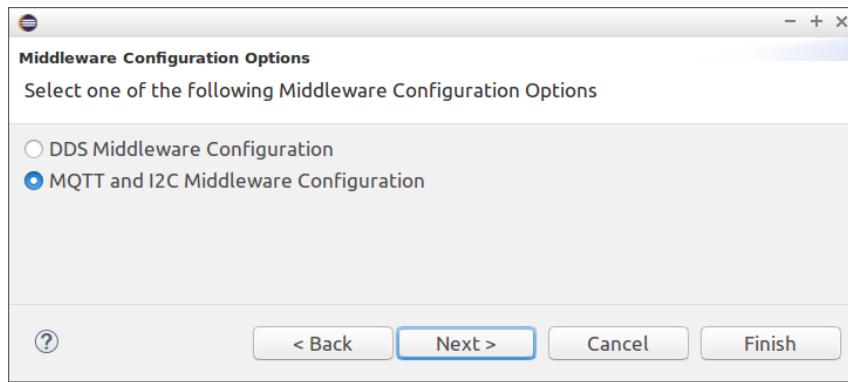


Abbildung 6.7.: Container-Transformation, Teil 3

Nach dem Klicken auf „Next >“ werden der Workspace als Dateisystem und der Zielpfad, der ausgewählt wurde, angezeigt. Per Ordnerauswahl im Dateisystem oder per „Filesystem...“ kann der Generierungsort des Transformationsergebnisses angepasst werden. Im Rahmen dieser Ausarbeitung wird für die Unterscheidung zwischen generiertem unvollständigem Code und verwendbarem Code als Zielort im Projektordner ein Ordner namens „generated-files“ angelegt und eingetragen.

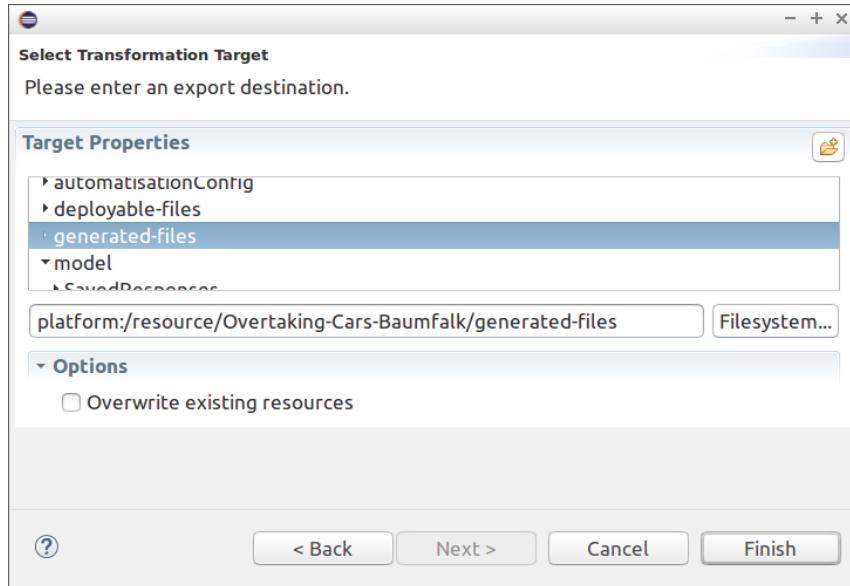


Abbildung 6.8.: Container-Transformation, Teil 4

Über den „Finish“-Knopf wird die Container-Transformation gestartet.

Als Pipelineschritt sieht der Eintrag für alle diese Handgriffe mit der GUI beispielsweise wie folgt aus:

```
ContainerTransformation:
  in:
    roboCar_mumlSourceFile: direct model/roboCar.muml
    middlewareOption: direct MQTT_I2C_CONFIG # Or DDS_CONFIG
    muml_containerFileDestination: from muml_containerFilePath
  out:
    ifSuccessful: ifSuccessful
```

In dem Eingabeparameter `roboCar_mumlSourceFile` wird der Pfad zu der .muml-Datei, die sozusagen die Kerndatei der MUML-Modelldateien ist, oder ein Variablenverweis darauf eingetragen. In diesem Fall wurde der Pfad direkt und relativ zum Projektordner eingetragen.

Der Eingabeparameter `middlewareOption` repräsentiert die Middleware der Kommunikation bzw. den Ansatz für die Kommunikation. Hier wurde die Zeichenkette für die automatische interne Auswahl von „MQTT and I2C Middleware Configuration“ direkt eingefügt. Jedoch sei hierbei angemerkt, dass bei den Anpassungen an die Version der Roboterautos, die bei dieser Ausarbeitung verwendet wird, genau genommen die Kommunikation zwischen dem Driver- und dem Koordinator-Mikrokontroller intern durch die serielle Kommunikation ersetzt wurde. Für weitere Details siehe Kapitel 7.1 „Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken“.

Der Eingabeparameter `muml_containerFileDestination` dient für das Festlegen des Zielortes der zu generierenden „MUML_Container.muml_container“-Datei. In dem Beispiel wird dieser aus der Variable „`muml_containerFilePath`“, die den Pfad „generated-files/MUML_Container.muml_container“ relativ zum Projektordner enthält, abgerufen.

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

Der Ausgabeparameter `ifSuccessful` gibt in eine Variable aus, ob der Vorgang erfolgreich durchgeführt wurde. In diesem Fall wird die standardmäßig wiederholt überschriebene Variable „`ifSuccessful`“ verwendet.

6.4.2. Post-Processing: Beispiel: WLAN-Einstellungen

Von dem Post-Processing wurde der Schritt für das Eintragen der WLAN-Daten gewählt. Bei der manuellen Durchführung mussten diese nach jeder Code-Generierung manuell in jede „`*CarCoordinatorECU.ino`“-Datei eingetragen werden.

Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

```
PostProcessingConfigureWLANSsettings:  
    in:  
        arduinoContainersPath: from deployableFilesFolderPath  
        ecuEnding: direct CarCoordinatorECU  
        nameOrSSID: from WLANNameOrSSID  
        password: from WLANPassword_MakeSureThisStaysSafe  
    out:  
        ifSuccessful: ifSuccessful
```

Der Eingabeparameter `arduinoContainersPath` steht für den Ordnerpfad, in dem die Ordner für die Sketches der verschiedenen Boards enthalten sind. In diesem Fall wird von der Variable „`deployableFilesFolderPath`“ der Pfad „`deployable-files`“ relativ zu dem Projektordner angegeben.

Im Eingabeparameter `ecuEnding` wird der Namensfilter für die Ordnernamensenden eingetragen. Im Beispiel wird direkt per „`CarCoordinatorECU`“ die Auswahl auf „`fastCarCoordinatorECU`“ und „`slowCarCoordinatorECU`“ begrenzt.

Der Eingabeparameter `nameOrSSID` dient dem Festlegen des Namens des zu nutzenden WLAN-Netzwerks. Bei der standardmäßigen Generation der Pipeline ist in der Variable „`WLANNameOrSSID`“ nur ein Dummywert eingetragen.

Für den Eingabeparameter `password` ist dasselbe für die Passworteinstellung der Fall. Die Namensgebung der Variable soll die Nutzerschaft daran erinnern, an dieser Stelle vorsichtig zu sein, weil sonst durch das Hochladen des verwendungsbereiten Sketches das Passwort an öffentliche einsehbare Stellen gelangen kann.

Wie zuvor gibt der Ausgabeparameter `ifSuccessful` in eine Variable aus, ob der Vorgang erfolgreich erfolgte. Auch hier wird die Variable „`ifSuccessful`“ wiederverwendet.

6.4.3. Verwendung von Arduino-Software: Beispiel: Kompilieren

Intern handelt es sich bei dem Verifizieren von Arduinocode durch die Arduino-IDE genau genommen um einen Kompiliervorgang, dessen Meldungen angezeigt werden. Die hierbei kompilierten Dateien werden für das Hochladen temporär gespeichert. Das Starten der Verifikation ist also das Starten eines Kompiliervorgangs, dessen Ergebnisse temporär gespeichert werden.

Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

```
Compile:  
    in:  
        boardTypeIdentifierFQBN: from usedCoordinatorBoardIdentifierFQBN  
        targetInoFile: from slowCarCoordinatorECUINOFilePath
```

```
saveCompiledFilesNearby: direct true
out:
  ifSuccessful: ifSuccessful
  resultMessage: resultMessage
```

Der Eingabeparameter `boardTypeIdentifierFQBN` dient dem Festlegen des Ziel-Boardtyps. Im Fall des Koordinator-Boards wird von der Pipeline-Generierung standardmäßig in der Variable „`usedCoordinatorBoardIdentifierFQBN`“ der Arduino-Mega eingetragen.

Der Eingabeparameter `targetInoFile` steht für den Pfad zu der zu kompilierenden `.ino`-Datei. Bei diesem Ausschnitt der Pipelinesequenz handelt es sich relativ zum Projektordner um „`deployable-files/slowCarCoordinatorECU/slowCarCoordinatorECU.ino`“.

Über den Eingabeparameter `saveCompiledFilesNearby` wird eingestellt, ob die beim Kompilervorgang generierten Dateien in einem für sie erstellten Ordner „`CompiledFiles`“ gespeichert werden sollen oder nicht. Dies ist standardmäßig direkt auf das Speichern eingestellt.

Hierbei ist zu beachten, dass am Speicherort der kompilierten Dateien auch eine Textdatei mit dem verwendeten bzw. angegebenen Boardtyp angelegt wird. Diese Datei bzw. deren darin angegebener Boardtyp wird für das Gegenvergleichen der Boardtypen beim Hochladen verwendet (siehe Abschnitt 7.2.5 „Implementierung als Plug-In“).

Der Ausgabeparameter `ifSuccessful` hat das typische Verhalten.

Über `resultMessage` wird bei einem Fehlschlag mitunter auf den Ort der Datei mit dem Report über den Kompilervorgang verwiesen.

6.4.4. Hilfsschritte: Beispiel: Löschen von Ordnern

Es müssen vor jedem vollständigen Durchlauf die alten erstellten Ordner gelöscht werden, um sicher zu gehen, dass alte Dateien nicht für Fehler oder Verwirrung sorgen. Manuell kann es über den Explorer oder über die Ordner-Ansicht der Eclipse-IDE und dem Löschen der entsprechenden Einträge erfolgen.

Der Schritt `DeleteFolder` ist für solche Bereinigungen von alten generierten Ordnern inklusive deren Inhalten konzipiert.

Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

```
DeleteFolder:
  in:
    path: from deployableFilesFolderPath
  out:
    ifSuccessful: ifSuccessful
```

Der Eingabeparameter `path` steht für den Pfad des zu löschenen Ordners. Bei diesem Ausschnitt der Pipelinesequenz handelt es sich relativ zum Projektordner um „`deployable-files/slowCarCoordinatorECU/slowCarCoordinatorECU.ino`“.

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

6.4.5. Abbruch mit Feedback bei Fehlschlag eines Schrittes mit einer Erklärung

Abbrüche mit Feedback sind für Pipelines wichtig und dies wird über OnlyContinueIfFulfilledElseAbort durchgeführt. Theoretisch hätte das System für eine selbstständige Überprüfung des Erfolgs konzipiert werden können, aber so soll mehr Flexibilität beim Umgang mit Fehlschlägen und deren Meldungen ermöglicht werden.

Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

```
OnlyContinueIfFulfilledElseAbort:  
  in:  
    condition: from ifSuccessful  
    message: direct Compile for fastCarCoordinatorECU has failed!
```

Wenn der in Eingabeparameter `condition` übergebene Wahrheitswert „true“ ist, so wird der diesen liefernde Schritt als erfolgreich angesehen und die Pipelinedurchführung fährt fort. Ansonsten wird der im Eingabeparameter `message` angegebene Text in einem Fenster angezeigt und die Pipelineausführung abgebrochen. Im Fall des Beispiels wird überprüft, ob beim Kompilieren des Arduinocodes für die Programmierung des Koordinatorboards des schnellen Roboterautos ein Fehlschlag festgestellt werden konnte und wenn ja, so wird dies leicht lesbar gemeldet.

6.4.6. Anzeigen eines Textes in einem Fenster

Für sonstige Meldungen wurde der Pipelineschritt PopupWindowMessage konzipiert. So kann per Parameter `condition` bei Bedarf bedingt ein Fenster mit der in Parameter `message` angegeben Nachricht angezeigt werden, ohne die Ausführung der Pipeline abzubrechen.

Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

```
DialogMessage:  
  in:  
    condition: direct true  
    message: direct Pipeline execution completed!
```

6.4.7. Pipelineschritte für Flexibilität und Langlebigkeit

Es wurden für mehr Flexibilität und längerer Nützlichkeit des Pipelinesystems auch ohne Anpassungen an dessen Sourcecode weitere Pipelineschritte konzipiert, die andere Aufgaben bewältigen können. Zur Übersichtlichkeit dieses Kapitels werden diese und deren Beschreibungen im Ergänzungsmaterial H „Ergänzungsmaterial Übersicht der Pipelineschritte“ aufgelistet. So ist beispielsweise der in Abschnitt 6.3.1 „Langlebige Flexibilität über mögliche Terminalaufrufe“ kurz umschriebene Pipelineschritt TerminalCommand ein sehr nützlicher Befehl für die Einbindung der automatischen Ausführung von Skripten und Programmen.

6.5. Aufbau der Konfigurationsdatei

Es wurden zunächst verschiedene Entwürfe für den Aufbau der Konfigurationsdatei „pipeline-Settings.yaml“ aufgestellt, von denen auf Grundlage von Diskussionen mit dem Betreuer der im Folgenden beschriebene Ansatz ausgewählt wurde.

Es sind in der Konfigurationsdatei vier Bereiche bzw. Mappings vorgesehen: `VariableDefs` für die anfänglichen Variablendefinitionen, `TransformationAndCodeGenerationPreconfigurations` für die Einstellungen für die einzelnen automatischen Ausführungen von den MUML-Werkzeugen, `PostProcessingSequence` für die Einstellungen für das Post-Processing und `PipelineSequence` für die eigentliche Pipeline. Für eine grafische Darstellung siehe Abbildung 6.9.

In den folgenden Unterabschnitten werden diese vier genannten Felder und ihre Konzepte der Reihe nach genauer beschrieben werden.

6.5.1. VariableDefs - Variablen

Um Redundanzen und Fehlergelegenheiten beim Eintragen von Werten zu vermeiden und um das Weitergeben von Werten zu ermöglichen, wurde das Konzept von Variablen eingeführt. Hierbei wird ein Typenkonzept wie bei vielen Programmiersprachen verwendet, um das Risiko von unpassend verwendeten Werten zu reduzieren. Hierfür stehen in dem Pipelinesystem die Typen `Number`, `String`, `Boolean`, `Path`, `FolderPath`, `FilePath`, `BoardSerialNumber`, `BoardIdentifierFQBN`, `ConnectionPort`, `WLANNName`, `WLANPassword`, `ServerIPAddress` und `ServerPort` zur Verfügung. Jeder Parameter hat einen zugeordneten Typ.

Bei Eingabeparametern wird dieser für die Korrektheitsüberprüfung verwendet, aber ein interner Spezialtyp erlaubt alle Variablentypen. So kann z.B. per `DialogMessage` auch der gefundene Port eines Boards angezeigt werden.

Bei Ausgabeparametern hat die Typenangabe zwei Aufgaben: Zum einen während des dynamischen Definierens und Initialisierens das Festlegen des Variablentyps und zum anderen während des Überschreibens das Verhindern einer Typenänderung. Dies soll das Fehlerrisiko bei der Nutzung verringern.

Variablen können sowohl direkt am Anfang als auch dynamisch zur Laufzeit definiert und initialisiert werden. Das Definieren und Initialisieren am Anfang erfolgt innerhalb von `VariableDefs` als Liste, wobei die Variableneinträge jeweils nach der folgenden Syntax eingetragen werden:

```
<Variablenname> <Variablenwert>
```

Es können auch innerhalb des Eintrags für den Variablenwert Leerzeichen vorkommen, weil nur die ersten zwei Leerzeichen zum Aufteilen verwendet werden.

Das dynamische Definieren und Initialisieren erfolgt in einem Pipelineschritt durch das Eintragen des gewählten Namens als Speicherziel von einem Ausgabeparameter (siehe auch Abschnitt 6.5.5 „Schrittdefinitionen“):

```
...
out:
<Ausgabeparametername>: <Variablenname>
```

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

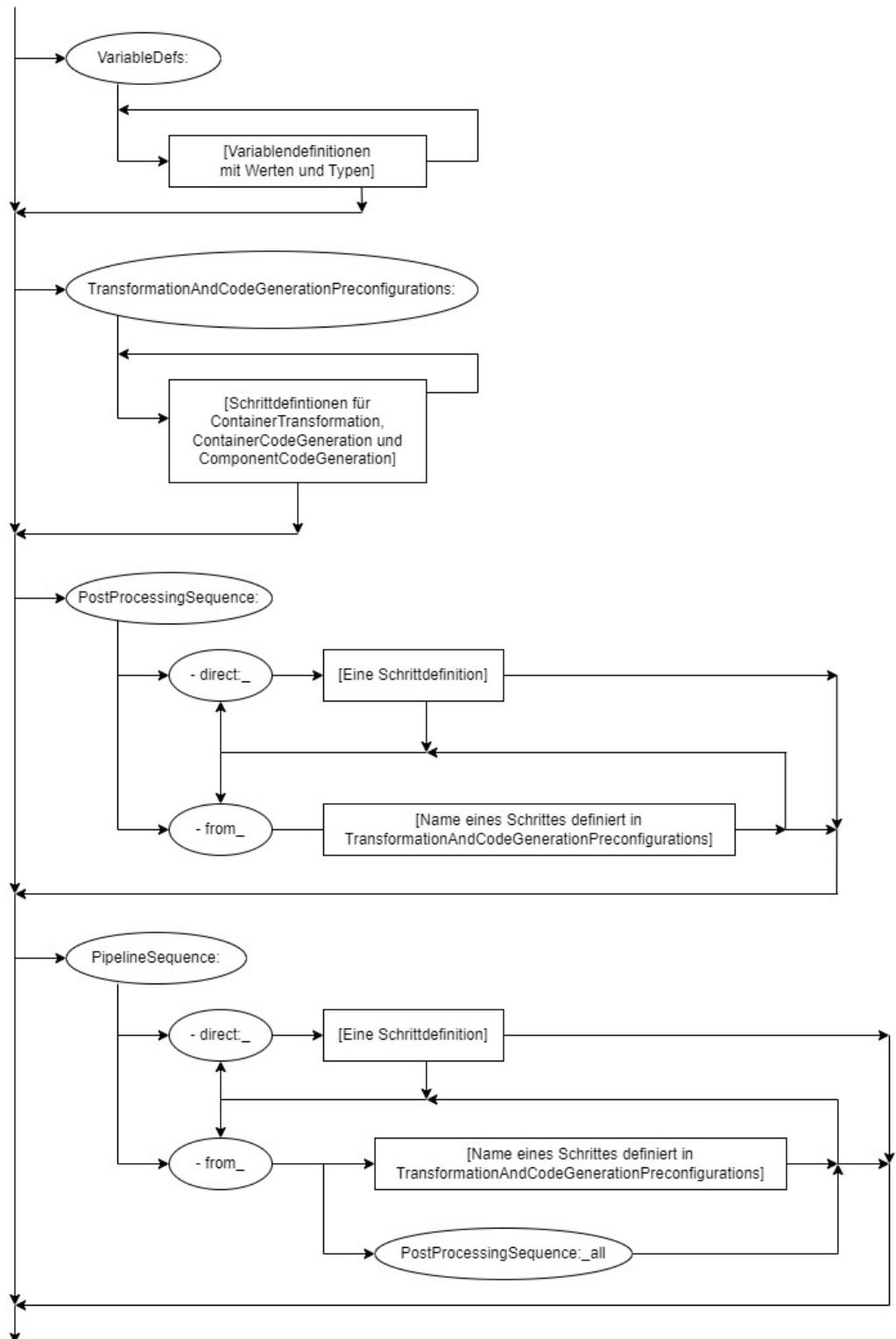


Abbildung 6.9.: Übersicht über den Aufbau der Konfigurationsdatei der Pipeline „pipelinesettings.yaml“

Hierbei erfolgt die Typenzuweisung intern und automatisch.

Zur Vereinfachung beim Schreiben der Konfigurationen wurde entschieden, dass das Überschreiben eines Variableninhalts auf demselben Weg, also über die Angabe von ihrem Namen als Speicherziel, erfolgen soll. Wie oben beschrieben wird, ist hierbei keine Typenänderung erlaubt. Die Unterscheidung zwischen dem Aufnehmen einer neuen Variable und dem Überschreiben des Inhalts einer vorhandenen Variable erfolgt intern abhängig davon, ob der angegebene Variablenname bereits intern bekannt ist.

6.5.2. TransformationAndCodeGenerationPreconfigurations - Einstellungen für die einzelne automatische Ausführung von den MUML-Werkzeugen

Unter dem Mapping-Schlüssel `TransformationAndCodeGenerationPreconfigurations` sind die Mappings zu den Einstellungen bzw. Schrittdefinitionen für die Container-Transformation, die Generation des Container-Codes und die Generation des Komponenten-Codes enthalten. Für das Vermeiden von Redundanzen können innerhalb von `PostProcessingSequence` und `PipelineSequence` Bezüge auf diese eingetragen werden, die wiederum jeweils an ihrer Stelle das Ausführen des entsprechenden Werkzeuges mit den jeweiligen Einstellungen darstellt.

6.5.3. PostProcessingSequence - Einstellungen für die automatische Ausführung von den Post-Processing-Schritten

Unter dem Mapping-Schlüssel `PostProcessingSequence` ist die Sequenz mit den Einstellungen bzw. Schrittangaben für die Pipelineschritte, die beim Post-Processing durchgeführt werden sollen, enthalten. Zur Redundanzvermeidung wurde für `PipelineSequence` das Beziehen auf die Sequenz des Post-Processings bzw. `PostProcessingSequence` konzipiert, wobei hierbei der Bezug auf diese das komplette Einfügen von ihr darstellt.

Zwecks Flexibilität wurden auch Referenzen auf die Einstellungen aus `TransformationAndCodeGenerationPreconfigurations` ermöglicht.

6.5.4. PipelineSequence - Die Sequenz für die eigentliche Pipeline

Die Pipelinesequenz besteht aus einer Reihe an direkt darin definierten Pipelineschritten, aber es sind auch Verweise auf die Einstellungen aus `TransformationAndCodeGenerationPreconfigurations` und auf die komplette Sequenz aus `PostProcessingSequence` möglich. So müssen eben diese Einstellungen nicht erneut komplett eingetragen werden, wodurch mögliche Redundanzen vermieden werden können.

6.5.5. Schrittdefinitionen

Fast jeder Pipelineschritt repräsentiert einen manuellen Arbeitsschritt, wie beispielsweise `Compile` das Compilieren per Arduino-CLI. Jeder Eintrag, der einen Pipelineschritt und seine Einstellungen deklariert, hat die folgende Syntax:

6. Entwurf der CI/CD-Pipeline für die MUML-Toolsuite

```
<Pipelineschrittname>
[in:
  <Sequenz der Eingabeparameter und jeweiligem Inhalt>]
[out:
  <Sequenz der Ausgabeparameter und jeweiligem Speicherziel bei seinem Variablenamen>]
```

Interne Standardwerte für Parameter wurden nicht konzipiert, also müssen immer exakt alle jeweiligen Eingangs- und Ausgangsparameter belegt werden. Zusätzliche oder fehlende Einträge werden nicht toleriert.

Die Unterteilung zwischen Eingabe- und Ausgabeparametern erfolgt passend benannt über die Felder `in` und `out`.

Die Eingabeparameter werden als eine Sequenz aus einzelnen Mappings eingetragen. Diese bestehen jeweils aus dem entsprechenden Parameternamen und einer Werteangabe. Die Werteangabe erfolgt entweder als direkt eingetragener Wert per Schlüsselwort `direct`, gefolgt von dem beabsichtigten Wert, oder dem Lesen von Werten aus Variablen per Schlüsselwort `from`, gefolgt von dem Namen der Variable, deren Inhalt gelesen werden soll.

Bei direkt eingetragenen Werten erfolgt jedoch keine Typenüberprüfung, weil davon ausgegangen wird, dass bei den aussagekräftig gewählten Eingabeparameternamen und innerhalb eines so kleinen Kontextes keine Typenverwechslung passiert.

Syntaktisch beschrieben sehen die Eingabeparametersequenzeinträge jeweils folgendermaßen aus:

```
<Eingabeparametername>: direct: <VariableValue>
```

oder

```
<Eingabeparametername>: from: <VariableName>
```

Die Ausgabeparameter werden ebenfalls als eine Sequenz aus einzelnen Mappings eingetragen. Diese bestehen jeweils aus dem entsprechenden Parameternamen und dem Namen einer Variablen, in der der jeweilige Ergebniswert gespeichert werden soll. Wie in Abschnitt 6.5.1 „VariableDefs - Variablen“ beschrieben wurde, wird hierbei automatisch je nach Fall eine neue Variable aufgenommen oder die Inhaltsüberschreibung einer bekannten Variable durchgeführt.

Grafisch dargestellt sieht das folgendermaßen aus:

6.5. Aufbau der Konfigurationsdatei

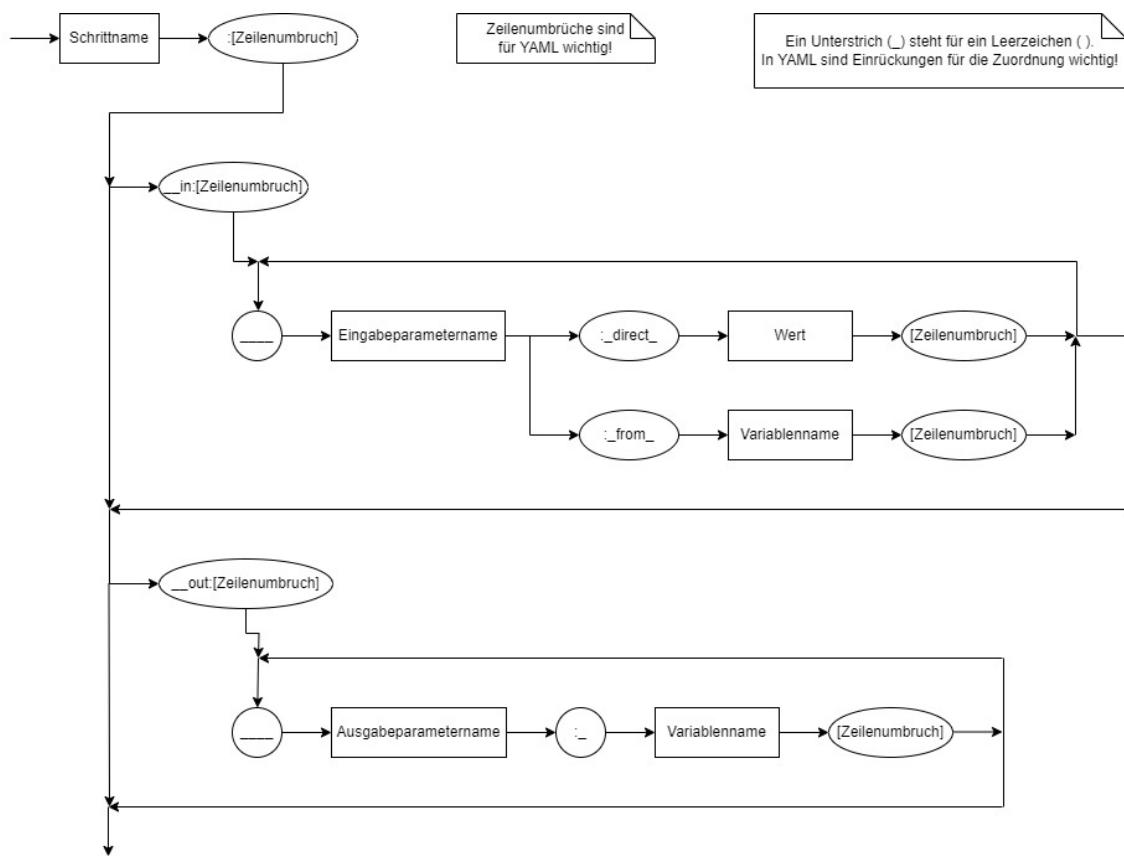


Abbildung 6.10.: Aufbau des Auftrages eines Pipelineschrittes

7. Umsetzungen

Dieses Kapitel erklärt, wie die verschiedenen Entwicklungsaufgaben umgesetzt wurden. In Kapitel 7.1 „Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken“ wird beschrieben, wie die MUML-Modelle und der Post-Processing-Ablauf angepasst wurden. Kapitel 7.2 „Integration der ArduinoCLI“ erklärt das entwickelte Eclipse-Plug-In „ArduinoCLUtilizer“. In Kapitel 7.3 „Vollständige Automatisierung per Pipeline“ werden der Entwicklungsprozess der CI/CD-Pipeline sowie die aufgetretenen Hürden und deren Workarounds dargelegt.

7.1. Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken

Dieses Kapitel beschreibt die Änderungen an den MUML-Modellen, den internen Ressourcen und dem Post-Processing-Ablauf sowie die Analysen und Abwägungen, die jeweils zuvor durchgeführt wurden.

Im Rahmen dieses Kapitels werden also die folgenden Forschungsziele beantwortet:

FZ1.3 Welche Kriterien sind für die Auswertung von Ansätzen relevant?

FZ2.1 Werden die entsprechend betroffenen Artefakte standardmäßig integriert? Und falls ja, wie?

FZ2.2 Welche validen Anpassungsansätze gibt es?

FZ2.3 Welcher Anpassungsansatz erfüllt die Kriterien am besten und wird umgesetzt werden?

Die FZ 1.3, 2.1, 2.2 und 2.3 sind jeweils nicht nur für ein Thema oder einen Teil relevant, sondern durch die verschiedenen Änderungsstellen für mehrere Themen. Deshalb treten in diesem Kapitel auf diese vier FZ jeweils mehrere Verweise und Antworten auf.

Die notwendigen Anpassungen an den aktuellen Stand der Roboterautos werden durch die folgenden Anforderungen spezifiziert:

ANF 7.1 Anpassung der MUML-Modelle an die von dieser Ausarbeitung verwendete Version der Roboterautos

ANF 7.2 Bessere Integration der Bibliothek SofdCar-HAL

ANF 7.3 Serielle Kommunikation statt I2C-Kommunikation

ANF 7.4 Anpassung der Post-Processing-Schritte

7. Umsetzungen

Die Analysen und vorgenommenen Änderungen sind auf die folgenden Abschnitte aufgeteilt:

In 7.1.1 „Anpassungen an den MUML-Modellen“ werden die Anpassungen an den MUML-Modellen beschrieben.

In 7.1.2 „Änderung des Kommunikationsprotokolls zwischen den Boards innerhalb von einem Roboterauto von I2C auf Seriell“ wird die Änderung der Kommunikationsressourcen von I2C auf Seriell erklärt.

In 7.1.3 „Anpassung der Post-Processing-Schritte“ werden die Analyse des Post-Processings und seine Anpassung beschrieben.

7.1.1. Anpassungen an den MUML-Modellen

Dieser Abschnitt beschreibt zusammengefasst die Änderungen, die an den verschiedenen MUML-Modellen vorgenommen wurden, um Kompatibilität zu den unter dieser Ausarbeitung verwendeten Roboterautos herzustellen. Für die genauen Details siehe Ergänzungsmaterial C „Ergänzungsmaterial Änderungen an MUML-Modellen“.

Hier ist die Antwort auf FZ2.1, dass die Modelle der verschiedenen Komponenten und deren Instanzen angepasst sowie die Ports und Verbindungen zwischen deren (der Komponenten und Instanzen) Elementen in den verschiedenen Diagrammen teils geändert und teils hinzugefügt werden.

7.1.1.1 Mögliche Ansätze

Bei den Analysen, wie die Anpassungen durchgeführt werden können, wurden hinsichtlich der Software-Komponenten zwei Ansätze gefunden:

1. Der erste Ansatz beruht auf dem Beibehalten der alten Architektur und dem Ersetzen der alten Komponenten mit entsprechenden neuen Komponenten aus Sofdcar-HAL. Architekturänderungen wären einzig für das Richtungslenken erfolgt. Die Komponente `DriveControl` aus Stürners Ausarbeitung[Stü22] hätte also seitens des Modells direkten Zugriff auf konkrete Kindklassen der abstrakten Klassen `SteerableAxe` und `Motor` aus Sofdcar-HAL erhalten. Diese repräsentieren das Einstellen der Fahrgeschwindigkeit und die Lenkachse und sollen eigentlich von einer `DriveController`-Klasse verwendet werden. Das Konzept der Modularisierung würde also durchbrochen werden.
2. Das Priorisieren der Modularisierung, d.h. die Komponente `PowerTrain`, die das Einstellen der Geschwindigkeit repräsentiert, würde mit einer Kindklasse der abstrakten Klasse `DriveController` ersetzt werden. So würde die Modularisierung beibehalten werden und alles aus dieser Bibliothek könnte so arbeiten oder verwendet werden, wie es jeweils vorgesehen ist. Um Verwechslungen vorzubeugen, würde `DriveControl` zu `CourseControl` umbenannt werden. Dies würde auch zu einer passenderen Namensgebung führen, weil dann diese Kontrollkomponente vielmehr den Kurs auf den Fahrbahnen festlegt, aber nicht für Lenken beim Folgen der zum entsprechenden Zeitpunkt genutzten Fahrbahn verantwortlich ist. Diese beiden Möglichkeiten sind die Antwort auf FZ2.2 für dieses Thema.

Es konnten keine Bewertungskriterien für Modelländerungen gefunden werden, also werden Kriterien für die Code-Qualität verwendet. Dies beantwortet für dieses Thema FZ1.3 . In der Codequality ist Modularisierung ein Qualitätsaspekt (siehe Abschnitt 3.4 „Codequality“), also wurde beschlossen, Ansatz 2 umzusetzen.

7.1.1.2 Umsetzung von Ansatz 2

Im Folgenden werden die hierfür durchgeführten Änderungen zusammengefasst beschrieben. Hierbei ist zu beachten, dass beim Durchführen von Änderungen in der korrekten Stelle diese an andere Dateien weiter propagiert werden und man deshalb für manche Vorgänge nicht an allen Stellen die Änderungen vornehmen muss. So wird z.B. unter „Model/component/“ in „roboCar.component_diagram“ die statische strukturierte Komponente RoboCar um einen Komponententeil namens steeringAngle erweitert und MUML propagiert dies u.a. nach „Model/instance/roboCar2.componentinstanceconfiguration_diagram“ weiter.

1. Umbenennungen von Komponenten: In verschiedenen Diagrammen wurden die Komponenten PowerTrain und DriveControl entsprechend zu DriveController und CourseControl umbenannt.
2. Anpassung der Datentypen an den Ports für Geschwindigkeitswerte: In der Bibliothek „Sofdcar- HAL“ ist in den Funktionen der abstrakten Klasse DriveController die Geschwindigkeit als Wert des Datentyps int16 eingetragen. Folglich wurden bei den ehemaligen PowerTrain und DriveControl bzw. nun DriveController und CourseControl die Ports für die Geschwindigkeit auf Primitive Data Type int16 geändert.
3. Ports für die Winkel-Werte: Bei DriveController und CourseControl wurden die Ports für die Übertragung der Winkelwerte hinzugefügt und miteinander verbunden. Der Datentyp Primitive Data Type int8 wurde gewählt, weil in der Bibliothek „Sofdcar-HAL“ in den Funktionen der abstrakten Klasse DriveController der Winkel als Wert des Datentyps int16 eingetragen ist.
4. Hinzufügen des Lenk-Servos: Für den Lenkservo wurden in verschiedenen Diagrammen die entsprechenden Elemente hinzugefügt. Innerhalb der Plattform-Diagramme ist dieser Servo als steeringServo modelliert. Für die Kommunikation mit dem Servo wurde das Protokoll „Link Protocol ServoControl“ definiert und die verschiedenen Ports erstellt. In den Diagrammelementen für die Fahrer-Boards bzw. -ECUs ist sein Port als „HW Port SteeringAxe“ definiert. Es wurden auch die entsprechenden Verbindungen vorgenommen.
5. Entfernen überschüssiger Motoren: Die Modelle arbeiteten mit zwei virtuellen Elektromotoren. Für das Modellieren wurde der linke Motor leftMotor zu fixedMotor (fixierter Motor) umbenannt und der rechte Motor rightMotor entfernt.
6. Anpassung von „roboCarLibraries.osdsl“: Für das Anpassen der zu verwendenden APIs wurde MotorDriver durch DriveController ersetzt und in dessen Funktionsdefinitionen bzw. angegebenen Schnittstellen wurde in setSpeed der Parametertyp auf int16 angepasst und die Funktionsdefinition void setAngle(int8 angle); hinzugefügt. Dies hat den folgenden Hintergrund: Aus der Bibliothek „MotorDriver“ wurde die Klasse MotorDriver von Stürner [Stü22] für das Ansteuern der Motoren verwendet. Diese Funktionalität wird in der „Sofdcar- HAL“-Bibliothek indirekt von der abstrakten Klasse DriveController durchgeführt, aber die Nutzung ist ähnlich, so dass die beschriebenen Änderungen ausreichen.

7. Umsetzungen

- 7. Anpassung der Allokationen:** Hier wurden in den Coallokationsdefinitionen `PowerTrain` und `DriveControl` entsprechend mit `DriveController` und `CourseControl` ersetzt.

Somit wurde der ANF 6.1 „Anpassung der MUML-Modelle an die von dieser Ausarbeitung verwendete Version der Roboterautos“ erfüllt. Zusätzlich wurde dabei die Integration der Bibliothek „SofdCar-HAL“ verbessert, wodurch ANF 6.2 „Bessere Integration der Bibliothek SofdCar-HAL“ ebenfalls erfüllt wird.

7.1.2. Änderung des Kommunikationsprotokolls zwischen den Boards innerhalb von einem Roboterauto von I2C auf Seriell

Um die Zuverlässigkeit der Kommunikation beim Nachrichtenaustausch innerhalb desselben Roboterautos zwischen dem Koordinator-AMK und dem Fahrer-AMK zu verbessern, hat Georg Reißner vorgeschlagen, das Kommunikationsprotokoll von I2C auf Seriell abzuändern. Hierbei weist die von ihm geschriebene Bibliothek die selben Schnittstellen wie Stürners Implementierung [Stü22] auf. Durchgeführte Tests haben gezeigt, dass diese Verbesserung der Kommunikation zwischen den AMKs eines Roboterautos korrekt arbeitet.

Nach einer erfolglosen Suche, wie die serielle Kommunikation in MUML oder seinen Erweiterungen verwaltet wird, wurde beschlossen, innerhalb der MUML-Plug-In-Projekte die hierfür verwendeten Dateien zu ersetzen und alle Beschreibungen und Bezüge darauf anzupassen. Das Hinzufügen der seriellen Kommunikation als Typ ist wahrscheinlich prinzipiell möglich, aber die zuvor genannte Suchergebnis ist ein starkes Indiz für erforderliche komplexe Erweiterungen, für die die Kenntnisse fehlten. Dies beantwortet hierfür FZ 2.1 und 2.2. Die anderen FZ1.3 und 2.3 entfallen mangels möglicher bekannter oder konzipierbarer Möglichkeiten.

Im Laufe dieses Abschnittes wurde also die Erfüllung von ANF6.3 „Serielle Kommunikation statt I2C-Kommunikation“ behandelt.

7.1.2.1 Einfügen der Code-Dateien für die serielle Kommunikation

Die Generation des Komponenten-Codes bezieht die Dateien „I2cCustomLib.cpp“ und „I2cCustomLib.hpp“ aus internen Ressourcen. In dem Sourcecode bzw. in den Eclipse-Plug-In-Projekten waren die beiden I2cCustomLib-Dateien, in dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“ unter dem Package „org.mumlarduino.adapter.container“ unter dem Ordnerverzeichnis „resources/container_lib“ abgelegt. Dort wurden sie mit Reißners Code, also „SerialCustomLib.cpp“ und „SerialCustomLib.hpp“ ersetzt.

7.1.2.2 Anpassen der Bezüge

In demselben Package unter „/bin/org/muml/arduino/adapter/container/main“ wurden in den Dateien „MainFile.mtl“ und „MainFile.emtl“ die Bezüge von „I2cCustomLib.hpp“ auf „SerialCustomLib.hpp“ geändert.

In demselben Ordnerverzeichnis wurde in den Dateien „main.mtl“ und „main.emtl“ sowie in „src/org/muml/arduino/adapter/container/main/main.mtl“ der Kommentartext „For I2C communication, the Arduino is expected to be using its I2C pins. See more information in the I2cCustomLib.hpp in

7.1. Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken

the respective ECU's directory.“ nach „For the serial communication, the Arduino is expected to be using its I2C pins. See more information in the SerialCustomLib.hpp in the respective ECU's directory.“ umgeschrieben.

7.1.3. Anpassung der Post-Processing-Schritte

Das Post-Processing nach Reißner[Reib] besteht aus einer Vielzahl an manuell auszuführenden Schritten. Deshalb werden in diesem Abschnitt die Änderungen nur zusammengefasst bzw. die groben Details beschrieben. Bei einem Vorher-Nachher-Vergleich der generierten unvollständigen Codeteile vor und nach den Änderungen an den Modellen und Ressourcen stellte sich heraus, dass die Änderungen an den MUML-Modellen zu anders benannten und zusätzlichen Dateien sowie zu leicht anderen Dateiinhalten führten. Nicht alle von diesen erfordern Änderungen an dem Post-Processing-Ablauf, weil über die Code-Generatoren beispielsweise die Referenzen auf verschiedene Objekte bereits angepasst wurden. Die genauen Details können in Abschnitt D.1 „Änderungen am Post-Processing-Ablauf“ nachgelesen werden.

In den folgenden Abschnitten werden zu den Änderungen die jeweiligen Anpassungen beschrieben. Diese stellen auch die Erfüllung von ANF 6.4 „Anpassung der Post-Processing-Schritte“ dar.

Die erwähnte manuelle Ausführung beantwortet FZ 2.1. Die notwendigen Änderungen sind in den Hauptaspekten eindeutig und die Anpassungslösungen unterscheiden sich lediglich in der Reihenfolge der Ausführung, was FZ 2.2 beantwortet. Somit entfällt FZ 1.3.

FZ 2.3 wird in den folgenden Abschnitten kompakt beantwortet und in dem oben erwähnten Abschnitt D.1 „Änderungen am Post-Processing-Ablauf“ ausführlich.

7.1.3.1 Berücksichtigung anders benannter Dateien

Wegen der in Abschnitt „7.1.1.2 Umsetzung von Ansatz 2“ beschrieben Namensänderung kommt es in den generierten unvollständigen Codeteilen zu anderen Namen, was sich in Form von anders benannten Dateien und angepassten Referenzen auswirkt.

So wurde beispielsweise aus DriveControl in allen Dateinamen und Bezügen CourseControl, was wiederum beim Post-Processing-Ablauf berücksichtigt werden muss. In der Tabelle 7.1 werden diese Umbenennungen gegenübergestellt.

Alt	Aktuell
driveControl	courseControl
PowerTrain	DriveController
I2CCustomLib	SerialCustomLib

Tabelle 7.1.: Namensänderungen

7.1.3.2 Weitere auszufüllende und nachzubearbeitende API-Dateien

Das Hinzufügen von weiteren Ports, beispielsweise für den Winkel bei der Anpassung von PowerTrain, verursacht weitere Dateien und zu implementierende Befehle. So gibt es in den API-Mappings (Ordner „APIMappings“) die neu hinzugekommenen Datei-

7. Umsetzungen

en „CI_DRIVECONTROLLERFDRIVECONTROLLERanglePortaccessCommand.c“, „CI_DRIVECONTROLLERFDRIVECONTROLLERanglePortaccessCommand.h“, „CI_DRIVECONTROLLERSDRIVECONTROLLERanglePortaccessCommand.c“ und „CI_DRIVECONTROLLERSDRIVECONTROLLERanglePortaccessCommand.h“, in denen der Befehl `*angle = SimpleHardwareController_DriveController_GetAngle();` eingetragen werden muss.

7.2. Integration der ArduinoCLI

Dieses Kapitel führt die Integration der ArduinoCLI in die Eclipse-IDE als ein Eclipse-Plug-In-Projekt ein. Hiermit soll die Nutzerschaft beim Arbeiten mit MUML und dem generierten Code nur noch selten von der Eclipse-IDE in das Dateisystem und in die Arduino-IDE wechseln müssen, indem die erforderlichen manuell durchgeführten Arbeitsschritte mittels Aufrufen von der Eclipse-IDE aus durchgeführt werden können. Zusätzlich soll es für die später beschriebene Umsetzung der Pipeline (siehe Kapitel 7.3 „Vollständige Automatisierung per Pipeline“) seine Funktionalität für andere Eclipse-Plug-In-Projekte bereitstellen. Im Rahmen dieses Kapitels werden also die folgenden Forschungsziele beantwortet:

FZ3.1 Welche der manuellen Schritte mit der Arduino-IDE oder deren Teilen entsprechen welchen Befehlen der Arduino-CLI?

FZ3.2 Welche Aufrufe müssen bei der Arduino-CLI zusätzlich gemacht werden?

FZ3.3 Können durch die Arduino-CLI dieselben Ergebnisse wie mit der Arduino-IDE erzielt werden?

FZ3.4 Wie kann Sofdcar-HAL per Arduino-CLI installiert werden?

Die hierfür fehlenden Fähigkeiten der Eclipse-IDE werden durch die folgenden Anforderungen spezifiziert:

ANF7.2.1 Alle Schritte, die bisher über das Öffnen der .ino-Dateien mit der Arduino-IDE und dem Verwenden von dieser (die Arduino-IDE) erfolgten (siehe FZ3.1), müssen über die GUI der Eclipse-IDE durchgeführt oder ausgelöst werden können.

ANF7.2.2 Das resultierende Eclipse-Plug-In muss mögliche zusätzliche erforderliche Aufrufe durchführen können.

ANF7.2.3 Das Verhalten von Boards, auf die durch das resultierende Eclipse-Plug-In ein Sketch installiert wurde, muss identisch sein zu dem Verhalten von Boards, auf die über die Arduino-IDE dasselbe Sketch installiert wurde.

ANF7.2.4 Es müssen Schnittstellen vorhanden sein, damit andere Eclipse-Plug-Ins auf diese Funktionen zugreifen können.

Um die Umsetzung zu beschreiben ist dieses Kapitel in die folgenden Abschnitte unterteilt:

In 7.2.1 „Manuelle Schritte mit der Arduino-IDE und entsprechende Schritte mit der Arduino-CLI“ werden die manuellen mit der Arduino-IDE durchgeführten Schritte den Befehlen der Arduino-CLI zugeordnet und mögliche zusätzliche Schritte erörtert.

In 7.2.2 „Test der Arduino-CLI“ wird der Test der Arduino-CLI beschrieben werden.

In 7.2.3 „Installation erforderlicher Bibliotheken“ wird die Installation der Bibliotheken „Servo“, „WiFiEsp“ und „PubSubClient“ mit der Arduino-CLI beschrieben werden.

In 7.2.4 „Installation der Bibliothek Sofdcar-HAL mit der Arduino-CLI“ wird die Installation der Bibliothek „Sofdcar-HAL“ mit der Arduino-CLI beschrieben werden.

In 7.2.5 „Implementierung als Plug-In“ wird die Implementierung beschrieben.

In 7.2.6 „Behandlung von Boards, die ihren eigenen Boardtyp bzw. eigene FQBN nicht angeben“ wird das Behandeln von Boards , die ihre eigenen Typinformationen nicht angeben, erklärt.

In 7.2.7 „Hinweis auf die verwendete Version der Arduino-CLI“ wird die verwendete Version der Arduino-CLI beschrieben.

7.2.1. Manuelle Schritte mit der Arduino-IDE und entsprechende Schritte mit der Arduino-CLI

Zunächst wurde untersucht, welche manuellen Schritte auf der grafischen Oberfläche der Arduino-IDE durchgeführt werden. Die in Tabellen 7.2 und 7.3 aufgestellte Liste an manuellen Schritten und wie diese durchgeführt werden, wird hier jeweils mit dem entsprechenden Befehl oder den entsprechenden Befehlen der Arduino-CLI gegenübergestellt.

7. Umsetzungen

Schritt	Arduino-IDE	Arduino-CLI
Auflisten angegeschlossener Boards	Werkzeuge/ Port/ [die Liste ablesen] oder [in der horizontalen Leiste direkt über den Dokumenten die Auswahlliste ablesen]	arduino-cli board list
Auswählen eines Ports [PORT]	Werkzeuge/ Port/ [gewünschte Auswahl auswählen] oder [in der horizontalen Leiste direkt über den Dokumenten die gewünschte Auswahl treffen]	(Beim Hochladen durch --port [PORT])
Boardauswahl	Werkzeuge/ Board: [Evtl. steht hier ein anderes]/ [entsprechende Auswahl auswählen] oder [in der horizontalen Leiste direkt über den Dokumenten die gewünschte Auswahl treffen]	(Beim Kompilieren und beim Hochladen)
Kompilieren/ Überprüfen der .ino-Datei [INOFILE]	Sketch/ Überprüfen oder [in der horizontalen Leiste direkt über den Dokumenten den Haken anklicken]	arduino-cli compile [INOFILE]
Hochladen der .ino-Datei bzw. des Sketches [INOFILE]	Sketch/ Hochladen oder [in der horizontalen Leiste direkt über den Dokumenten den Pfeil anklicken]	arduino-cli upload --port [PORT] --fqbn [BOARD] [INOFILE] oder arduino-cli upload --port [PORT] --fqbn [BOARD] --input -dir [DIR]

Tabelle 7.2.: Gegenüberstellung der manuellen Schritte und der Kommandozeilenbefehle, Teil 1

Schritt	Arduino-IDE	Arduino-CLI
Bibliotheksinstallation (Automatisch)	Werkzeuge/ „Bibliotheken verwalten“/ [In der Liste bei der gewünschten Bibliothek den Installieren-Knopf klicken] oder Seitliche Leiste/ Bibliotheksverwalter/ [In der Liste bei der gewünschten Bibliothek den Installieren-Knopf klicken]	Empfohlene Vorbereitung: <code>arduino-cli lib update-index</code> Befehl selbst: <code>arduino-cli lib install LIBRARY[@VERSION_NUMBER]... [flags]</code>
Bibliotheksinstallation (.zip-Datei)	Sketch/ Bibliothek einbinden/ .ZIP-Bibliothek hinzufügen... [Bibliotheksdatei heraussuchen] oder Inhalt der .zip-Datei bzw. entpackten Bibliotheksordner nach Dokumente/Arduino/libraries verschieben	Aufruf 1: <code>arduino-cli config set library. enable_unsafe_install true</code> Aufruf 2: <code>arduino-cli lib install --zip-path [LIBRARYZIPFILE]</code> Aufruf 3: <code>arduino-cli config set library. enable_unsafe_install false oder Inhalt der .zip-Datei bzw. entpackten Bibliotheksordner nach „/home/Arduino15/libraries“ verschieben</code>
Update von Bibliotheken	Werkzeuge/ Bibliotheken verwalten/ [In der Liste bei der updatebaren Bibliothek den Update-Knopf klicken]	<code>arduino-cli lib upgrade [Bibliothek]</code>

Tabelle 7.3.: Gegenüberstellung der manuellen Schritte und der Kommandozeilenbefehle, Teil 2

Hier folgt für weitere Aktionen, die keinem der manuellen Arbeitsschritte entsprechen, aber in der Umsetzung verwendet werden, eine weitere Tabelle:

7. Umsetzungen

Schritt	Arduino-IDE	Arduino-CLI
Update der Kern-Daten	[Seitliche Leiste]/ Board-Verwaltung/ [In der Liste bei den updatebaren Daten den Update-Knopf klicken]	Empfohlene Vorbereitung: arduino-cli core update-index Befehl selbst: arduino-cli core update
Installieren von Kernen (Zu arduino:xyz)	[Seitliche Leiste]/ Board-Verwaltung/ [In der Liste bei den updatebaren Daten den Installieren-Knopf klicken]	arduino-cli core install [arduino:xyz]
Kompilieren der .ino-Datei [INOFILE] und Speichern im Ordner [OUT]	[Nicht manuell möglich]	arduino-cli compile --fqbn [BOARD] [INOFILE] --output-dir [OUT]
Hochladen von bereits kompiliertem Code bzw. einer .hex-Datei [INOHEXFILE]	[Nicht manuell möglich]	arduino-cli upload --port [PORT] --fqbn [BOARD] --input-file [INOHEXFILE]

Tabelle 7.4.: Gegenüberstellung weiterer manueller Schritte und Kommandozeilenbefehle

Diese drei Tabellen 7.2, 7.3 und 7.4 stellen die Antwort auf die Forschungsfrage FZ3.1: „Welche der manuellen Schritte mit der Arduino-IDE oder deren Teilen entsprechen welchen Befehlen der Arduino-CLI?“ dar.

Je nach Form der Installation kann der Pfad zu dem Installationsort der Arduino-CLI nicht automatisch in den Systemdateien eingetragen worden sein. Wenn kein manuelles erfolgreiches Nachtragen durchgeführt wurde, muss als Vorbereitung für die Nutzung in der Konsole der Pfad temporär eingestellt werden. Dies erfolgt durch den Befehl `export PATH=[Pfad zu arduinocli-Datei]:$PATH`. Sein Effekt beschränkt sich nur auf die Terminalinstanz, in der er durchgeführt wurde, und solange diese aktiv ist. Neu geöffnete Terminalinstanzen werden also nicht beeinflusst.

Bei dem Installieren von Bibliotheken ist der Befehl `arduino-cli lib update-index` empfohlen, damit die Daten für das Herunterladen von den verschiedenen Adressen aktuell sind.

Dies beantwortet die Forschungsfrage FZ3.2 „Welche Aufrufe müssen bei der Arduino-CLI zusätzlich gemacht werden?“

7.2.2. Test der Arduino-CLI

Um sicher zu gehen, dass die Arduino-CLI wirklich die gleichen Ergebnisse wie die Arduino-IDE liefert, wurden zu dem Beispiel „Blink“ Variationen mit unterschiedlichen Phasenlängen jeweils mit der Arduino-IDE und mit der Arduino-CLI auf einem Arduino-UNO-Board installiert und es traten keine Unterschiede auf. Andere Testprogramme wiesen auch keine erkennbaren Unterschiede auf. Spätere Tests mit den Arduinocodes und Arduinoboards für das Anwendungsszenario ergaben auch keine Unterschiede.

Somit kann FZ3.3 mit „Ja“ beantwortet werden.

7.2.3. Installation erforderlicher Bibliotheken

Für das Kompilieren des Codes der Roboterautos werden die Bibliotheken „Servo“, „WiFiEsp“ und „PubSubClient“ benötigt [Stü22]. Diese werden über die folgenden Befehle installiert:

Listing 7.1 Arduino-CLI-Befehle für die Installation der benötigten Arduino-Bibliotheken „Servo“, „WiFiEsp“ und „PubSubClient“.

```
arduino-cli lib install Servo  
arduino-cli lib install WiFiEsp  
arduino-cli lib install PubSubClient
```

7.2.4. Installation der Bibliothek Sofdcar-HAL mit der Arduino-CLI

Das Installieren der Bibliothek Sofdcar-HAL aus dem Archiv „Sofdcar-HAL.zip“ erfolgt über eine Reihe an Anweisungen:

```
arduino-cli config set library.enable_unsafe_install true  
arduino-cli lib install --zip-path [LIBRARYZIPFILE]  
arduino-cli config set library.enable_unsafe_install false
```

Hierbei sind die Aufrufe `arduino-cli config set library.enable_unsafe_install true` und `arduino-cli config set library.enable_unsafe_install false` notwendig, um nur für den Zeitraum der Installation zeitweilig die Installation von Bibliotheken in .zip-Archiven zu erlauben und dann wieder zu verbieten.

Die Anweisung `arduino-cli lib install --zip-path [LIBRARYZIPFILE]` mit dem an der Stelle [LIBRARYZIPFILE] eingetragenen Pfad zu dem Archiv der heruntergeladenen Sofdcar-HAL-Bibliothek führt die Installation selbst aus.

Dies beantwortet FZ3.4: „Wie kann Sofdcar-HAL per Arduino-CLI installiert werden?“.

7.2.5. Implementierung als Plug-In

In diesem Abschnitt wird die Implementierung der oben genannten Aspekte als das Eclipse-Plug-In-Projekt „ArduinoCLUtilizer“ beschrieben und es wird hierbei auf die wichtigsten der hierfür getroffenen Entscheidungen eingegangen werden. Für weitere Informationen und Diagramme siehe Ergänzungsmaterial E „Ergänzungsmaterial für Eclipse-Plug-In-Projekt ,ArduinoCLUtilizer““. Um ANF7.2.4 zu erfüllen, wurde beschlossen, dass die Klassen, die für die funktionalen Aspekte dieses Eclipse-Plug-In-Projekts verantwortlich sind, nur eine geringe Kopplung zu den Klassen und Teilen von Eclipse, die an der Interaktion mit dem Nutzer/ der Nutzerin beteiligt sind, aufweisen sollen. Zusätzlich soll das theoretische Loslösen aus der Verwendung mit Eclipse leicht und schnell umsetzbar erfolgen können. Abbildung 7.1 stellt diesen Aufbau grob dar.

7. Umsetzungen

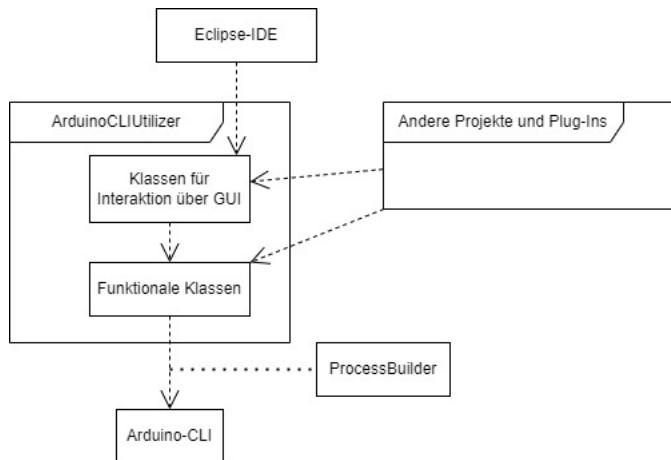


Abbildung 7.1.: Konzeptgrafik des Aufbaus des Eclipse-Plug-Ins „ArduinoCLIUtilizer“. Die Nutzungsmöglichkeiten werden auch grob dargestellt.

Deshalb wurden Klassen, die für die Interaktionen mit dem Nutzer/ der Nutzerin verwendet werden, in das Package „de.ust.arduinocliutilizer.popup.actions“ eingeplant.

Es wird eine Konfigurationsdatei namens „arduinoCLIUtilizerKonfig.yaml“ in dem Ordner „automatisationConfig“ in dem Projektordner des Eclipseprojekts, in dem gearbeitet wird, verwendet. In dieser ist unter der Einstellung arduinoCLIPathSetInPathEnvironment eingetragen, ob der Installationspfad zu der Arduino-CLI in den Systemdateien vorhanden ist, und unter der Einstellung arduinoCLIDirectory der volle Pfad des Ordners, in dem die Arduino-CLI installiert ist. So kann sich die Arduino-CLI, oder genauer gesagt, die Klasse ArduinoCLICommandLineHandler, auf für das System möglicherweise fehlende Pfaddaten zu der Arduino-CLI anpassen. Weitere Informationen zu dem Leseablauf folgen unten in diesem Abschnitt.

Die Erstellung der Konfigurationsdatei wird per Aufruf in dem Kontextmenü ([Rechtsklick auf eine .zip-, .ino- or .hex-Datei]/„ArduinoCLIUtilizer“/„GenerateArduinoCLIUtilizer config file“) gestartet. Die Klasse ArduinoCLIUtilizerConfigGenerator ist u.a. für die Erstellung des Inhalts selbst verantwortlich. Sie prüft dabei den Installationszustand der Arduino-CLI und passt den Wert der Einstellung arduinoCLIPathSetInPathEnvironment entsprechend an.

Für die Interaktion zwischen dem Javacode für das Eclipse-Plug-in und der Arduino-CLI wird java.lang.ProcessBuilder verwendet. Die darin enthaltene Klasse ProcessBuilder ermöglicht von Java-Programmen aus das Durchführen von Kommandozeilenbefehlen in Form von Prozessen. Für diese Ausarbeitung werden von diesen wiederum der Beendigungswert bzw. Exitcode, die aufgezeichneten normalen Ausgaben sowie die aufgezeichneten Fehlerausgaben verwendet und zusammen mit dem jeweiligen verwendeten Befehl in der Klasse CallAndResponses gespeichert. Für das leichtere Finden von Fehlern erstellt jede Klasse, die einen Arbeitsschritt mit der Arduinosoftware durchführt, automatisch in demselben Verzeichnis wie die verwendete Datei (z.B. Sofdcar-HAL.zip) einen Ordner namens „SavedResponses“ und speichert in diesem eine einfache Textdatei mit den folgenden Ausführungsinformationen:

Command:
[Konsolenbefehl der entsprechenden Klasse]
Results:
Exit code: [Beendigungswert bzw. Exitcode]

Normal response stream:
 [Aufgezeichnete normale Ausgaben]
 Error response stream:
 [Aufgezeichnete Fehlerausgaben]

Die Befehle für die Arduino-CLI werden intern von der Klasse `ArduinoCLICCommandLineHandler` verwaltet, weil, wie bereits zuvor in Abschnitt 7.2.1 „Manuelle Schritte mit der Arduino-IDE und entsprechende Schritte mit der Arduino-CLI“ berichtet wurde, je nach Form der Installation der Pfad zu dem Installationsort der ArduinoCLI in den Systemdateien fehlen kann. Hierfür liest es die zuvor erwähnte Datei „`arduinoCLIUtilizerKonfig.yaml`“. Zunächst wird der Wert von `arduinoCLIPathSetInPathEnvironment` gelesen und so intern festgelegt, ob dem System der Pfad zu der Arduino-CLI bekannt ist und somit direkt genutzt werden kann, oder ob er fehlt und bei den Aufrufen temporär eingestellt werden muss. Falls ersteres der Fall ist (`arduinoCLIPathSetInPathEnvironment: true`), so werden die Befehle direkt für die Ausführung übernommen. Falls das zweite der Fall ist (`arduinoCLIPathSetInPathEnvironment: false`), so wird der Wert von `arduinoCLIDirectory` gelesen, `export PATH=[Pfad zu Arduino-CLI]:\PATH &&` wird vor den auszuführenden Befehl gesetzt und dann wird der so erweiterte Befehl genutzt.

Der Kompilierbefehl `arduino-cli compile --fqbn [Boardtypenkennung] [Dateipfad] --output-dir [Ausgabeverzeichnis]` erstellt zu einer kompilierten Datei oder einem kompiliertem Projekt immer eine Reihe an Dateien. Mit `[Sketchname]` stellvertretend für die kompilierte `.ino`-Datei ohne der Typenendung oder dem Projektnamen ist dies die Liste an erstellten Dateien:

- `[Sketchname].ino.eep`
- `[Sketchname].ino.elf`
- `[Sketchname].ino.hex`
- `[Sketchname].ino.with_bootloader.bin`
- `[Sketchname].ino.with_bootloader.hex`

`CompilationCall` nutzt diesen Befehl und stellt seine Optionen so ein, dass diese in demselben Verzeichnis wie die verwendete `.ino`-Datei in einem eigenen Ordner namens „`CompiledFiles`“ abgespeichert werden. Zusätzlich wird zum späteren Nachsehen und Vergleichen die dabei genutzte Boardtypenkennung in der Textdatei „`fqbn.txt`“ bei den Ergebnisdateien gespeichert. `UploadCall` nutzt für `.hex`-Dateien den Befehl `arduino-cli upload --port [Portadresse] --fqbn [Boardtypenkennung] --input-file [Dateipfad]` und für `.ino`-Dateien `arduino-cli upload --port [Portadresse] --fqbn [Boardtypenkennung] [Dateipfad]`. Dabei greift sie auch auf diese Datei („`fqbn.txt`“) zu, um selber vor dem Upload der hochzuladenden `.hex`-Datei in den Speicher des zu programmierenden Mikrocontrollers die Kompatibilität zu prüfen und nur bei Übereinstimmung den Vorgang durchzuführen.

Die Klasse `FQBNAndCoresHandler` kann zu der bei ihrer Nutzung gegeben Boardtypenkennung prüfen, ob diese der lokalen Installation bekannt ist und, falls notwendig, eigenständig die Kern-Daten dazu aus den offiziellen Quellen heraussuchen und herunterladen. Sie verwendet hierbei die Befehle `arduino-cli board listall --format yaml`, `arduino-cli core update-index` und `arduino-cli core search [Kern-ID]`. Das Herunterladen wird über `InstallCoreForBoards` durchgeführt.

`InstallCoreForBoards` verwendet den Befehl `arduino-cli core install [Kern-ID]` um das zuvor erwähnte Herunterladen durchzuführen.

`ConnectedBoardsfinder` wird von den Klassen `BoardAutoSelectionAndInstallation` und

7. Umsetzungen

`ListAllConnectedBoardsAction`, also zwei der Klassen für die Interaktion mit dem Nutzer/ der Nutzerin, verwendet und nutzt den Befehl `arduino-cli board list --format yaml` für das Auflisten aller angeschlossenen und für die Arduino-CLI erkennbaren Boards.

Für die Interaktion zwischen dem Nutzer/ der Nutzerin und den funktionalen Teilen auf Basis der Eclipse-GUI wurde in der Datei „`plugin.xml`“ eingestellt, dass bei Rechtsklick auf eine Datei mit der Endung `.zip`, `.ino` oder `.hex` in dem Kontextmenü der Eintrag „`ArduinoCLIUtilizer`“ erscheint. Dieser führt zu einem Untermenü, dessen Einträge von der Endung der erwähnten angeklickten Datei abhängen. So soll sichergestellt werden, dass das Plugin dem Nutzer/ der Nutzerin nur dann etwas anzeigt, wenn damit potenziell etwas gemacht werden kann.

Hier folgt eine Übersicht, zu welchem Typ jeweils welche Aktionen angezeigt werden:

- Bei `.zip`-Dateien „`Generate ArduinoCLIUtilizer config file`“ und „`Install zipped arduino library`“.
- Bei `.ino`-Dateien „`List connected Arduino boards`“, „`Generate ArduinoCLIUtilizer config file`“, „`Compile and upload Arduino project`“, „`Compile Arduino project`“ und „`Verify Arduino project`“.
- Bei `.hex`-Dateien „`List connected Arduino boards`“, „`Generate ArduinoCLIUtilizer config file`“ und „`Upload.hex file`“.

Die Beschriftungen der soeben genannten Einträge wurden so gewählt, dass sie zu dem jeweiligen Zweck passen bzw. diesen kurz beschreiben. Unter der grafischen Benutzeroberfläche wird jeweils eine Klasse aus dem Paket „`de.ust.arduinocliutilizer.popup.actions`“ aufgerufen. Für eine vereinfachte Übersicht über diese Klassen und den jeweils verwendeten funktionalen Klassen siehe Abbildung 7.2.

In fast allen Fällen ist die Programmierung dieser Klassen (die Klassen aus dem Paket „`de.ust.arduinocliutilizer.popup.actions`“) auf Meldungen für den Nutzer und das Aufrufen von funktionalen Klassen beschränkt.

Die Ausnahme ist die Klasse `BoardAutoSelectionAndInstallation`, die angeschlossene Boards sucht. Wegen der einfach gehaltenen automatischen Boardauswahl bricht sie bei anderen Anzahlen an erkannten Boards mit der entsprechenden Fehlermeldung ab. Je nach Notwendigkeit lädt sie fehlende Kern-Daten zu dem angeschlossenen Board automatisch herunter. Sie stellt sowohl den Port als auch die Boardkennung zu dem angeschlossenen Board bereit. Prinzipiell könnte diese Klasse auch als funktionale Klasse gewertet werden, aber sie führt fallabhängig Meldungen aus und intern besteht sie fast nur aus Zugriffen auf vorhandene Funktionen bzw. sie bringt keine nennenswerte zusätzliche Funktionalität ein. Deshalb wurde sie nicht zu den funktionalen Klassen eingesortiert.

Wegen der Konfigurationsdatei als Anforderung für das Arbeiten mit der Arduino-CLI wurde beschlossen, keine Aktionen in der Menüleiste einzutragen, sondern nur Kontextmenüeinträge zu verwenden.

Durch die bisher umschriebenen Abläufe und Fähigkeiten werden ANF7.2.1 und ANF7.2.2 erfüllt. Tests von dieser Implementierung haben gezeigt, dass durch die Nutzung der Arduino-CLI als offizielle Software für das Arbeiten mit Arduinocode und kompatiblen Boards und Mikrokontrollern ANF7.2.3 erfüllt wird.

7.2. Integration der ArduinoCLI

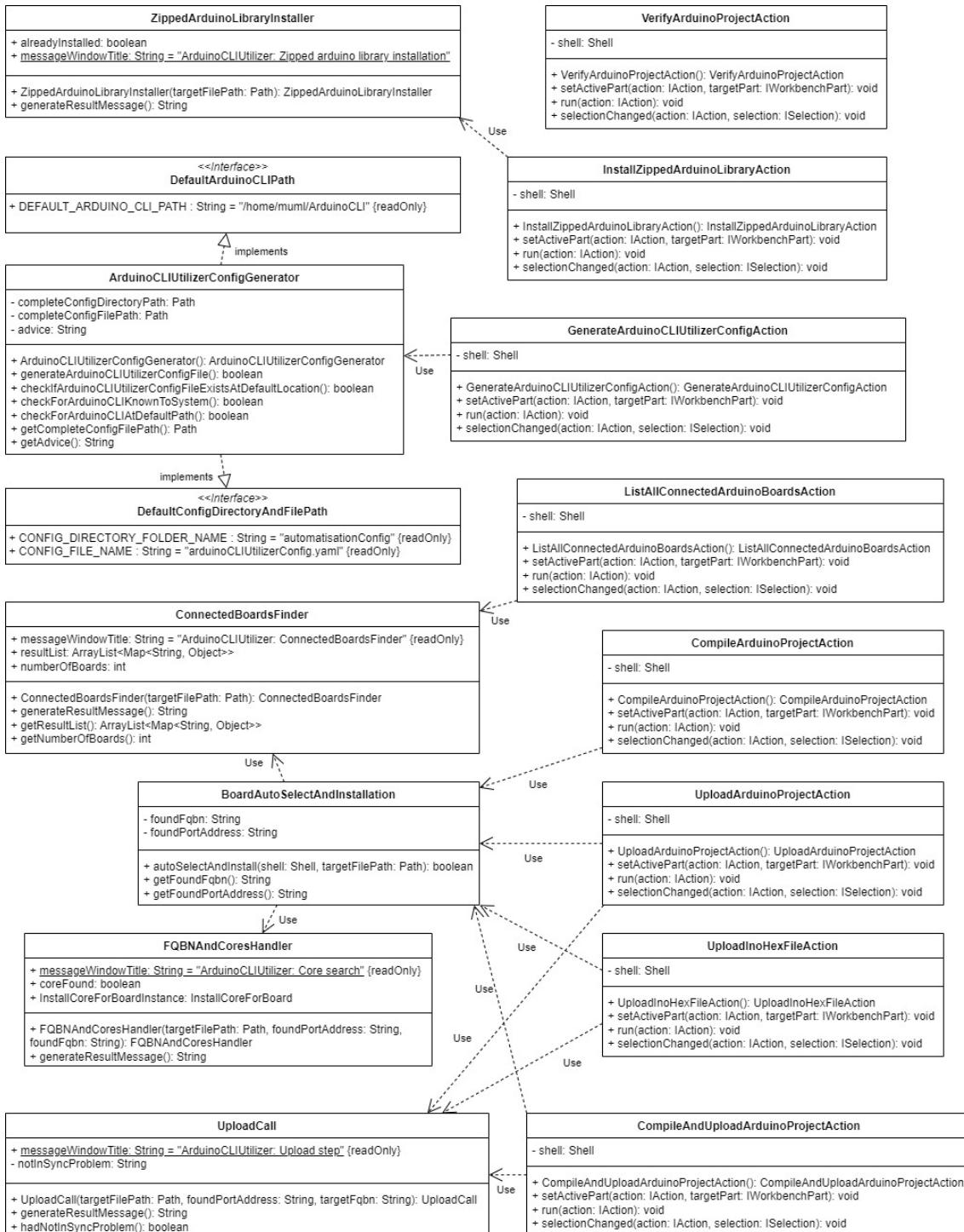


Abbildung 7.2.: Vereinfachte Übersicht über die *Action-Klassen aus dem Paket „de.ust.arduinocliutilizer.popup.actions“.

7. Umsetzungen

7.2.6. Behandlung von Boards, die ihren eigenen Boardtyp bzw. eigene FQBN nicht angeben

In manchen Fällen geben Boards keine Informationen über ihren jeweiligen Typ an. Ein Beispiel sind die Arduino Nano Exemplare, die zur Verfügung standen, obwohl es offizielle Produkte der Firma Arduino sind. Dies zeigt sich durch die leere Liste unter matchingboards.

Hier folgt der Inhalt des Textfensters von einem „List connected Arduino boards“-Aufruf mit einem angeschlossenen Arduino Mega 2560 und einem Arduino Nano als Beispiel:

```
- matchingboards:
  - name: Arduino Mega or Mega 2560
    fqbn: arduino:avr:mega
    ishidden: false
    platform: null
  port:
    address: /dev/ttyACM0
    label: /dev/ttyACM0
    protocol: serial
    protocollabel: Serial Port (USB)
    properties:
      pid: 0x0042
      serialNumber: 8593731333735141A130
      vid: 0x2341
    hardwareid: 8593731333735141A130
- matchingboards: []
  port:
    address: /dev/ttyUSB0
    label: /dev/ttyUSB0
    protocol: serial
    protocollabel: Serial Port (USB)
    properties:
      pid: 0x6001
      serialNumber: AB0LRZAC
      vid: 0x0403
    hardwareid: AB0LRZAC
```

Wie man leicht erkennen kann, kann der Arduino Mega 2560 seinen Typ angeben, aber zu dem Arduino Nano muss der Nutzer oder die Nutzerin wissen, dass es sich um einen Arduino Nano handelt.

Deshalb wurde beschlossen, dass in solchen Fällen intern ein konfigurierbarer Ersatztyp angenommen wird, um so die Informationslücke aufzufüllen. In der Konfigurationsdatei wird hierfür unter fallbackBoardIdentifierFQBN ein Boardtyp bzw. eine FQBN eingetragen. Im Fall der in dieser Ausarbeitung verwendeten Roboterautos sind es, wie bereits beschrieben wurde, die vorhandenen Arduino Nano Exemplare, die das beschriebene Problem aufweisen. Also ist standardmäßig „arduino:avr:nano“ (ohne Anführungszeichen) eingetragen. Das Bereitstellen dieser Informationen auf Anfrage ist Aufgabe der Klasse `FallbackForBoardsWithoutInternalFQBNDataHandler`, die hierfür auch die Konfigurationsdatei liest. Sobald die Klassen `BoardAutoSelectionAndInstallation` und der Pipelineschritt `LookupBoardBySerialNumber` bei der Durchführung ihrer jeweiligen Aufgabe auf ein

Board stoßen, dass keine Informationen über seinen Typen angibt, fordern sie einen Ersatztyp bei `FallbackForBoardsWithoutInternalFQBNDataHandler` an. Dann kann der jeweilige Ablauf wie normal fortgeführt werden.

Dieser Lösungsansatz funktioniert nur dann nicht, wenn gleichzeitig unterschiedliche Boardtypen angeschlossen werden, deren Exemplare ihren jeweiligen Typ nicht angeben können. Dieser Fall tritt jedoch nur unter extrem seltenen Umständen auf, weil eine Großzahl der Mikrocontroller den jeweiligen Typen angeben kann.

Ein alternativer Ansatz wäre eine Liste gewesen, die zu einer Board-Seriensnummer den jeweiligen Typen enthält, aber dies hätte für die Nutzerschaft mehr Aufwand bedeutet und sich nur in den bereits erwähnten extrem seltenen Umständen gelohnt. Also wurde dieser nicht implementiert. Falls es doch passieren sollte, kann als Workaround für solche Fälle ein kleines Skript-Programm aushelfen.

7.2.7. Hinweis auf die verwendete Version der Arduino-CLI

Es wird im Rahmen dieser Ausarbeitung die Version 0.35.3 verwendet. Sie wurde nach der Fertigstellung von „ArduinoCLUtilizer“ nicht geändert, um die Stabilität nicht zu gefährden.

7.3. Vollständige Automatisierung per Pipeline

In diesem Kapitel wird die Implementierung einer pipelinebasierten Automatisierung für CI/CD, die den in Kapitel 5 „Analyse“ beschriebenen Arbeitsablauf ermöglicht, erörtert. Dieses Pipelinesystem wurde darauf konzipiert, sowohl Teile des Eclipse-Plug-In-Projekts „ArduinoCLUtilizer“ aus Kapitel 7.2 „Integration der ArduinoCLI“ und der MUML-Plug-Ins, mit denen Stürner gearbeitet hat oder die von ihm erstellt wurden[Stü22], zu nutzen als auch eigene Teile für die Verwendung von anderen Plug-Ins bereitzustellen. Bei der Implementierung sind jedoch Probleme, die nicht im Rahmen dieser Ausarbeitung gelöst werden konnten, aufgetreten und deshalb werden in diesem Kapitel auch die vorgenommenen Improvisationen bzw. Workarounds beschrieben und erklärt. Die in diesem Kapitel beschriebenen Implementierungen sind auf die beiden Eclipse-Plug-In-Projekte „MUMLACGPPA“ (für „MUML Arduino Code Generation and PostProcessing Automatisation“) und „PipelineExecution“ aufgeteilt, müssen jedoch für die volle Funktionalität zusammen verwendet werden. Die Aufteilung in zwei Eclipse-Plug-In-Projekte hat den folgenden Grund: Die Workarounds sollten so weit wie möglich in dem separaten Projekt „PipelineExecution“ gebündelt werden, damit diese später bei einer Verbesserung von „MUMLACGPPA“ mit einer ordentlichen Implementierung der Problemstellen leichter entfernt werden können.

Dieses Kapitel ist in folgende Abschnitte unterteilt:

In 7.3.1 werden die aufgetretenen Probleme beschrieben und deren Improvisationen erklärt. Zusätzlich wird der Aufbau der umgesetzten Implementierung mit den Workarounds beschrieben.

In 7.3.2 wird die Implementierung das Eclipse-Plug-In-Projekts „PipelineExecution“ erläutert.

In 7.3.3 wird die Implementierung des Eclipse-Plug-In-Projekts „MUMLACGPPA“ bzw. des eigentlichen Systems der Pipeline, also z.B. das Interpretieren der Schritte oder die Verwaltung der Variablen und deren Daten, erklärt.

In 7.3.4 wird auf die Export-Änderung an einem bestehenden MUML-Plug-In „mechatronicuml-cadapter-component-container“ eingegangen.

7. Umsetzungen

7.3.1. Aufgetretene Probleme und Improvisationen

Im Laufe des Implementierungsprozesses traten verschiedene Probleme auf, die weder direkt gelöst noch umgangen werden konnten. Deshalb musste improvisiert werden, damit dennoch die beabsichtigten Ziele bzw. Fähigkeiten der Automatisierung umgesetzt werden konnten. Folglich wurde die ursprünglich geplante Implementierung angepasst. Die beiden Fälle, die auftraten, und die jeweilige Improvisation werden in den folgenden Abschnitten beschrieben.

7.3.1.1 Gescheitertes Verstehen und Nachahmen der automatischen Auswahlen bei der Container-Transformation und der Komponentencodes

Mangels weitreichenderen Kenntnissen bei der Export-Plug-In-Programmierung, den Arbeitsweisen der MUML-Plug-Ins und verfügbarer Zeit für das hierfür erforderliche ausführliche Einarbeiten konnten keine automatisierbaren Ressourcenauswahlen der Systemallokationen und Komponenteninstanzkonfigurationen aus den MUML-Dateien entwickelt werden. Deshalb wurde beschlossen, von den gegebenen bereits automatisch erfolgenden Auswahlen in den Export-Wizards „Container Model and Middleware Configuration“ für die Container-Transformation und „Source Code_Workspace“ für die Generation der Komponenten-Codes Gebrauch zu machen, indem aus deren Quellcode ein Export-Wizard namens PipelineExecutionAsExport und die Hilfsklassen ContainerTransformationImprovisation, ComponentCodeGenerationImprovisation und ContainerCodeGenerationImprovisation erstellt wurden.

So können die MUML-Werkzeuge ausgeführt und auch die Ausführung der Pipeline verwaltet werden.

Für mögliche spätere Entwicklungen und Problembehebungen wurde jedoch beschlossen, mit Ausnahme des Startens der Pipeline und der Verwendung der MUML-Werkzeuge dennoch das „MUMLACGPPA“ separat zu belassen und so viel wie möglich vollständig zu implementieren und selbstständig die jeweilige Aufgabe ausführen lassen zu können. Folglich wurde der Export-Wizard PipelineExecutionAsExport in „PipelineExecution“ darauf konzipiert, aus „MUMLACGPPA“ das eigentliche Pipelinesystem für das Interpretieren der Pipelinekonfigurationen und dessen Bereitstellen der auszuführenden Schritte zu nutzen. So übernimmt er bei Schritten, die für ihre geplante Funktionsweise eine Improvisation erfordern, deren beabsichtigte Funktionsweise.

Bei den Pipelineschritten, die ihre Arbeit selber durchführen können, lässt PipelineExecutionAsExport diese ihre Aufgabe eigenständig erfüllen.

7.3.1.2 Blockiertes Anzeigen von Nachrichten- und Text-Fenstern

Während der Implementierung von Export-Wizard PipelineExecutionAsExport und dabei erfolgten Tests stellte sich heraus, dass das Erstellen von z.B. Nachrichtenfenstern weder von den Klassen der Pipelineschritte noch von dem programmierten Export-Wizard möglich ist. Bei allen Versuchen zeigte Eclipse eine Warnmeldung mit dem Titel „Internal error“ und dem Text „Invalid thread access“ an und brach die weitere Ausführung ab.

Möglicherweise kann dieses Problem gelöst werden, aber aus Zeitmangel wurde beschlossen, die Nachrichten und Texte in der Konsole ausgeben zu lassen und beim Starten der Pipeline auf diesen Umstand hinzuweisen.

7.3.1.3 Ausführungsprobleme bei Pfadfindung

Während der Programmierung der Plug-Ins schlug das Heraussuchen des gesuchten Datei über die Klassen `IWorkbenchWindow`, `IStructuredSelection`, `IProject`, `IRessource` und `IFile` und ihre jeweiligen Methoden schlugen bizarreweise fehl, wenn durch `MUMLACGPPA` etwas aus `ArduinoCLUtilizer`, das die ausgewählte Datei heraussucht, verwendet wird. Für einzelne verwendete Dateien können die jeweils verantwortlichen `*Action`-Klassen das Heraussuchen zuverlässig übernehmen und die Information als Parameterbelegung weiter geben. Für größere Unternehmungen hingegen wird der Ordnerpfad des aktiven Projekts benötigt. Also wurde beschlossen, diesen Projektordnerpfad global zugänglich zu speichern. Dies wurde über das Eclipse-Plug-In „`ProjectFolderPathStoragePlugIn`“ mit seiner Klasse `ProjectFolderPathStorage` mit dem statischen Feld `projectFolderPath` vom Typ `Path` ermöglicht. Jede im Rahmen dieser Ausarbeitung erstellte funktionale `*Action`-Klasse ermittelt den Projektordnerpfad und legt diesen in dem statischen Feld dafür ab. So können mehrere Komponenten gleichzeitig diesen abrufen und mit den jeweiligen Dateien arbeiten. Später wurde für den Typ `IProject` das statische Feld `project` hinzugefügt.

7.3.2. Implementierung der Pipelineausführung

Es wurde das Eclipse-Plug-In-Projekt „`PipelineExecution`“ angelegt, das den Export-Wizard `PipelineExecutionAsExport` enthält. Dieser ist darauf konzipiert, aus „`MUMLACGPPA`“ das eigentliche Pipelinesystem bzw. dessen Konfigurationslesesystem mit einer Konfigurationsdatei zu instanzieren, deren Konfigurationen validieren zu lassen und nacheinander die auszuführenden Schritte anzufordern. Dies erfolgt innerhalb der Funktion `addPages()`, also beim Aufbauen der verschiedenen Fenster, wie sie normalerweise für das Konfigurieren von Exportvorgängen verwendet werden. Zu jedem erhaltenen Schritt soll `PipelineExecutionAsExport` die Ausführung der jeweiligen beabsichtigten Funktionsweise sicherstellen. Abseits der Ausführung von der Pipeline wurden für die Nutzerschaft Informationsmeldungen entworfen, die nach Notwendigkeit angezeigt werden, und es wurde eine mögliche Optimierung vorgenommen.

7.3.2.1 Pipelineschritte mit improvisierter Durchführung der jeweiligen Funktion

Bei den Pipelineschritten, die nicht eigenständig ihren Zweck erfüllen können, übernimmt `PipelineExecutionAsExport` die jeweilige Arbeit.

Für beispielsweise `ContainerTransformation`, das der Container-Transformation entspricht, erstellt `PipelineExecutionAsExport` das aus dem Originalcode übernommene Auswahlfenster für die Auswahl des Systemallokationseintrages aus der Datei „`roboCar.muwl`“, aber die weiteren Daten für die Middleware der Kommunikation bzw. den Ansatz für die Kommunikation und für den Speicherort des zu generierenden Containers werden aus den gegebenen Konfigurationen übernommen und bei der Nutzung von `ContainerGenerationJob` aus den MUML-Werkzeugen eingetragen.

7.3.2.2 Starten von funktionierenden Pipelineschritten

Bei den Pipelineschritten, die ihre Arbeit selber durchführen können, lässt `PipelineExecutionAsExport` sie ihre jeweilige Aufgabe eigenständig erfüllen, indem es bei diesen Instanzen jeweils die Ausführungsfunktion `execute()` aufruft.

7. Umsetzungen

7.3.2.3 Situationsabhängige Informationsmeldungen

Um bei fehlenden oder fehlerhaften Konfigurationsdateien die Ausführung der Pipeline nicht zu starten, sondern stattdessen einen Hinweis auf den entsprechenden Missstand anzuzeigen, übernimmt `PipelineExecutionAsExport` auch das Überprüfen der Konfigurationsdateien auf deren Existenz. Wenn aber beispielsweise keine Schritte aufgespürt werden, die auf einer Nutzung von „`ArduinoCLIUtilizer`“ beruhen, so wird weder nach dessen Konfigurationsdatei gesucht noch ihre Existenz gefordert. Zusätzlich wird die Überprüfung der Pipelinekonfigurationen gestartet. Wenn hierbei ein Fehler aufgespürt wird, so wird dieser gemeldet. Nur wenn alle Prüfungsschritte keinen Fehler finden konnten, kann die Pipeline gestartet werden. Weitere Details zu dem System der Pipeline und den Korrektheitsprüfungen werden in Abschnitt 7.3.3 „Implementierung des eigentlichen Systems der Pipeline“ erläutert.

7.3.2.4 Weitere Varianten

Die Varianten für das jeweilige alleinige Ausführen der MUML-Werkzeuge und der Post-Processing-Schritte funktionieren ähnlich, wenn auch auf die jeweilige Aufgabe reduziert. Hierfür erben sie von `PipelineExecutionAsExport`, verwenden jedoch nur die jeweils relevanten Funktionen. Zusätzlich sind jeweils der Aufbau und die Folge der verschiedenen Fenster sowie der interne Ausführungsablauf auf den entsprechenden Zweck angepasst bzw. vereinfacht.

7.3.3. Implementierung des eigentlichen Systems der Pipeline

Das eigentliche Pipelinesystem wurde als das Eclipse-Plug-In-Projekt „MUMLACGPPA“ implementiert und im Folgenden wird auf seine Konzepte und Teile eingegangen.

7.3.3.1 Variablen- und Werte-Verwaltung

Die Klasse `VariableHandler` wurde dafür entworfen, die Variablen und deren Nutzung wie dem Festlegen von Werten zu verwalten. Hierbei werden eine Liste mit initialisierten Variablennamen und zwei HashMaps, die zu den verschiedenen Namen entsprechend den jeweiligen Typ oder Inhalt speichern, verwendet.

In Folge von mangelnden Kenntnissen mit Generics unter Java und gescheiterten Implementierungsversuchen sind die Variableninhalte selbst Instanzen der Klasse `VariableContent`. Diese sieht das Speichern von Daten als String bzw. Zeichenkette vor und stellt für das Lesen bzw. Abrufen die Getter-Funktionen mit den Umwandlungen für die verschiedenen Datentypen bereit.

Weil das globale Zugreifen auf die Variablenverwaltung als gefährlich bewertet wurde, wird beim Lesen der Konfigurationsdatei ein `VariableHandler` angelegt und jeder Pipelineschritt erhält bei seiner Instanziierung eine Referenz darauf.

7.3.3.2 Konzept hinter den Pipelineschritten

Zu den einzelnen Typen von Pipelineschritten wurden verschiedene Klassen mit der jeweiligen Funktionalität geschrieben. Diese erben jedoch gemeinsame Funktionen von der abstrakten Klasse `PipelineStep`. So wird durch das Prinzip des Polymorphismus die Nutzung vereinfacht, indem z.B. eine mit eben dieser Klasse als Datentyp festgelegte Liste alle Pipelineschritte speichern kann. Beim Iterieren über diese (Liste) kann bei allen darin enthaltenen Schritten die Ausführung leicht aufgerufen werden.

Die gemeinsamen Funktionen behandeln die Instanziierung, das interne Festlegen der zu konfigurierenden Parameter, das Überprüfen der eigenen Einstellungen, das Lesen und Schreiben von Variablen, das Interpretieren von Pfadangaben sowie das Ausführen der jeweiligen Aufgabe.

Die abstrakte statische Funktion `getRequiredInsAndOuts()` ist erforderlich, weil aus Zeitgründen keine Möglichkeit gefunden wurde, wie über statische Felder innerhalb der Unterklassen von `PipelineStep` der jeweilige erwartete Konfigurationsaufbau festgelegt werden kann. Mehr zu den Überprüfungsabläufen wird in Abschnitt „7.3.3.4 Überprüfung der Pipelinekonfigurationen auf Fehler“ beschrieben.

Es ist aber zu beachten, dass `getContentOfInput(String key)`, alle Varianten von `setContentOfOutput(...)` und `getResolvedPathContentOfInput(String key)` Hilfsfunktionen sind, die wegen der Improvisation bei der Nutzung von MUML-Werkzeugen per Export-Wizard (siehe Abschnitt 7.3.1 „Aufgetretene Probleme und Improvisationen“) hinzugefügt wurden.

7.3.3.3 Interpretation der Pipelinekonfiguration

Die Interpretation der Konfigurationen erfolgt über die Klasse `PipelineSettingsReader`. Zuerst wird bei ihrer Instanziierung die Konfigurationsdatei geladen, die daraufhin interpretiert wird. Hierbei werden die Bereiche für die Einstellungen in folgender Reihenfolge interpretiert:

1. Zuerst `VariableDefs`, um die Variablen und ihre Werte zu erhalten.
2. Dann `TransformationAndCodeGenerationPreconfigurations`, um für die einzelne vorkonfigurierte Ausführung von MUML-Werkzeugen die Einstellungen zu erhalten, falls sie von späteren Teilen der Konfigurationsdatei geladen werden.
3. Danach `PostProcessingSequence`. So können einzelne Einstellungen aus `TransformationAndCodeGenerationPreconfigurations` geladen werden, aber in `PipelineSequence` kann es komplett eingefügt werden. Hierbei ist das Einbinden von einzelnen Schritten aus der Post-Processing-Sequenz Zwecks Zuverlässigkeit bei Änderungen nicht erlaubt.
4. Zuletzt `PipelineSequence`. So können sowohl einzelne Einstellungen aus `TransformationAndCodeGenerationPreconfigurations` als auch die komplette Schrittliste von `PostProcessingSequence` mit eingebunden werden.

Es ist erwähnenswert, dass alle Versuche, zu den eingestellten Pipelineschritten die jeweiligen Klasse anhand des angegebenen Namens herauszusuchen und zu instanzieren, fehlgeschlagen sind. Deshalb wird sozusagen mit einem Wörterbuch gearbeitet. Hierfür wird die Klasse `PipelineStepDictionaryMUMLPostProcessingAndArduinoCLIUtilizer` verwendet. Diese führt mittels einer langen switch-case-Abfrage jeweils das Heraussuchen und Instanziieren der erforderlichen Unterklasse von `PipelineStep` durch. Um für eigene Zwecke leichter eigene Klassen verwenden

7. Umsetzungen

zu können, wurde eine Superklasse `PipelineStepDictionary` angelegt, sodass leichter eigene Implementationen für das Heraussuchen der Klassen erstellt und diese in dem Konstruktor für eine `PipelineSettingsReader`-Instanz eingetragen werden können.

7.3.3.4 Überprüfung der Pipelinekonfigurationen auf Fehler

Vor dem möglichen Ausführen der Pipeline wird von `PipelineExecutionAsExport` oder einer ihrer Unterklassen eine Fehlersuche gestartet. Falls ein Konfigurationsfehler gefunden wird, wird zu diesem eine Meldung ausgegeben und die Pipeline wird nicht gestartet. Die implementierten Mechanismen der Fehlersuche können folgendes aufspüren:

1. Fehlende Konfigurationsdateien
2. Syntax-Fehler
3. Typen-Fehler
4. Werteangaben, die gegen die YAML-Standards verstößen
5. unerlaubte Strukturen wie z.B. `from direct` oder sonstige Strukturverstöße
6. fehlplatzierte oder falsche Schlüsselwörter
7. fehlende oder überschüssige Parameterangaben
8. Verwendung von nicht vorhandenen oder noch nicht initialisierten Variablen
9. Versuchte oder versehentliche Typenänderungen bei Variablen
10. Verwendung von nicht vorhandenen Einträgen aus `TransformationAndCodeGenerationPreconfigurations`
11. Versuchtes rekursives Einbinden der Einträge aus `PostProcessingSequence` in `PostProcessingSequence`, d.h. nur in `PipelineSequence` darf der Inhalt aus `PostProcessingSequence` eingebunden werden.
12. Unerlaubte Pipelineschritttypen in `TransformationAndCodeGenerationPreconfigurations`

Das Prüfen der Einträge dieser Liste erfolgt teils durch die verwendete YAML-Bibliothek „`Snakeyaml`“ (Version 2.2), teils durch das Prüfen der Strukturen und Inhalte beim Interpretieren und teils durch die Funktion `checkForDetectableErrors()`.

Die Funktion `checkForDetectableErrors()` ist der Startpunkt für die Suche nach Fehlern, die nicht das Einlesen und Interpretieren selbst betreffen.

Die Fehlersuche bei den Variablen und beim Datenfluss erfolgt hinsichtlich des Existierens oder Fehlens von Variablen, der Typenprüfung und des versuchten oder versehentlichen Typenänderns. Der logische Pipelineablauf wird auch beachtet.

Hier ein Beispielszenario: Eine Variable namens „`ifSuccessfulContainerTransformation`“, die dynamisch in dem Bereich `TransformationAndCodeGenerationPreconfigurations` in `ContainerTransformation` als Ausgabeziel des Ausgabeparameters `ifSuccessful` erstellt wird, ist in dem Bereich `PipelineSequence` erst nach dem Eintrag `from TransformationAndCodeGenerationPreconfigurations: ContainerTransformation` verfügbar.

Hierbei wird ebenfalls die Überprüfung der Parameterangaben bei den verschiedenen Instanzen

der den konfigurierten Pipelineschritten entsprechenden Klassen gestartet. Wenn nicht alle der erwarteten Parameter der jeweiligen Klasse oder unerwartete Einträge gefunden werden, so wird dies als Fehler gemeldet und dabei ein Vergleich der jeweils angegebenen und der erwarteten Einträge angezeigt.

7.3.3.5 Zugriff auf die interpretierte Pipeline

PipelineSequence wurde dafür konzipiert, wie eine Liste verwendet zu werden. Dementsprechend wurde das Konzept der Methoden `getNext()` und `hasNext()`, die üblicherweise für das Iterieren über Listen oder ähnliche Klassen verwendet werden, auf den Inhalt der Pipelineeinstellungen angewandt.

Für das Abfragen der Listen PostProcessingSequence und PipelineSequence wurden die entsprechenden Methoden `hasNextPostProcessingStep()`, `getNextPostProcessingStep()`, `hasNextPipelineSequenceStep()` und `getNextPipelineSequenceStep()` implementiert. Um das erneute Einlesen und Überprüfen zu ersparen, wurden die Methoden `resetPostProcessingProgress()` und `resetPipelineSequenceProgress()` konzipiert, die den Fortschritt über die entsprechende Liste zurücksetzen. Es wird hierbei jedoch davon ausgegangen, dass nur erfahrene Programmierer oder Programmiererinnen mit so tiefen Teilen des Pipelinesystems arbeiten. Vermutlich werden in einem Großteil der Modifikationsfälle weitere Pipelineschritte notwendig sein. Deshalb wurden in diesen Methoden keine Sicherheitsvorkehrungen integriert.

Das direkte Abfragen auf die Einstellungen unter TransformationAndCodeGenerationPreconfigurations erfolgt per `getTransformationAndCodeGenerationPreconfigurationsDef(String stepName)`. Ob ein Schritt unter TransformationAndCodeGenerationPreconfigurations konfiguriert ist, kann per `IsEntryInTransformationAndCodeGenerationPreconfigurations(String name)` abgefragt werden.

7.3.3.6 Generierung der Beispiele

Um der Nutzerschaft den Einstieg in die Nutzung dieses Pipelinesystems zu erleichtern, wurden zwei Dateigeneratoren für eine Liste mit Beispielkonfigurationen aller Pipelineschritte und für eine auf den Anwendungsfall dieser Ausarbeitung ausgerichtete Beispiel-Pipelinekonfiguration konzipiert.

Für das erste erfolgt der Aufruf per Rechts-Klick auf „roboCar.muml“]/„MUMLACGPPA“/„Generate all pipeline step example settings“ und dann wird eine Textdatei mit allen Pipelineschritten und direkten Beispielbelegungen der Parameter generiert.

Die Generierung der Beispiel-Pipelinekonfiguration wird per Rechts-Klick auf „roboCar.muml“]/„MUMLACGPPA“/„Generate default/example pipeline settings“ gestartet. Hierbei werden jedoch für die Seriennummern der zu verwendenden Boards, WLAN-Informationen und MQTT-Informationen keine reellen Daten, sondern Dummy-Werte eingetragen. Diese fangen mit „Dummy“ an und ein Informationsfenster weist auf diesen Umstand hin. So können die zu ersetzenen Dummy-Werte mit einer Suche nach „Dummy“ leicht und schnell gefunden werden.

7. Umsetzungen

7.3.3.7 Generierung einer lokalen „SofdCar-Hal“-Konfigurationsdatei

Die Bibliothek „Sofdcar-HAL“ erfordert u.a. Konfigurationen, die in einer als Konfigurationsdatei dienenden Codedatei „Config.hpp“ eingetragen sind. Der ursprüngliche Post-Processing-Ablauf [Reib] sah hierfür das Herunterladen von ihr aus dem Repository [Reic] und dem Platzieren von ihr in jedem „*CarDriverECU“-Ordner vor. Wegen möglicher Versionsunterschiede zwischen den Dateien in der Versionsverwaltung und der lokalen Kopie kann dies beispielsweise Kompilierfehler oder unerwartetes Verhalten verursachen. Deshalb wurde ein zu dem Herunterladen alternativer Schritt konzipiert, durch den stattdessen eine lokale Instanz aus dem Ordner „automatisationConfig“ kopiert werden kann. Zur leichteren Nutzung wurde ein Generator für diese Konfigurationsdatei konzipiert, der per Rechts-Klick auf „roboCar.muml“]/„MUMLACGPPA“/„Generate local settings for Sofdcar-Hal“ gestartet werden kann. Für das intuitive Verständnis dieser Konfigurationsdatei wird sie innerhalb von „automatisationConfig“ unter dem Namen „LocalSofdcarHalConfig.hpp“ aufgelistet, aber ihre Kopien an den Zielorten unter dem korrekten Namen „Config.hpp“.

7.3.4. Export-Änderung an MUML-Plug-In „mechatronicuml-cadAPTER-component-container“

Es wurde die Klasse GenerateAll aus dem Package „org.mumlarduino.adapter.container.ui.common“ aus dem MUML-Plug-In-Projekt „mechatronicuml-cadAPTER-component-container“ bzw. dessen Unterprojekt „org.mumlcodegen.componenttype.export.ui“ benötigt. Deshalb wurde in den Einstellungen von diesem Unterprojekt das genannte Package in den Plug-In-Einstellungen per „Exported Packages“ als nach außen sichtbar eingestellt. So kann es für die automatisierte Generation des Containercodes verwendet werden.

8. Evaluation

Dieses Kapitel befasst sich mit der Evaluation der erreichten Umsetzungen, die im Laufe dieser Ausarbeitung erbracht wurden.

Hierfür werden die verschiedenen Kategorien der erbrachten Leistungen und Ergebnisse beleuchtet und ausgewertet:

- Forschungsziele und deren Antworten
- Verschiedene Qualitätsbewertungen
- Anwendungsszenario (siehe Kapitel 4 „Anwendungsszenario“) als Praxistest und Leistungsvergleich

Zum Abschluss wird auf die verschiedenen Gefahren für die Validität eingegangen.

Dieses Kapitel ist wie folgt aufgeteilt:

In 8.1 „Vorgehensweise“ werden die Vorgehensweisen im Hinblick auf die Forschungsziele, die Tests der entwickelten Software, die Qualitätsbewertungen und das Durchführen des Anwendungsszenarios als wichtiger Leistungsvergleich beschrieben.

In 8.2 „Ergebnisse“ werden die Ergebnisse hinsichtlich der Forschungsziele, den Qualitätsbewertungen und des Anwendungsszenarios als Leistungsvergleich vorgestellt.

In 8.3 „Diskussion“ findet eine Diskussion zu allen Themen statt, die gleichzeitig die Ergebnisse und Analyseresultate zusammenfasst.

In 8.4 „Gefahren für die Gültigkeit“ wird auf mögliche Gefahren für die Validität eingegangen.

8.1. Vorgehensweise

8.1.1. Vorgehen bei den Forschungszielen

Zunächst werden die im Laufe dieser Ausarbeitung definierten und beantworteten Forschungsziele noch einmal benannt und die jeweilige zusammengefasste Antwort dargelegt. Diese Antworten dienten als Vorbereitung für das Entwerfen und Umsetzen der Anpassungen auf die während dieser Ausarbeitung verwendeten Roboterautos sowie der Automatisierung der Codegeneration ausgehend von MUML-Modellen und des Post-Processings. Dies geschah, indem durch diese Forschungsziele und deren Antworten ein Großteil der benötigten Informationen zusammengetragen wurde.

8. Evaluation

8.1.2. Vorgehen beim Überprüfen der entwickelten Software hinsichtlich der Korrektheit ihrer Funktionsweise und ihrer Ausgaben

Während der Entwicklung der verschiedenen Softwarekomponenten wurden verschiedene Tests durchgeführt, um sicher zu stellen, dass diese jeweils auf die gewünschte Weise arbeiten und die erwarteten Ergebnisse liefern.

Hinsichtlich der Roboterautos wurde aber auch geprüft, ob in der Praxis alles den Erwartungen entspricht bzw. ob Verhaltensfehler auftreten.

MUML-Modellanpassungen

Hier wurden grobe Vergleiche zwischen alten und neuen generierten Dateien sowie den jeweils erwarteten Unterschieden vorgenommen.

Post-Processing-Änderungen

Auch hier wurden alte und neue nachbearbeitete Codedateien sowie die jeweils erwarteten Unterschiede miteinander abgeglichen.

Änderungen an den Modellautos

Hier wurde manuell geprüft, ob die Hardware wie gewünscht arbeitet.

Die geänderte Kommunikation wurde getestet, indem an den Datenleitungen, die den Koordinator-AMK mit dem Fahrer-AMK verbinden, ein weiterer AMK angeschlossen wurde, der darauf programmiert wurde, die Kommunikation mit aufzunehmen und an den tragbaren PC weiterzuleiten. So wurde geprüft, ob die Kommunikation stattfindet und der Inhalt der Nachrichten jeweils den Erwartungen entspricht.

Das Prüfen der Zuverlässigkeit der WiFi-Module fand teils auf ähnlichem Weg statt und teils auf Basis der bereits vorhandenen seriellen Debug-Nachrichten.

ArduinoCLIUtilizer

Jede entwickelte Klasse wurde zunächst mit Debug-Ausgaben versehen, um ihre dynamische Ausführung und ihre Ergebnisse, z.B. die Ausagaben der Arduino-CLI, beobachten zu können. Sobald jeweils der gewünschte Ablauf erreicht wurde, wurden diese Ausgaben entfernt.

Die Prüfung, ob die Arduino-CLI wirklich die gleichen Ergebnisse wie die Arduino-IDE liefert, erfolgte, indem Beispielprogramme jeweils zweimal auf einen Arduino Mega 2560 hochgeladen wurden: Das erste Mal mit der Arduino-IDE und danach das zweite Mal mit der Arduino-CLI.

Nach jedem Hochladevorgang wurde jeweils das Verhalten beobachtet und mit dem verwendeten Testcode verglichen. Danach wurde analysiert, ob es zwischen den beiden Methoden Verhaltensunterschiede gab. Wie in Abschnitt „Test der Arduino-CLI“ berichtet wird, konnten keine Unterschiede gefunden werden.

MUMLACGPPA

Während der Entwicklung wurden JUnit-Tests verwendet, um automatisierte Funktionstests zu nutzen. Diese Tests sind noch in dem Code vorhanden.

Die Teile, die nicht durch diese Testform überprüft werden konnten, wie beispielsweise Pipelineschritte, welche die Arduino-CLI verwenden, wurden an den entsprechenden Stellen mit Debug-Ausgaben versehen und ihre Ergebnisse, z.B. ob der Inhalt von „SavedResponses“ und dessen Informationsdateien den Erwartungen entspricht und wie sich der verwendete AMK verhält, wurden untersucht. Man kann hier also von manuellen Tests sprechen.

PipelineExecution

Hier wurde jede Komponente manuell getestet. Für die internen Abläufe wurden für weitere Informationen Debug-Ausgaben integriert.

Hinsichtlich der erwarteten Ergebnisse wurden die entsprechenden Beobachtungen durchgeführt. Hierbei wurden gezielt entworfene Abläufe in der Pipelinekonfiguration eingetragen und genutzt. Für vollständige Verhaltenstests mit einer Pipeline, die das Anwendungsszenario von den Modelltransformationen bis einschließlich des Hochladens auf die AMKs automatisieren soll, wurden neben ausgeliehenen Arduino Nanos auch private Arduino Megas verwendet, um abgesehen von der fehlenden Roboterauto-Hardware die Durchführung von Modellversuchen des Anwendungsszenarios exakt nachzustellen.

8.1.3. Vorgehen für Qualitätsbewertungen

Zu den im Laufe dieser Ausarbeitung vorgenommenen Umsetzungen wird die Codequalität nach den Kriterien aus Abschnitt 3.4 „Codequalität“, bzw. nach der ISO-25010-Norm [Anw], ausgewertet. Zu dem Pipelinesystem wurde untersucht, wie gut es die Eigenschaften und Fähigkeiten von CI/CD-Pipelines aus Abschnitt 3.5 „CI/CD-Pipeline“ nach Bigelow [Big] erfüllt. Bei diesen Themen bzw. deren Bewertungen wird jeweils eine Taxonomie angewandt. Mit der Ausarbeitung von Usman et al. [UBBM17] als Bezugspunkt sind die Taxonomien in Abschnitten 8.2.3 „Qualitätsbewertungen“ und 8.2.4 „Qualität der CI/CD-Pipeline“ wie folgt aufgebaut:

Absicht: Bei der jeweiligen Umsetzung soll geprüft werden, wie gut die jeweils allgemein akzeptierten Aspekte, Attribute, Faktoren oder Funktionen, die in dem Forschungsfeld oder in der Branche üblicherweise als wichtig erachtet werden, erfüllt werden.

Beschreibungsgrundlagen bzw. Terminologie: Es werden die Begriffe bzw. es wird die Terminologie, die in dem Forschungsfeld oder in der Branche der jeweiligen Umsetzung üblicherweise verwendet werden, übernommen.

Klassifizierungsprozedur: Es wird geprüft, wie weit die Umsetzung die jeweiligen Eigenschaften oder/und Fähigkeiten, die in dem entsprechenden Forschungsfeld oder in der entsprechenden Branche üblicherweise als wichtig erachtet werden, erfüllt. Der jeweilige Anteil wird in einer Bewertungsskala von 3 bis 0 klassifiziert bzw. eingestuft. Diese wird als Erfüllungsstufe bezeichnet und ist wie folgt definiert:

8. Evaluation

- 3 Die Umsetzung erfüllt alle beschriebenen Eigenschaften oder/und besitzt alle beschriebenen Fähigkeiten.
- 2 Die Umsetzung erfüllt die beschriebenen Eigenschaften zu einem hohen Anteil oder/und besitzt einen hohen Anteil der beschriebenen Fähigkeiten.
- 1 Die Umsetzung erfüllt die beschriebenen Eigenschaften zu einem geringen Anteil oder/und besitzt einen geringen Anteil der beschriebenen Fähigkeiten
- 0 Die Umsetzung erfüllt keine der beschriebenen Eigenschaften oder/und besitzt keine der beschriebenen Fähigkeiten.

8.1.4. Das Anwendungsszenario als Praxistest

Das in Kapitel 4 „Anwendungsszenario“ beschriebene Anwendungsszenario wird wie folgt durchgeführt:

Schritt 1: Die im Rahmen dieser Ausarbeitung implementierte und konfigurierte CI/CD-Pipeline wird auf die modifizierten MUML-Modelle angewandt, um Arduinocode bzw. nutzungsbereite Sketches zu erhalten, die direkt von der Arduino-IDE kompiliert und auf die Roboterautos bzw. deren AMKs hochgeladen werden können.

Hierfür wird die generierbare Pipelinekonfiguration geöffnet und angepasst:

Schritt 1.1: Die Seriennummern der verschiedenen AMKs der Roboterautos werden passend eingetragen.

Schritt 1.2: Die WLAN-Informationen des verwendeten Hotspots wurden eingesetzt. Das Verwenden des Uni-WLANs wurde wegen dessen Sicherheitsvorkehrungen abgelehnt.

Schritt 1.3: Die Daten für die Verbindung mit dem MQTT-Server wurden eingetragen.

Schritt 1.4: Die Schritte für das Kompilieren und Hochladen wurden entfernt. So sollen die nutzungsbereiten Sketches generiert werden, aber keine weiteren Schritte durchgeführt werden.

Schritt 2: Anschließend werden diese Sketches über die Arduino-IDE auf die entsprechenden AMKs hochgeladen.

Schritt 3: Danach werden die beiden Roboterautos auf der Teststrecke platziert und gestartet, um ihr Verhalten beobachten zu können.

Das Weglassen des automatisierten Hochladens hat den folgenden Grund: Es stand kein tragbarer PC zur Verfügung, der ausreichend Leistung für eine virtuelle Maschine mit der verwendeten Lubuntu-Installation mit MUML bereitstellen konnte.

Dies war jedoch auf dem privaten stationären PC möglich und deshalb wurde beschlossen, zu Hause den Code vorzubereiten, um ihn über den tragbaren Tablet-Computer auf die AMKs hochzuladen. Durch die Verkapselung als virtuelle Maschine kann die entwickelte Software nicht vom Zustand des privaten PCs gestört werden und somit sollte von dieser Seite keine Gefahr für die Qualität ausgehen.

Frühere Tests mit privaten AMKs haben bestätigt, dass das automatisierte Kompilieren und Hochladen korrekt funktioniert, wodurch auch hier keine Qualitätsminderung zu befürchten ist. Das gleiche gilt für das Hochladen von Sketches über die Arduino-IDE statt der Arduino-CLI, wie bereits in Kapitel 7.2 „Integration der ArduinoCLI“ berichtet wurde.

8.2. Ergebnisse

8.2.1. Die Forschungsziele und ihre Resultate

FZ1.1 Welche Schritte des Arbeitsablaufes werden von welchen Komponenten und mit welchen Daten ausgeführt?

Arbeitsschritt 1: T3.2: „Deployment Configuration bzw. Container Transformation“ (siehe Abbildung 5.1) Per Export-Operation „MechatronicUML“/„Container Model and Middleware Configuration“ wird anhand der Systemallokationen-Ressource in „roboCar.muml“ die Datei „MUML_Container.muml_container“ mit den verschiedenen Containern für die verschiedenen Komponenten und deren Konfigurationen generiert.

Arbeitsschritt 2: T3.6 und T3.7: „Container Code Generation“ (siehe Abbildung 5.1) Dies wird per [Rechtsklick auf „MUML_Container.muml_container“-Datei]/„mumlContainer“/„Generate Arduino Container Code“ gestartet. Am Zielort befinden sich die verschiedenen APIs und die Ordner für die verschiedenen ECUs bzw. AMKs. In letzterem befinden sich Code-Dateien wie einige interne Bibliotheken aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“, Package „org.mumlarduino.adapter.container“ und Ordnerverzeichnis „resources/container_lib“. Intern erfolgt die Generierung per „GenerateAll“ aus dem Package „org.mumlarduino.adapter.container.ui.common“ aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“.

Arbeitsschritt 3: T3.3: „Component Code Generation“ (siehe Abbildung 5.1) Hier werden per Export-Operation „MechatronicUML“/„Source_Code_workspace“ aus der Datei „roboCar.muml“ die Komponentencodes generiert und im Ordner „fastAndSlowCar_v2“ platziert.

Arbeitsschritt 4: T3.8: „Program Building“ (siehe Abbildung 5.1) Im ersten Teil davon, dem Post-Processing, wird zunächst der generierte unvollständige Code in einen direkt verwendbaren und vollständigen Zustand gebracht. So kann die Arduino-IDE die verschiedenen .ino-Dateien jeweils korrekt kompilieren.

Für die nach Stürners Ausarbeitung verwendete Version der Roboterautos und der Bibliothek „Sofdcar-HAL“ hat Georg Reißner einen angepassten Verlauf beschrieben, der unter [Reib] nachgeschlagen werden kann.

Im letzten Teil, dem Deployment bzw. Hochladen auf die verschiedenen AMKs, werden die verschiedenen .ino-Dateien mit der Arduino-IDE geöffnet und jeweils auf dem entsprechenden AMK hochgeladen [Stü22].

FZ1.2 Welche Schritte des Arbeitsablaufes müssen manuell durchgeführt werden?

In den Arbeitsschritten 1 bis 3 aus FZ1.1 wird jeweils ein interner Vorgang ggf. konfiguriert und gestartet. Von Schritt 4 erfolgt das Post-Processing gänzlich manuell. Beim Deployment werden

8. Evaluation

die verschiedenen .ino-Dateien über das Ordnersystem geöffnet und auf den jeweiligen AMK hochgeladen.

FZ1.3 Welche Kriterien sind für die Auswertung von Ansätzen relevant?

Es konnten keine Bewertungskriterien für Modelländerungen gefunden werden, also werden Kriterien für die Code-Qualität verwendet.

FZ2.1 Werden die entsprechend betroffenen Artefakte standardmäßig integriert? Und falls ja, wie?

Anpassungen an den MUML-Modellen: Die Modelle der verschiedenen Komponenten und deren Instanzen sowie die Ports und Verbindungen zwischen diesen werden in den verschiedenen Diagrammen modelliert.

Änderung des Kommunikationsprotokolls zwischen den Boards innerhalb eines Roboterautos:

Die standardmäßige Umsetzung einer Änderung des Kommunikationsprotokolls von Inter-Integrated Circuit (I2C) nach „Seriell“ würde wahrscheinlich das Hinzufügen der seriellen Kommunikation als Kommunikationstyp sowie das Hinzufügen der Dateien „SerialCustomLib.cpp“ und „SerialCustomLib.hpp“ als interne Ressourcen vorsehen.

Post-Processing: Das Post-Processing wurde vor der Ermöglichung der Automatisierung immer manuell durchgeführt. Für die genaue Liste der Handgriffe siehe Kapitel 5 „Analyse“.

FZ2.2 Welche validen Anpassungsansätze gibt es?

Anpassungen an den MUML-Modellen: Es gab zwei Möglichkeiten:

1. Der erste Ansatz beruhte auf dem Beibehalten der alten Architektur und dem Ersetzen der alten Komponenten mit entsprechenden neueren Komponenten. Architekturänderungen wären einzig für das Richtungslenken erfolgt. Die Komponente DriveControl aus Stürners Ausarbeitung[Stü22] hätte also seitens des Modells direkten Zugriff auf konkrete Kindklassen der abstrakten Klassen SteerableAxe und Motor aus Sofdcar-HAL erhalten. Das Konzept der Modularisierung würde also durchbrochen werden.
2. Das Priorisieren der Modularisierung, d.h. die Komponente PowerTrain, die das Einstellen der Geschwindigkeit repräsentiert, würde mit einer Kindklasse der abstrakten Klasse DriveController ersetzt werden. So würde die Modularisierung beibehalten werden und alles aus dieser Bibliothek könnte so arbeiten oder verwendet werden, wie es jeweils vorgesehen ist. Um Verwechslungen vorzubeugen, würde DriveControl zu CourseControl umbenannt werden. Dies würde auch zu einer passenderen Namensgebung führen, weil dann diese Kontrollkomponente vielmehr den Kurs auf den Fahrbahnen festlegt, aber nicht für Lenken beim Befolgen der Bahn verantwortlich ist.

Post-Processing: Hier gibt es nur einen Ansatz, der erwähnenswert ist: Die Namensänderungen werden berücksichtigt und beim Ausfüllen der API-Mapping-Dateien wird zusätzlich entsprechend das Abfragen des Winkels eingetragen. Details zu den Änderungen werden im Ergänzungsmaterialabschnitt D.1 „Änderungen am Post-Processing-Ablauf“ beschrieben. Die anderen hatten sich nur in der Reihenfolge unterschieden.

FZ2.3 Welcher Anpassungsansatz erfüllt die Kriterien am besten und wird umgesetzt werden?

Wegen des Qualitätsaspektes der Modularisierung (siehe Abschnitt 3.4 „Codequalität“) wurde entschieden, aus FZ2.2 Ansatz 2 umzusetzen.

FZ3.1: Welche der manuellen Schritte mit der Arduino-IDE oder deren Teilen entsprechen welchen Befehlen der Arduino-CLI?

Die Übersichtstabellen 7.2, 7.3 und 7.4 aus Abschnitt 7.2.1 „Manuelle Schritte mit der Arduino-IDE

und entsprechende Schritte mit der Arduino-CLI“ beantworten diese Frage in einer übersichtlichen Form. Alles, was mit der IDE durchgeführt werden kann, kann auch mit Befehlen der CLI durchgeführt werden und es sind auch weitere Fähigkeiten vorhanden. Ein Beispiel ist das Kompilieren einer .ino-Datei mit dem Speichern von dessen Ergebnissen in einem gewünschten Ordner.

FZ3.2: Welche Aufrufe müssen bei der Arduino-CLI zusätzlich gemacht werden?

Für die Nutzung selbst muss als Vorbereitung in der Konsole der Pfad temporär eingestellt werden, falls kein manuelles erfolgreiches Nachtragen des Pfades der Arduino-CLI in den Systemdateien durchgeführt wurde. Dies erfolgt durch den Befehl `export PATH=[Pfad zu arduinocli-Datei]:$PATH`. Sein Effekt beschränkt sich nur auf die Terminalinstanz, in der er durchgeführt wurde und solange diese aktiv ist.

Beim Installieren von Bibliotheken aus dem Internet ist der Befehl `arduino-cli lib update-index` empfohlen, damit die Daten für das Herunterladen von den verschiedenen Adressen aktuell sind.

FZ3.3: Können durch die Arduino-CLI dieselben Ergebnisse wie mit der Arduino-IDE erzielt werden?

Ja, denn verschiedene Testprogramme wie „Blink“ und deren Variationen wiesen beim Hochladen per Arduino-IDE und per Arduino-CLI in ihrem Verhalten keine erkennbaren Unterschiede auf.

FZ3.4: Wie kann „Sofdcar-HAL“ per Arduino-CLI installiert werden?

Für das Installieren der Bibliothek „Sofdcar-HAL“ aus dem Archiv „Sofdcar-HAL.zip“ muss zunächst per `arduino-cli config set library.enable_unsafe_install true` das Installieren von Bibliotheken aus .zip-Archiven erlaubt werden, damit dies selbst per `arduino-cli lib install --zip -path Pfad/zu/Sofdcar-HAL.zip` durchgeführt werden kann. Danach sollte `arduino-cli config set library.enable_unsafe_install false` verwendet werden, um die mögliche Sicherheitslücke wieder zu schließen.

8.2.2. Vorgehen beim Überprüfen der entwickelten Software hinsichtlich der Korrektheit ihrer Funktionsweise und ihrer Ausgaben

Die oben beschriebenen Tests wurden während der Entwicklungszeit durchgeführt, um Fehler zu finden und zu beheben sowie die entwickelte Software vollständig zu validieren.

8.2.3. Qualitätsbewertungen

In diesem Abschnitt wird die Qualität des Codes der Eclipse-Plug-In-Projekte „ArduinoCLITilizer“, „MUMLACGPPA“ und „PipelineExecution“ nach den in Abschnitt 3.4 „Codequalität“ beschriebenen Kriterien bewertet.

Danach wird beurteilt, wie weit die CI/CD-Pipeline die in Abschnitt 3.5 „CI/CD-Pipeline“ beschriebenen Eigenschaften und Fähigkeiten erfüllt.

8. Evaluation

Richtlinie	Aspekt	Erfüllungsstufe
Funktionale Eignung	Funktionale Vollständigkeit	2
	Funktionale Korrektheit	2
	Funktionale Angemessenheit	2
Kompatibilität	Koexistenz	3
	Interoperabilität	1
Verwendbarkeit	Eignungserkennbarkeit	2
	Lernbarkeit	3
	Bedienbarkeit	3
	Nutzerfehlerschutz	2
	Benutzerinterface-Ästhetik	2
	Zugänglichkeit	2
Zuverlässigkeit	Ausgereiftheit	3
	Verfügbarkeit	3
	Fehlertoleranz	2
	Wiederherstellbarkeit	2
Wartbarkeit	Modularität	3
	Wiederverwendbarkeit	2
	Analysierbarkeit	2
	Modifizierbarkeit	3
	Testbarkeit	2
Portabilität	Adaptierbarkeit	2
	Installierbarkeit	3
	Ersetzbarkeit	-

Tabelle 8.1.: Übersicht der Richtlinien und der Aspekte nach ISO-25010[Anw] sowie der jeweiligen Erfüllungsstufe in einer Tabelle

8.2.3.1 Funktionale Eignung

Funktionale Vollständigkeit: Erfüllungsstufe: 2

Die Modelle des Overtaking-Cars-Projektes werden automatisch entsprechend exportiert, transformiert, nachbearbeitet und hochgeladen. Insgesamt funktioniert also mit dem automatischen Arbeitsablauf von den Modellen ein Großteil der Funktionen und nur ein kleiner Teil nicht.

Funktionale Korrektheit: Erfüllungsstufe: 2

Manuellen Überprüfungen zufolge produzieren die Nachbearbeitungsschritte (erkennbar durch das „PostProcessing“ am Anfang ihrer Namen) korrekten Code. Dieser lässt sich mit der Arduino-CLI oder der Arduino-IDE kompilieren und auf die Boards übertragen. Aufgrund von Beschränkungen seitens des Plug-In-Framework von Eclipse können von den Schritten DialogMessage und SelectableTextWindow keine Fenster erzeugt werden. Deren Nachrichten werden improvisiert auf der

Konsole der Eclipse-Workbench-Instanz, von der aus die Plug-Ins gestartet wurden, angezeigt. Dies ist im Vergleich zu den anderen funktionierenden Funktionen und Abläufen nur ein kleiner nicht funktionierender Teil der Gesamtmenge.

Funktionale Angemessenheit: Erfüllungsstufe: 2

Die Änderungen und Eclipse-Plug-Ins zusammen können fast alle bisher genannten Anforderungen erfüllen. Die Nachbearbeitungsschritte könnten nach größeren Modelländerungen nicht mehr zum generierten Code passen. Dies ist jedoch praktisch gesehen normal und wird deshalb nur als ein kleines Problem gesehen. Aufgrund von Eclipse können von den Schritten `DialogMessage` und `SelectableTextWindow` keine Fenster erzeugt werden. Deren Nachrichten werden improvisiert auf der Konsole der Workbench-Instanz, von der aus die Plug-Ins gestartet wurden, angezeigt. Dies stellt jedoch bei der Anwendung nur ein kleines Problem dar. Insgesamt sind die Änderungen und Eclipse-Plug-Ins zusammen bis auf die beiden Problemstellen komplett für das Arbeiten mit MUML, den automatisierten Abläufen, den Roboterautos und dem Anwendungsszenario (siehe Kapitel 4 „Anwendungsszenario“) geeignet.

8.2.3.2 Kompatibilität

Koexistenz: Erfüllungsstufe: 3

Weder die Arduino-CLI noch die Arduino-IDE haben einen negativen Einfluss auf das System. Die im Rahmen dieser Ausarbeitung erstellten oder modifizierten Projekte haben keinen negativen Einfluss auf andere Plug-Ins oder Software. Es ist also die Koexistenz mit anderer Hardware vollständig gegeben.

Interoperabilität: Erfüllungsstufe: 1

Von dem Code, der im Rahmen dieser Masterarbeit geschrieben wurde, wird das „ArduinoCLIUtilizer“-Eclipse-Plug-In effizient und effektiv von den Eclipse-Plug-Ins „MUMLACGPPA“ und (indirekt) „PipelineExecutor“ verwendet. „MUMLACGPPA“ wird ebenfalls effizient und effektiv von „PipelineExecutor“ verwendet. Aufrufe von anderen Eclipse-Plug-ins sind auch möglich. Es konnten jedoch keine Möglichkeiten gefunden werden, Eclipse-Plug-Ins von außen zu starten. Hierfür wären also kleine Anpassungen notwendig. Alle Eclipse-Plug-Ins, die im Rahmen dieser Ausarbeitung erstellt wurden, sind miteinander und mit Eclipse interoperabel, jedoch nicht mit anderer Software, was eine geringe Interoperabilität bedeutet.

8.2.3.3 Verwendbarkeit

Eignungserkennbarkeit: Erfüllungsstufe: 2

Es ist hauptsächlich durch das Lesen der Ausarbeitung und der Readme-Dateien erkennbar, dass die erstellten Eclipse-Plug-Ins für das Generieren von hochladbarem kompliziertem Code für AMKs und dem Installieren von diesem auf arduinobasierte Roboterautos ausgehend von MUML-Modellen

8. Evaluation

konzipiert sind. Zusätzlich sind die Resultate mancher Post-Processing-Abläufe nur für das Anwendungsszenario (siehe Kapitel 4 „Anwendungsszenario“) oder ausreichend ähnlichen Projekten korrekt. Teile der Pipeline und ein Großteil des ArduinoCLUtilizer-Plug-Ins können aber auch außerhalb davon genutzt werden. Einerseits würden andere Personen diese Plug-Ins wegen des Kontextes der MDSE mit MUML für die Roboterautos der Universität Stuttgart wahrscheinlich verwerfen oder nicht finden, obwohl sie diese möglicherweise brauchen oder nutzen könnten. Andererseits macht die Begrenzung auf Eclipse Neon das Arbeitsfeld dieser Ausarbeitung zu einer Nische: „Eclipse C++ Tools for Arduino 2.0“ und die in Abschnitt 3.2.2 „TEA“ vorgestellte Automatisierungsmöglichkeit TEA würden unter neueren Eclipse-Editionen wahrscheinlich funktionieren und die im Rahmen dieser Ausarbeitung erstellten Plug-Ins obsolet machen. Personen, die mit MUML und AMKs oder Ähnlichem arbeiten, würden jedoch wahrscheinlich leicht bei einer Suche diese Ausarbeitung sowie ihre Ergebnisse finden und deren Eignung einschätzen können, was für eine hohe Eignungserkennbarkeit in dem relevanten Personenkreis spricht.

Lernbarkeit: Erfüllungsstufe: 3

Die Kontextmenüeinträge tauchen nur dann auf, wenn auf die jeweils betreffenden Dateien geklickt wird. Bei .zip-Dateien, die keine Bibliothek für die ArduinoCLI enthalten, bricht diese den Vorgang ab. Generierbare Beispiele und Standardeinstellungen helfen beim schnellen Nutzen. Die Namen der Klassen und Variablen wurden so gewählt, dass sie zu dem jeweiligen Verwendungszweck passen oder diesen kurz beschreiben. Es finden auch Erklärungen hinsichtlich den geplanten Schritten mit Fenstern statt. In vielen Fällen kann die programmierte Software Fehler selbstständig erkennen und zeigt beim Abbruch eine Erklärung, was z.B. fehlerhaft eingetragen ist, an. Man kann also ihre Nutzung leicht erlernen.

Bedienbarkeit: Erfüllungsstufe: 3

Dieselben Faktoren wie bei der Lernbarkeit gelten auch für die Bedienbarkeit. Das Wechseln zu der die Plug-Ins startenden Workbench-Instanz kann als Bedienungshürde vernachlässigt werden. Die Bedienbarkeit ist folglich komplett gegeben.

Nutzerfehlerschutz: Erfüllungsstufe: 2

Die Kontextmenüeinträge tauchen nur dann auf, wenn auf die jeweils betreffenden Dateien geklickt wird. Bei .zip-Dateien, die keine Bibliothek für die ArduinoCLI enthalten, bricht diese den Vorgang ab. Es konnten bei der Pipeline viele Sicherheitsvorkehrungen konzipiert und umgesetzt werden. Es werden der Datenfluss und die Typenkorrektheit über die Variablen überprüft. Folglich kann z.B. in einem Upload-Schritt kein Ordnerpfad für den Portparameter portAddress verwendet werden, ohne bereits vor dem Starten der Pipeline auf den Fehler hingewiesen zu werden. Zusätzlich wird die Pipeline nicht gestartet, falls Fehler gefunden wurden. Jedoch können z.B. am Anfang oder bei direkt ausgefüllten Parametern falsche Werte bzw. Einstellungen eingetragen werden. Bei solchen Fällen würden in der Regel die entsprechenden Abläufe scheitern und so der Pipelineablauf abgebrochen werden. Gegen falsche oder schädigende Pfade (z.B. solche, die in ein anderes Projekt hineinzeigen) existieren keine Sicherheitsvorkehrungen seitens der pipelinebasierten Automatisierung. Jedoch

müssten für Pfade außerhalb des Projekts absolute Pfade angegeben werden und relative Pfade würden nur zu Orten innerhalb des Projekts aufgelöst werden. Schäden können praktisch nur entstehen, wenn z.B. der Pipelineschritt DeleteFolder böswillig genutzt wird, oder wenn leichtsinnig mit den Pfaden gearbeitet wird. Fehlkonfigurierte Roboterautos sollten wegen ihrer geringen Größe und Antriebsstärke keine oder keine nennenswerten Schäden verursachen können. In Anbetracht der jeweiligen Bedingungen für das Auftreten von Schäden sowie deren entsprechenden Auswirkungen kann man also bei diesen Plug-Ins nur von wenigen Problemen bzw. Zwischenfällen ausgehen.

Benutzerinterface-Ästhetik: Erfüllungsstufe: 2

Es erfolgen keine Arbeitsschritte mit der Kommandozeile oder anderen Programmen, sondern innerhalb von Eclipse. Die Aufrufe erfolgen jeweils per Auswählen des entsprechenden Eintrages in dem Kontextmenü und in der Liste der Exportmöglichkeiten. Das Konfigurieren der Pipeline erfolgt textuell in einer Konfigurationsdatei, ist aber gut lesbar und schreibbar. Infolge von Beschränkungen seitens der Eclipse-IDE können von Exportprozessen aus keine Fenster angezeigt werden. Als Improvisation werden diese Meldungen in der Konsole der Workbench-Instanz, von der aus die Plug-Ins gestartet wurden, ausgegeben. Dies ist für Entwickler und Entwicklerinnen mit wenig Erfahrung beim Umgang mit der Eclipse-IDE unintuitiv und gewöhnungsbedürftig. Die Konsolenausgaben werden jedoch nur als ein kleines Problem bei der Nutzung gesehen, weil der jeweils geplante Inhalt in diesen dennoch gut lesbar ist. Bei der Nutzung überwiegen die gut bedienbaren anderen Teile.

Zugänglichkeit: Erfüllungsstufe: 2

Auch diejenigen, die wenig Erfahrung mit arduinobasierten Mikrocontrollern haben, können mit „ArduinoCLIUtilizer“ und den auf AMKs orientierten Aspekten des Pipelinesystems gut arbeiten, weil sehr viel von den internen Abläufen übernommen wird oder Informationen leicht aufgerufen werden können. Je nach Änderung an den Modellen und den notwendigen Anpassungen der Nachbearbeitungsschritte wird jedoch grundlegendes Programmierverständnis notwendig sein, um die Post-Processing-Sequenz anzupassen. Infolge von Beschränkungen seitens der Eclipse-IDE können von Exportprozessen aus keine Fenster angezeigt werden. Wie oben erwähnt werden diese Meldungen als Improvisation in der Konsole der Workbench-Instanz, von der aus die Plug-Ins gestartet wurden, ausgegeben. Dies ist für Personal mit wenig Erfahrung beim Umgang mit der Eclipse-IDE verwirrend und gewöhnungsbedürftig. Gleich der Benutzerinterface-Ästhetik werden die Konsolenausgaben nur als ein kleines Problem bei der Nutzung gesehen, weil der jeweils geplante Inhalt in diesen dennoch gut lesbar ist. Bei der Zugänglichkeit überwiegen die gut bedienbaren anderen Teile.

8.2.3.4 Zuverlässigkeit

Ausgereiftheit: Erfüllungsstufe: 3

Die Plug-Ins liefern bei der normalen Nutzung korrekte Ergebnisse, wie Tests und manuelle Vergleiche gezeigt haben.

8. Evaluation

Verfügbarkeit: Erfüllungsstufe: 3

Ein Schnellstart ist durch das Erstellen der Konfigurationsdateien mit den Standard-Einstellungen möglich. Das Starten erfolgt durch das Auswählen der jeweiligen Menüeinträge. Lediglich das Starten einer neuen Workbench-Instanz mit den Erweiterungen für die MUML-Toolsuite ist als Vorbereitung erforderlich (siehe Abschnitt 2.5 „Eclipse-Plug-Ins“), aber dieser Moment ist vernachlässigbar im Vergleich zu der gesamten Arbeitszeit.

Fehlertoleranz: Erfüllungsstufe: 2

Die Pipeline ist von der Eclipse-IDE und der MUML-Toolsuite abhängig und würde ohne diesen beiden nicht mehr funktionieren. Jedoch sind dies in Anbetracht der Ziele dieser Ausarbeitung die gegebenen Rahmenbedingungen. Es funktioniert nicht, wenn wichtige Software, wie z.B. eines der MUML-Plug-Ins, fehlt. Ein Teil der Schritte erfordert die MUML-Plug-Ins oder die Arduino-CLI und hängt von deren Durchführung und Korrektheit ab. Bei Durchführungsabbrüchen oder bei als unerwünscht erkannten Rückmeldungen beendet die Pipeline ihre Ausführung und meldet die Probleme. Für die normale Nutzung sollte dies ausreichen, aber bei unvorhergesehenen Fehlern, die während der Tests nicht beobachtet werden konnten, können die für diese Ausarbeitung geschriebenen Plug-Ins möglicherweise keinen gut lesbaren Bericht generieren.

Wiederherstellbarkeit: Erfüllungsstufe: 2

Bei standardmäßiger Nutzung werden die zwei Ordner „generated-files“ und „deployable-files“ generiert. Es werden keine MUML-Modelle usw. gelöscht oder geändert. Folglich würden im schlimmsten Fall lediglich die Zwischenergebnisse geschädigt werden oder verloren gehen, aber ein Neustart der Pipeline würde dies leicht beheben. In weniger schlimmen Fällen kann die Pipeline zwar nicht direkt mit dem gescheiterten Schritt fortfahren, aber die resultierenden Probleme können leicht behoben werden. Sollte beispielsweise nach der Generierung der in diesem Zustand unvollständigen Codedateien das Post-Processing unterbrochen werden, so kann beispielsweise durch Löschen des Ordners „deployable-files“ und dem anschließenden Starten der Post-Processing-Sequenz das Post-Processing für den kompilierbaren Code neu gestartet werden. Insgesamt wäre also eine Pipeline-Fehlfunktion nicht für die Projektarbeit verhängnisvoll, sondern die Wiederherstellung der generierten Dateien würde leicht und schnell erfolgen.

8.2.3.5 Wartbarkeit

Modularität: Erfüllungsstufe: 3

Die Aufgaben wurden auf verschiedene Module und Klassen aufgeteilt. Gleichzeitig bestehen die Abhängigkeiten zwischen diesen nur soweit, wie es für die Durchführung der Aufgaben erforderlich ist. Solange keine Interfaces oder Statusangaben geändert werden, können Änderungen an einer Komponente durchgeführt werden, ohne dass dies große Effekte auf das gesamte System hat.

Wiederverwendbarkeit: Erfüllungsstufe: 2

Abgesehen vom Plug-In-Projekt „ProjectFolderPathStoragePlugIn“, das nur den Pfad des Projektordners speichert, gilt folgendes:

- Das Plug-In-Projekt „ArduinoCLIUtilizer“ kann allein verwendet werden oder sein Inhalt kann von anderen Projekten aufgerufen werden.
- Aus dem Projekt „MUMLACGPPA“ kann prinzipiell mit relativ wenigen Handgriffen der Code für das Anwendungszenario entfernt werden (z.B. durch das Kopieren der nutzbaren Schritte und dem Entfernen der Pakete mit „mumlpostprocessingandarduincli“ im Namen). So kann dieses als Basis für Pipelines für andere Anwendungen umfunktioniert werden. Alternativ kann in dem Konstruktor für PipelineSettingsReader eine andere Kindklasse von PipelineStepDictionary, das die Nutzung von anderen Schritten beinhaltet, eingetragen werden.
- PipeLineExecutionAsExport und ggf. die anderen Klassen aus „de.ust.pipelineexecution.ui.exports“ können mit ein paar Änderungen wie z.B. dem Entfernen der Improvisationen für ContainerTransformation per doExecuteContainerTransformationPart(final EObject [] sourceElementsSystemAllocation, ContainerTransformation step, IProgressMonitor progressMonitor) für andere Anwendungszwecke wiederverwendet werden.

Die Änderungen für das Wiederverwenden sollten also voraussichtlich relativ leicht und zügig durchgeführt werden können.

Analysierbarkeit: Erfüllungsstufe: 2

Anhand der UML-Diagramme (siehe Ergänzungsmaterial A „GitHub-Repository mit Sourcecode“), den Readmes und den Beschreibungen aus Kapitel 7.2 „Integration der ArduinoCLI“ und Ergänzungsmaterialien E „Ergänzungsmaterial für Eclipse-Plug-In-Projekt ,ArduinoCLIUtilizer“, F „Ergänzungsmaterial für Eclipse-Plug-In-Projekt ,MUMLACGPPA“ und G „Ergänzungsmaterial für Eclipse-Plug-In-Projekt ,PipelineExecution“ kann relativ gut abgeschätzt werden, wie die beabsichtigten Änderungen weitere Änderungen an den anderen Modulen und Komponenten verursachen würden. Jedoch fehlen weitere Diagramme, z.B. für die große Menge der Schritte, das Gesamtsystem oder aber auch für den Datenfluss. Dies sollte jedoch schätzungsweise nur eine relativ geringe Verzögerung beim Verstehen des Codes verursachen. Insgesamt sind die Hürden für das Analysieren also gering.

Modifizierbarkeit: Erfüllungsstufe: 3

Durch die Aufgabenteilung und Modularisierung zwischen den Modulen und Klassen sollten Änderungen an einer Stelle keine schädlichen Effekte auf andere Stellen haben.

8. Evaluation

Testbarkeit: Erfüllungsstufe: 2

Es wurden für die Funktionalität des Pipelinesystems verschiedene JUnit-Tests erstellt, die fast alle Aspekte davon abdecken. Nach Änderungen am System sollte also das Nutzen dieser automatischen Tests mögliche Fehler schnell und relativ genau aufzeigen. Bei den Pipelineschritten sind jedoch manuelle Tests auf deren Auswirkungen erforderlich. Die Modularität sollte den Aufwand hierfür jedoch gering halten, also wird diese Schwäche nicht als ein großer Faktor eingeschätzt.

Es ist noch kein automatisches Testen der MUML-Modelle durch den Export von diesen nach Matlab (siehe Heinemann et al. [HRB+14]) umgesetzt. Dasselbe gilt für das automatisierte Testen der AMKs oder Roboterautos. Beim Kompilieren per Pipelineschritt `compile` zeigen sich früh Generations- oder Post-Processing-Fehler im Code. Außerhalb davon kann das Verhalten der Roboterautos nur durch die physische Durchführung des produzierten Codes bzw. der produzierten Sketches getestet werden. Dieses hängt hauptsächlich von den MUML-Modellen und den Werkzeugen ab, die wiederum nicht Teil dieser Ausarbeitung sind. Seitens der im Rahmen dieser Ausarbeitung vorgenommenen Anpassungen oder erstellten Erweiterungen muss in diesem Aspekt also „ArduinoCLIUtilizer“ nach Änderungen manuell überprüft werden, wobei sein relativ einfacher Aufbau nur zu einem geringen Testaufwand führen sollte.

Insgesamt ist die Testbarkeit also relativ gut oder leicht.

8.2.3.6 Portabilität

Adaptierbarkeit: Erfüllungsstufe: 2

Die verschiedenen entwickelten Eclipse-Plug-Ins sind als Java-Sourcecode, genauer gesagt, als Eclipse-Plug-in-Projekte, gegeben. Die Code-Teile für die Aufrufbarkeit auf der Eclipse-GUI sind von Eclipse abhängig, können aber prinzipiell leicht ersetzt werden. Die funktionalen Code-Teile sind abgesehen von denen, die mit Teilen der MUML-toolsuite arbeiten, nur gering von der Eclipse-Umgebung abhängig. Somit könnten die entwickelten Plug-Ins leicht auf eine andere Umgebung oder einen anderen Nutzungskontext angepasst werden. Die Portierbarkeit der Code-Teile, die mit Teilen der MUML-toolsuite arbeiten, hängt von der MUML-toolsuite ab, aber dies betrifft nur einen geringen Teil der Pipelineschritte.

Die Arduino-CLI ist für viele verschiedene Systeme erhältlich [Ardb] und stellt kein nennenswertes Hindernis beim Portieren dar.

Insgesamt basiert ein Großteil der Fähigkeiten, z.B. das Kopieren oder Löschen von Ordnern, nicht auf der MUML-toolsuite und somit kann die CI/CD-Pipeline auch ohne einen großen Qualitätsverlust auf eine Nutzung ohne dieser Toolsuite angepasst werden.

Installierbarkeit: Erfüllungsstufe: 3

Für die Installation werden die erstellten Eclipse-Plug-In-Projekte in den Workspace der startenden Workbench-Instanz importiert. Die Deinstallation der entwickelten Plug-Ins erfolgt durch das Entfernen der Plug-In-Projekte aus dem Workspace. Die Arduino-CLI kann auch z.B. über die angebotenen Installationsprogramme oder über das Herunterladen und Platzieren als Datei

leicht installiert werden. Ihre Deinstallation kann auch je nach gewähltem Installationsweg über die typischen Deinstallationsabläufe oder über das Löschen der heruntergeladenen Datei leicht durchgeführt werden.

Ersetzbarkeit: Erfüllungsstufe: -

Dies ist wegen mangelnder Erfahrung und fehlenden bekannten Vergleichsmöglichkeiten unklar und somit wurde hierzu keine Bewertung vorgenommen.

8.2.4. Qualität der CI/CD-Pipeline

Eigenschaft oder Fähigkeit	Erfüllungsstufe
Geschwindigkeit	3
Konsistenz	3
Enge Versionskontrolle	1
Automatisierung	2
Integrierte Rückmeldungsschleifen	2

Tabelle 8.2.: Übersicht der Eigenschaften und Fähigkeiten nach Bigelow[Big] sowie der jeweiligen Erfüllungsstufe in einer Tabelle

8.2.4.1 Geschwindigkeit: Erfüllungsstufe: 3

Bei manuellen Tests, deren Zeitmessung mit dem ersten Klick auf „Export...“ starteten, hat sich gezeigt, dass die Durchführung der Pipeline mit den standardmäßigen Pipelineeinstellungen deutlich schneller erfolgt als das Ausführen der entsprechenden Arbeitsschritte von Hand. Bei der ersten Ausführung, in der die Software noch nicht im Arbeitsspeicher vorliegt, wird eine Ausführungszeit von ca. 50 Sekunden benötigt, aber in den weiteren Ausführungen lediglich ca. 38 Sekunden. In 50 Sekunden können nur die Arbeitsschritte für die Container-Transformation, die Generation des Container-Codes und des Komponenten-Codes sowie die ersten Post-Processing-Schritte erfolgen. Das manuelle Durchführen des Post-Processings erfordert durch das Kopieren von Dateien, Pfad-Korrekturen, nachgetragenen Funktionsaufrufen und dem Eintragen von Einstellungen für z.B. den WLAN-Zugang auch mit Routine über eine Minute.

8.2.4.2 Konsistenz: Erfüllungsstufe: 3

Der einzige variable Faktor außerhalb der Pipelineeinstellungen und MUML-Modellen können die Portadressen für die angeschlossenen AMKs sein. Dieser Teil ist also je nach dem Verbinden der AMKs mit dem PC über die verschiedenen USB-Buchsen für jede Nutzung durchaus etwas anders und somit nicht konsistent. Aber hierfür ist ein kompensierender Ablauf konzipiert worden: Über den Pipelineschritt `LookupBoardBySerialNumber` wird zu dem jeweiligen gesuchten AMK anhand von dessen Seriennummer die zu dem Zeitpunkt verwendete Portadresse herausgesucht und in einer Variable gespeichert. Diese Information wird dann von dem entsprechenden `Upload`-Schritt

8. Evaluation

verwendet und somit der Upload zielgenau durchgeführt.

Manuelle Überprüfungen haben gezeigt, dass in manchen Dateien wie z.B. unter „fastCarDriver-ECU“ in „ECU_Identifier.h“ die Reihenfolge der Definitionen der ComponentInstances-Identifier variiert (siehe Ergänzungsmaterialabschnitt F.2 „Beispiel der Reihenfolgeunterschiede“), aber die anderen generierten Dateien aus demselben Pipelinedurchlauf sind auf die entsprechenden Unterschiede eingestellt und die folgenden Arbeitsschritte werden davon nicht beeinflusst. Somit können die Reihenfolgeunterschiede, die durch die offenbar nichtdeterministischen MUML-Werkzeuge produziert werden, ignoriert werden.

8.2.4.3 Enge Versionskontrolle: Erfüllungsstufe: 1

Aufrufe von Git sind über AutoGitCommitAllAndPushCommand oder TerminalCommand möglich. Andere Systeme für die Versionsverwaltung erfordern jedoch das Schreiben eines Skriptes, das das Hochladen durchführt und per TerminalCommand gestartet wird. Die Versionskontrolle ist also praktisch gesehen nur geringfügig erfüllt, aber es gibt hierfür verschiedene Systeme, z.B. „GitHub“, „GitLab“, „Beanstalk“ oder auch „Google Cloud Source Repositories“, wie einer zu der Beschreibung von Lohrey[Loh] beigelegten Liste entnommen werden kann. Die verschiedenen Systeme haben ihre eigenen Sicherheitsvorkehrungen, deren Unterstützung nicht im Rahmen dieser Ausarbeitung umgesetzt werden konnte.

8.2.4.4 Automatisierung: Erfüllungsstufe: 2

Es ist hauptsächlich nur das Konfigurieren der Schritte manuell. Danach ist der Arbeitsablauf von den Modelltransformationen bis zum Hochladen auf die AMKs durch die so konfigurierte Pipeline automatisiert, wie in den Kapiteln 5 „Analyse“ und 7.3 „Vollständige Automatisierung per Pipeline“ beschrieben wird. Für andere Anwendungsziele können durch den Pipelineschritt TerminalCommand sogar andere Skripte und Programme in die Automatisierung eingebaut werden. Aufgrund der Improvisation mit dem Export tauchen je nach Pipeline-Schritten noch die Auswahl-Fenster auf. Dieses Problem bzw. dieser Workaround ist jedoch im Vergleich zu den vorhandenen automatisierten Abläufen nur ein geringes Problem.

8.2.4.5 Integrierte Rückmeldungsschleifen: Erfüllungsstufe: 2

Feedback muss in Form der Schritte OnlyContinueIfFilledElseAbort, PopupWindowMessage oder SelectableTextWindow erfolgen. Hiervon dient OnlyContinueIfFilledElseAbort zum bedingten Abbrechen. Das Pipelinesystem kann in Folge von Beschränkungen durch die Eclipse-IDE außer dem Melden von Exceptions, die zum Abbruch geführt haben, nicht selbstständig zwischendurch Rückmeldungen wie das Anzeigen von Fenstern mit Meldungen durchführen, wie in Abschnitt „7.3.1.2 Blockiertes Anzeigen von Nachrichten- und Text-Fenstern“ beschrieben wird. Stattdessen werden deren Informationen auf die Konsole der startenden Workbench-Instanz umgeleitet. Nur für OnlyContinueIfFilledElseAbort konnte wegen seines Zweckes als mögliche Abbruchstelle der Pipelinedurchführung ein Workaround integriert werden, der seine Meldung anzeigt. In Anbetracht des manuellen Wechsels zwischen den Workbench-Instanzen für das Nachsehen von Informationen wird dieser Aspekt nicht als komplett, sondern als größtenteils erfüllt angesehen.

8.2.5. Das Anwendungsszenario als Praxistest und Vergleichspunkt

Zu den verschiedenen Schritten des Praxistests wurde folgendes beobachtet:

Zu Schritt 1: Die Pipeline konnte komplett durchgeführt werden und schloss erfolgreich ab. Ein Vergleich des generierten Codes mit dem Code von Stürner [Codb] hat gezeigt, dass nur die gewollten oder erwarteten Unterschiede aufgetreten sind. Dasselbe Ergebnis ist bei einem Vergleich zwischen dem generierten Code und dem Code [Reie], der für die neueren Roboterautos und der Verwendung der Bibliothek „Sofdcar-HAL“ manuell nachbearbeitet wurde, der Fall.

Zu Schritt 2: Beim Hochladen der Sketches auf die entsprechenden AMKs kam es zu keinen Zwischenfällen.

Zu Schritt 3: Die Roboterautos stellten die Verbindung mit dem WLAN-Netzwerk her und der Verbindungsauflauf zwischen den Koordinator-AMKs und dem Message Queuing Telemetry Transport (MQTT)-Server fand auch erfolgreich statt.

Beim Fahren wiesen die Roboterautos den gewünschten Geschwindigkeitsunterschied auf und sie beförderten erfolgreich die Fahrbahn, in der sie starteten. Es hat sich jedoch gezeigt, dass die Kommunikation, die für das Starten des Überholmanövers erforderlich ist, nur schwer ausgelöst werden kann und somit in Tests praktisch unzuverlässig stattfindet.

Recherchen zufolge liegt der Grund hierfür in einem nicht verifizierten Teil von Stürners Ergebnissen, da er das Koordinationsverhalten nicht getestet hat. Analysen des Pipelinesystems und der verwendeten Konfiguration für den automatisierten Build-Prozess deuten darauf hin, dass die im Rahmen dieser Ausarbeitung geschriebene Software ihre Funktionen korrekt erfüllt und somit nicht für die Kommunikationsprobleme verantwortlich ist.

8.3. Diskussion

Das Ziel dieser Ausarbeitung ist das Herstellen der Unterstützung der aktuellen Version der verwendeten Roboterautos sowie die Automatisierung des Build-Prozesses, der in seiner manuellen Ausführung viel Zeit beansprucht hat und durch das hohe Fehlerrisiko leicht scheitern konnte.

Alle Forschungsziele konnten beantwortet werden. Deren Antworten wiederum stellten für das Erfüllen der Ziele dieser Ausarbeitung sowie für das Planen und Programmieren der dabei entwickelten Software wichtige Informationen und Grundlagen bereit.

So wurden für das Unterstützen der aktuellen Generation der verwendeten Arduino-Roboterautos die notwendigen Änderungen an den MUML-Modellen und dem Post-Processing-Ablauf erforscht und durchgeführt. Zusätzlich wurden kleine Verbesserungen, wie eine verbesserte Kommunikation innerhalb der Roboterautos zwischen den jeweiligen AMKs und eine verbesserte Ausrichtung des hinteren Abstandssensors, durchgeführt. Der interne Workaround der seriellen Kommunikation trotz der in den Modellen angegebenen I2C-Methode ist eine notwendige, aber bei der Nutzung widersprüchliche, Umsetzung.

Für die Automatisierung der MUML-Werkzeuge wurde analysiert, welche Teile von dem Arbeitsablauf von der Nutzerschaft manuell durchgeführt wurden und die relevanten involvierten Softwarekomponenten und Daten wurden genannt. Die Handgriffe, die auf der Arduino-IDE durchgeführt wurden, wurden auch festgehalten.

8. Evaluation

Auf Basis der Antworten bzw. deren Informationen wurde das System für die Automatisierung per CI/CD-Pipeline entworfen.

Das sodann umgesetzte Pipelinesystem hat fast alle Anforderungen gut oder vollständig erfüllt, was sich in fast allen Aspekten durch das Erreichen der Erfüllungsstufe 2 oder 3 widerspiegelt. Die einzige Ausnahme ist die Versionskontrolle, zu der verschiedene Versionskontrollsysteme und mögliche Sicherheitsvorkehrungen existieren. Somit sollte zu diesem Aspekt entsprechend zu dem gewählten System ein passendes Skript geschrieben und verwendet werden. Abgesehen von diesem Aspekt kann die vollständige Automatisierung bzw. die Anwendung der CI/CD-Entwicklungspraxis von dem Pipelinesystem jedoch sehr gut durchgeführt werden, weil alles von den MUML-Werkzeugen bis zum Hochladen abgedeckt ist. Es wurden für eine bessere Flexibilität und für den Langzeitnutzen der Pipeline z.B. für das Post-Processing weitere Schritte konzipiert, um leicht weitere Post-Processing-Maßnahmen zu ergänzen. So können sogar über TerminalCommand andere Programme aufgerufen werden.

Bei den Qualitätskriterien nach ISO-25010[Anw] erreichen die erstellten Eclipse-Plug-Ins „ArduinoCLUtilizer“, „MUMLACGPPA“ und „PipelineExecution“ immer die Erfüllungsstufen 2 oder 3, weisen also eine gute Qualität auf. Dies wurde erreicht, indem während der jeweiligen Umsetzung auf eine gute Architektur und gute Nutzbarkeit der Software geachtet wurde. So kann die Nutzerschaft sich durch die generierbaren Beispiele und Konfigurationsdateien leicht und schnell in die Nutzung des Pipelinesystems einfinden. Die verschiedenen Fehlermeldungen mit deren Erklärungen helfen beim Finden und Verstehen von Fehlern. In allen bekannten Problemsituationen wird keine schwierige Fehlergrundsuche auftreten. Falls die vorhandenen Pipelineschritte für Anpassungen nicht ausreichen oder aus anderen Gründen Änderungen am Sourcecode durchgeführt werden müssen, sollten Modifikationen durch die hohe Codequalität in einer angemessenen Zeitspanne gut umsetzbar sein. Folglich sollte z.B. das Entfernen der Bindungen an die MUML-Werkzeuge gut bewerkstelligt werden können.

Der am Ende hergestellte nutzungsbereite Arduinocode bzw. das Code-Produkt ist auch korrekt, wie Tests und Vergleiche gezeigt haben. Das Anwendungsszenario „kooperatives Überholen“ hat als Anwendungsbeispiel und Vergleichspunkt mit Stürners Ergebnissen gezeigt, dass das Code-Produkt der im Rahmen dieser Ausarbeitung erstellten Software in keinem Aspekt schlechter abschneidet als der Code, der von Stürners erarbeitetem Arbeitsablauf allein produziert wird.

Durch die Automatisierung wird der gesamte Arbeitsablauf in weniger als einer Minute durchgeführt. Bei der manuellen Durchführung aller Arbeitsschritte kann man in dieser Zeit nur die MUML-Werkzeuge und die ersten Post-Processing-Schritte umsetzen. Alles darauf Folgende würde auch bei Routine mehrere Minuten beanspruchen. Gleichzeitig wird durch die Automatisierung die menschliche Fehlerrate vermieden.

Insgesamt ist die erreichte Automatisierungssoftware ein großer Zugewinn. Jedoch ist die Nutzung als Export-Operation anfangs ungewohnt und die Ausgabe des Textes von Nachrichten und Textfenstern auf der Konsole der startenden Workbench-Instanz ein gewöhnungsbedürftiger Workaround. Alle anderen Qualitätsaspekte werden jedoch gut erfüllt.

8.4. Gefahren für die Gültigkeit

In diesem Abschnitt werden die verschiedenen Gefahren für die Gültigkeit dieser Ausarbeitung und ihrer Ergebnisse beschrieben.

Bei dem Anwendungsszenario tritt dieselbe Unsicherheit, die Stürner in seiner Gefahrenanalyse erwähnt hat[Stü22], auf. Auch wurde der Entwurf der CI/CD-Pipeline sehr an dem Anwendungsszenario bzw. dessen Arbeitsschritten orientiert, weshalb das entwickelte Pipelinesystem ggf. auch nicht genau zu einem Anwendungsfall aus der realen Welt passen könnte. Hierbei können jedoch die Teile des Pipelinesystems, die für dessen Flexibilität und langer Nützlichkeit sorgen sollen, entgegenwirken.

Zusätzlich können die selben Unsicherheiten hinsichtlich der Gültigkeit der erstellten Software, die in Stürners Ausarbeitung erwähnt werden, auch hier auftreten, weil es teilweise darauf basiert. So wurde die dynamische Validität des Codes, z.B. eine detaillierte Überprüfung, ob das modellierte Verhalten korrekt durchgeführt wird, nicht geprüft und im Rahmen dieser Ausarbeitung konnten nur Beobachtungen von außen durchgeführt werden.

Der Autor dieser Ausarbeitung hat die Auswertungen alleine durchgeführt und die Bewertungen könnten subjektiv beeinflusst sein. Um diesem Faktor entgegenzuwirken, fand eine Diskussion mit dem Betreuer über die Bewertungsergebnisse statt.

9. Schlussfolgerung

Dieses Kapitel schließt die Ausarbeitung über die modellgetriebene Automatisierung von Code-Generation, -Integration und -Deployment von autonomen Fahrfunktionen ab. In 9.1 „Zusammenfassung“ werden Schlüsselaspekte zusammengefasst. In 9.2 „Vorteile“ wird dargelegt, wie die entwickelte Software beim Arbeiten hilft. In 9.3 „Limitierungen“ hingegen werden die Grenzen der Software dargelegt. In 9.4 „Gelernte Lektionen“ werden die Erkenntnisse, die im Laufe dieser Ausarbeitung gelernt wurden, beschrieben. Eine Reihe an zukünftigen Forschungsmöglichkeiten und Arbeitsfeldern wird in dem Abschnitt 9.5 „Zukünftige Arbeiten“ in Aussicht gestellt.

9.1. Zusammenfassung

Die beiden Hauptthemen dieser Ausarbeitung waren die Herstellung der Unterstützung der neueren Version der Roboterautos und das Ermöglichen der CI/CD-Methode per Pipeline. Für das erste Thema wurden zunächst Analysen durchgeführt und danach die entsprechenden und ausgewählten Anpassungen an den Modellen und dem Post-Processing-Ablauf vorgenommen. Zusätzlich wurden in den Roboterautos Verbesserungen wie eine zuverlässige Energieversorgung der WiFi-Module integriert.

Für die CI/CD-Pipeline wurde die Arduino-CLI in Eclipse integriert. Für die Automatisierung selbst wurde ein konfigurierbares Pipelinesystem entworfen. Bei den Umsetzungen als Eclipse-Plug-In-Projekte wurden verschiedene Workarounds benötigt, aber dennoch konnte ein gut verwendbares und flexibles Pipelinesystem implementiert werden. Durch dieses kann der Build-Prozess, also das Erstellen von verwendbarem Arduinocode ausgehend von MUML-Modellen, zuverlässig und automatisch durchgeführt werden. Des Weiteren kann das Pipelinesystem das Hochladen der generierten Codes auf die entsprechenden AMKs der Roboterautos übernehmen.

9.2. Vorteile

Alle Vorgänge, die zuvor von der Nutzerschaft durchgeführt werden mussten und dabei nicht nur Zeit kosteten, sondern auch Fehlergelegenheiten bargen, können nun von den Modelltransformationen bis zum Hochladen auf die Roboterautos automatisch durchgeführt werden.

So bestand bei jeder Nutzung der MUML-Werkzeuge Fehleregefahr z.B. beim Festlegen der Einstellungen in den Fenstern der Export-Wizards. Dies erforderte auch Zeit. Nun können für deren Automatisierung Konfigurationen in der „pipelineSettings.yaml“ eingetragen werden, sodass auch das Ausführen von einzelnen MUML-Werkzeugen zuverlässiger und etwas schneller erfolgen kann. Das Post-Processing war ein Vorgang aus vielen manuellen Schritten, der dadurch eine hohe Menge an Fehlergelegenheiten enthielt und insbesondere für ungeübte Personen viel Zeit erforderte. Durch die Konfiguration und automatisierte Ausführung der Post-Processing-Sequenz kann hierfür die

9. Schlussfolgerung

erforderliche Zeit auf Sekunden reduziert werden. Außerdem können so seitens der Nutzerschaft während des Build-Vorganges keine Fehler mehr geschehen.

Auch beim Hochladen von Code auf den entsprechenden AMKs kann die CI/CD-Pipeline helfen, indem anhand der jeweils angegebenen Dateipfade und Seriennummern zielgerichtet und zuverlässig die verschiedenen Arduinocodes auf die entsprechenden AMKs hochgeladen werden. So werden Verwirrung oder/und unerwartetes Verhalten durch z.B. Fehlauswahlen hinsichtlich der Dateien und der als Ziel angegebenen AMKs vermieden.

Die Pipeline stellt daher umso mehr eine große Hilfe für z.B. Forscherteams dar, je häufiger diese ausgehend von MUML-Modellen Code generieren lassen und mittels der Roboterautos testen.

Durch verschiedene Post-Processing-Pipelineschritte und Einfügeschritte wie `PostProcessingInsertAtIndex` können die Pipelinekonfigurationen auch auf größere Modelländerungen angepasst werden.

Die Pipeline kann hinsichtlich ihrer auszuführenden Schritte leicht rekonfiguriert werden, sodass sie für mögliche erforderliche Änderungen leicht erweitert oder angepasst werden kann. Durch das Parameterkonzept der Schritte führen diese jeweils nicht eine einzelne feste Aktion aus, sondern ihre Fähigkeiten können z.B. hinsichtlich der zu ändernden Datei flexibel angewandt werden. Die Pipeline kann durch den Pipelineschritt `TerminalCommand` auch andere Skripte und Programme automatisieren und somit prinzipiell auch für andere Zwecke als das Starten von anderen Build-Tools verwendet werden.

Hinsichtlich des Konfigurierens wird durch das Generieren von Beispielen ein leichter und schneller Start in die Nutzung der Pipeline ermöglicht. Zusätzlich sind die verschiedenen Überprüfungen der Pipelineeinstellungen durch das Pipelinesystem hilfreich beim Suchen von Fehlern, wie Typenfehler und vergessene Parameter. Wenn ein Fehler erkannt wird, so wird dieser gemeldet und die Pipeline kann nicht gestartet werden.

Bei den Roboterautos können durch die nun besser mit Energie versorgten WiFi-Module alle Experimente, die mit einer WLAN-Verbindung arbeiten, zuverlässiger durchgeführt werden. Die verbesserte Ausrichtung des hinteren Ultraschallsensors nach rechts sollte auch helfen, bei Überholszenerien das zu überholende Roboterauto bzw. den Partner zuverlässiger abzutasten und so die Erfolgserkennung besser abschätzbar zu machen.

9.3. Limitierungen

Alle im Rahmen dieser Ausarbeitung geschriebenen Eclipse-Plug-In-Projekte wurden mit dem Kontext, dass aus den MUML-Modellen für das Anwendungsszenario (siehe Abschnitt 4.1 „Das ‚kooperatives Überholen‘-Szenario“) Arduinocode für die beschriebenen Roboterautos (siehe Abschnitt 4.2 „Die Roboterauto-Zielplattform“) generiert werden soll, entwickelt. Somit wurden keine Tests für andere Roboterautoplattformen und deren jeweiliger Programmiersprache durchgeführt. Weiterhin ist es mit dem Pipelinesystem nicht möglich, komplexe Vorgänge mit z.B. unterschiedlichen Ausführungspfaden zu erstellen. Die Post-Processing-Sequenz und die eigentliche Pipelinesequenz werden beide jeweils als eine auszuführende Reihe an Befehlen gesehen, sodass `if-else`-Strukturen mit Pipelineschritten als bedingt auszuführenden Inhalt nicht möglich sind. Falsche Belegungen der Eingabeparameter können in nicht allen Fällen erkannt werden.

„ArduinoCLIUtilizer“ kann momentan auch nicht direkt über die GUI von Eclipse genutzt werden, wenn mehr als ein angestecktes Board registriert wird. So kann also PC-Peripherie, deren

Hardware von der Arduino-CLI erkannt wird, das Arbeiten mit Funktionen von „ArduinoCLUtilizer“ empfindlich stören und die interne Programmierung der PC-Peripherie kann überschrieben werden.

9.4. Gelernte Lektionen

Das Arbeiten mit Frameworks oder Software, die von anderen Personen programmiert wurden, kann schwieriger und zeitaufwändiger sein als zuvor angenommen. Zusätzlich besteht die Gefahr von Problemen, die nicht gelöst werden können und dann zu Workarounds führen. Es wurde auch deutlich, dass nicht mehr unterstützte Software viel schneller als erwartet unter Problemen wie nicht mehr vorhandenen oder falsch angegeben Softwarequellen leidet. So gab es beispielsweise bei TEA (siehe Abschnitt 3.2.2 „TEA“) Probleme, weil eine wichtige Adresse exklusiv zu neueren Versionen führt und so auch empfohlene Installationsmöglichkeiten nicht mehr zu Eclipse Neon passen. Auch hat sich gezeigt, dass Elektroniktutorials wie für die Nutzung von den ESP8266-01S-WiFi-Modulen mit AMKs häufig fehlerhaft sind: In einem Großteil wird der ESP8266-01S direkt über den Arduino versorgt, aber in der Praxis konnte dies nicht funktionierend nachgestellt werden, weil zumindest unsere Exemplare einen höheren Energiebedarf aufwiesen.

9.5. Zukünftige Arbeiten

Durch diese Ausarbeitung und deren Resultate werden in verschiedenen Bereichen weitere Forschungsmöglichkeiten oder Anwendungsmöglichkeiten erkennbar. Die während dieser Ausarbeitung genutzten Roboterautos stellen beispielsweise durch die hinzugefügte eigene Spannungsversorgung für das WiFi-Modul, das so nun zuverlässiger arbeiten kann, eine zuverlässigere Testplattform dar. Gleichzeitig können sie für andere Szenarien oder größere Nähe zur Realität weiter verbessert werden, z.B. durch das Nachstellen eines Vorderradantriebes oder weiteren Abstandssensoren. Schrittmotoren können beispielsweise ein zuverlässiges langsames Fahren ermöglichen. Durch die momentanen Motoren erfordert das langsame Fahren Feintuning, damit sie nicht blockieren, und ggf. Änderungen an der Geschwindigkeit des Überholers (siehe Abschnitt 4.1 „Das ‚kooperatives Überholen‘-Szenario“), damit dieser eine höhere Geschwindigkeit aufweist und nicht von der Fahrbahn abkommt. Stürner hat auch in seiner Ausarbeitung verschiedene Vorschläge festgehalten [Stü22].

Die entwickelten MUML-Plug-In-Projekte „ArduinoCLUtilizer“, „MUMLACGPPA“ und „PipelineExecution“ enthalten auch viele Punkte, an denen Verbesserungen durchgeführt werden können. So könnte die Integration der ArduinoCLI z.B. durch die Einführung eines Auswahlmenüs, wenn mehr als ein davon erkennbarer Mikrocontroller angeschlossen ist, wesentlich komfortabler genutzt werden.

Die Unterstützung von Boards, die nicht über einen Bootloader verfügen oder nicht über USB verbunden werden können, kann auch noch implementiert werden.

An der CI/CD-Pipeline ist das ordentliche Implementieren der Automatisierungen der Container-Transformation und der Generation des Componentencodes ein wichtiger Verbesserungspunkt, weil so der Export-Wizard-Workaround entfernt werden kann und eine eigene Startmethode implementiert werden kann. Das Implementieren von Interaktionsfähigkeiten mit verschiedenen

9. Schlussfolgerung

Versionsverwaltungssystemen kann die in der CI/CD-Methode vorgesehenen Schritte des Herunterladens und Hochladens mit der genutzten Versionsverwaltung ermöglichen. Als durchaus hilfreiche Erweiterung könnte ein besserer Kontrollfluss komplexere Abläufe ermöglichen. Das Verbessern des Heraussuchens von Pipelineschritten kann auch so überarbeitet werden, dass es intern das Heraussuchen und Instanziieren von Schritten besser durchführt und dass leichter eigene Klassen für Pipelineschritte hinzugefügt werden können.

Eine ordentliche Integration der Java-Generics kann die Verwaltung von Variablen und deren Typen verbessern.

Die Erstellung einer Variante, die leicht für andere Automatisierungen verwendet werden kann, wäre durchaus für andere Anwendungszwecke ohne MUML oder veralteter Eclipse-Software sehr hilfreich.

Ein wichtiger Punkt ist die Umsetzung einer Möglichkeit für das Updaten OTA. So können auch bei größeren Anzahlen an Roboterautos leichter neue Versionen der zu verwendenden Programmierung verteilt werden und es können auch Versuche hinsichtlich des Updatens ermöglicht werden. Die bereits implementierte Fähigkeit der Arduino-CLI und des Pipelinesystems, .ino-Dateien zu .hex-Dateien zu kompilieren kann hierbei eine große Hilfe darstellen.

Das automatische Testen von MUML-Modellen kann auch integriert werden, z.B. mittels der Transformation von MUML-Modellen zu Matlab-Modellen (siehe Heinzemann et al. [HRB+14]) und dem automatischen Starten von Matlab-Simulationen.

Auch könnte der Build-Prozess so verbessert werden, dass nach Modelländerungen nur die dadurch anders generierten Dateien neu erstellt und bearbeitet werden.

Literaturverzeichnis

- [Anw] N. I. und Anwendungen (DIN NIA). <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>. Version vom 23.2.24. Inzwischen wurde bemerkt, dass die Webseite und deren Punkte geändert wurden. Über <https://web.archive.org/> können alte Versionen eingesehen werden, soweit diese und deren Unter-Seiten archiviert wurden. Für die Ausarbeitung wurde mit einer lokalen Kopie gearbeitet, d.h. es wurde alles am 23.2.24 als Notiz in ein Textdokument kopiert. Diese alte Version wurde beibehalten, weil sie bei einer Besprechung über die Kriterien vorgelegt und für gut befunden wurde. URL: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010> (zitiert auf S. 3, 17–20, 77, 82, 92).
- [Arda] Arduino. *Arduino Sketches*. URL: <https://docs.arduino.cc/learn/programming/sketches/> (zitiert auf S. 26).
- [Ardb] Arduino. *Installation - Download - Latest release*. URL: <https://arduino.github.io/arduino-cli/1.0/installation/#download> (zitiert auf S. 88).
- [Bae] Baeldung. *Ant vs Maven vs Gradle*. URL: <https://www.baeldung.com/ant-maven-gradle> (zitiert auf S. 16, 17).
- [BCD+14] J. Bobolz, M. Czech, A. Dann, J. Geismann, M. Hüwe, A. Krieger, G. Piskachev, D. Schubert, R. Wohlrab. *Final Document*. Group Cyberton, Heinz Nixdorf Institute, University of Paderborn. Supervisors: Prof. Dr. Wilhelm Schäfer, Stefan Dziwok, and Uwe Pohlmann. 2014. URL: https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/PG_Cybertron/PG_Cybertron_Final_Document.pdf (zitiert auf S. 6, 13, 21).
- [BDG+] S. Becker, S. Dziwok, C. Gerking, W. Schäfer, C. Heinzemann, S. Thiele, M. Meyer, C. Priesterjahn, U. Pohlmann, M. Tichy. *The MechatronicUML Design Method - Process and Language for Platform-Independent Modeling*. März 2014 (zitiert auf S. 21).
- [Ben] P. D. O. Bendel. *Cyberphysische Systeme - Definition* | Gabler Wirtschaftslexikon. URL: <https://wirtschaftslexikon.gabler.de/definition/cyberphysische-systeme-54077> (zitiert auf S. 1).
- [Big] S. J. Bigelow. *CI/CD pipelines explained: Everything you need to know*. Abschnitt „Attributes of a good CI/CD pipeline“. URL: <https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know> (zitiert auf S. 18, 20, 77, 89).
- [CH03] K. Czarnecki, S. Helsen. „Classification of Model Transformation Approaches“. In: 2003. URL: <https://api.semanticscholar.org/CorpusID:17970101> (zitiert auf S. 6, 7).

Literaturverzeichnis

- [Coda] G. user Code-Schwabe. *Arduino-Car-Übersicht der Uni stuttgart, main Branch*. URL: <https://github.com/SQA-Robo-Lab/MUML-CodeGen-Wiki/blob/main/user-documentation/main.md> (zitiert auf S. 25–27, 39).
- [Codb] G. user Code-Schwabe. *Overtaking-Cars/arduino-containers_demo*. URL: https://github.com/SQA-Robo-Lab/Overtaking-Cars/tree/main/arduino-containers_demo (zitiert auf S. 91).
- [DP14] A. P. Dann, U. Pohlmann. *The MechatronicUML Hardware Platform Description Method - Process and Language*. Techn. Ber. tr-ri-14-336. v. 0.1. Heinz Nixdorf Institute, University of Paderborn, Feb. 2014 (zitiert auf S. 5, 6).
- [DPP+16] S. Dziwok, U. Pohlmann, G. Piskachev, D. Schubert, S. Thiele, C. Gerking. *The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling, Technical Report tr-ri-16-352*. 2016 (zitiert auf S. 1, 5, 6).
- [Foua] E. Foundation. *Creating the plug-in project*. Platform Plug-in Developer Guide > Programmer’s Guide > Simple plug-in example > Creating the plug-in project. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Ffirstplugin_create.htm (zitiert auf S. 10).
- [Foub] E. Foundation. *Downloads - MechatronicUML*. Virtual Box Image, also Lubuntu LTS 16.04 + MechatronicUML tool suite 1.0 RC 1 (Full-Eclipse Neon x64 Bundle; Released August 2016). URL: <https://www.mechatronicuml.org/download.html> (zitiert auf S. 11, 17).
- [Fouc] E. Foundation. *Eclipse Advanced Scripting Environment (EASE)*. URL: <https://eclipse.dev/ease/> (zitiert auf S. 14, 15).
- [Foud] E. Foundation. *Eclipse Tea*. URL: <https://eclipse.dev/tea/index.php> (zitiert auf S. 15).
- [Foue] E. Foundation. *Fragment*. Plug-in Development Environment Guide > Concepts > Fragment. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fconcepts%2Ffragment.htm&cp%3D4_1_2 (zitiert auf S. 10).
- [Fouf] E. Foundation. *Plug-in*. Plug-in Development Environment Guide > Concepts > Plug-in. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fconcepts%2Fplugin.htm&cp%3D4_1_3 (zitiert auf S. 10, 11).
- [Foug] E. Foundation. *Plug-in Development Environment Overview*. Plug-in Development Environment Guide > Plug-in Development Environment Overview. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Fintro%2Fpde_overview.htm (zitiert auf S. 10).
- [Fouh] E. Foundation. *Running the plug-in*. Platform Plug-in Developer Guide > Programmer’s Guide > Simple plug-in example > Running the plug-in. URL: https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Ffirstplugin_run.htm&cp%3D2_0_2_4 (zitiert auf S. 10).
- [Foui] E. Foundation. *Workbench wizard extension points*. Platform Plug-in Developer Guide > Programmer’s Guide > Dialogs and wizards > Workbench wizard extension points. URL: [https://help.eclipse.org/latest/index.jsp?topic=%2Org.eclipse.platform.doc.isv%2Fguide%2Fdials_wizards_extensions.htm](https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fdials_wizards_extensions.htm) (zitiert auf S. 10).

- [Fouj] E. Foundation. *Writing Modules*. URL: https://eclipse.dev/ease/documentation/writing_modules/ (zitiert auf S. 14, 15).
- [Fouk] T. A. Foundation. *Configuring Apache Maven*. URL: <https://maven.apache.org/configure.html> (zitiert auf S. 16).
- [Foul] T. A. Foundation. *Feature Summary*. URL: <https://maven.apache.org/maven-features.html> (zitiert auf S. 16).
- [Foum] T. A. Foundation. *Homepage von Ant*. URL: <https://ant.apache.org/> (zitiert auf S. 16).
- [Foun] T. A. Foundation. *Homepage von Maven*. URL: <https://maven.apache.org/> (zitiert auf S. 16).
- [Fouo] T. A. S. Foundation. *Apache Ant(TM) 1.10.15 Manual*. URL: <https://ant.apache.org/manual/index.html> (zitiert auf S. 16).
- [Fow20] M. Fowler. *Continuous Integration*. Mai 2020. URL: <https://martinfowler.com/articles/continuousIntegration.html> (zitiert auf S. 1, 7–9, 29).
- [Hata] R. Hat. *Was ist CI/CD? Konzepte und CI/CD Tools im Überblick*. URL: <https://www.redhat.com/de/topics/devops/what-is-ci-cd> (zitiert auf S. 1, 2, 7, 29).
- [Hatb] R. Hat. *Was ist eine CI/CD-Pipeline?* URL: <https://www.redhat.com/de/topics/devops/what-cicd-pipeline> (zitiert auf S. 2, 29).
- [HRB+14] C. Heinzemann, J. Rieke, J. Bröggelwirth, A. Pines, A. Volk. *Translating Mechatronic UML Models to MATLAB/Simulink and Stateflow*. Tech. rep. tr-ri-14-338. Version 0.4.1. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn. Mai 2014 (zitiert auf S. 88, 98).
- [Inca] G. Inc. *Gradle Guides*. URL: <https://gradle.org/guides/> (zitiert auf S. 17).
- [Incb] G. Inc. *Gradle User Manual*. URL: <https://docs.gradle.org/current/userguide/userguide.html> (zitiert auf S. 17).
- [Kab] A. Kabir. *Continuous Integration: A "Typical" Process*. URL: <https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process> (zitiert auf S. 7, 29).
- [Loh] M. Lohrey. *Versionsverwaltung Software im Vergleich*. Hier wird die Liste der verschiedenen Anbieter, die oben angezeigt wird, gemeint. URL: <https://trusted.de/versionsverwaltung> (zitiert auf S. 90).
- [Mus] O. Muscad. *What is Model-Driven Development, and Why is It the Most Important Concept in Low-Code*. URL: <https://datamyte.com/blog/model-driven-application-development/> (zitiert auf S. 1).
- [OMG14] O. M. G. (OMG). „MDA Guide rev. 2.0. Tech. rep. Object Management Group (OMG)“. In: *International Journal of Information Management* (2014) (zitiert auf S. 1).
- [Poh18] U. Pohlmann. „A model-driven software construction approach for cyber-physical systems“. Diss. University of Paderborn, Germany, 2018. URL: <http://d-nb.info/1159954666> (zitiert auf S. 21).

Literaturverzeichnis

- [Reia] G. Reißner. *Arduino-Car-Übersicht der Uni stuttgart, new-motor-driver Branch*. Commit 7a235aa vom 15.6.2023. URL: <https://github.com/SQA-Robo-Lab/Arduino-Car/tree/new-motor-driver> (zitiert auf S. 2).
- [Reib] G. Reißner. *Deployable Code*. URL: https://github.com/SQA-Robo-Lab/Overtaking-Cars/blob/hal_demo/arduino-containers_demo_hal/deployable-files-hal-test/README.md (zitiert auf S. 27, 55, 74, 79, 116–122).
- [Reic] G. Reißner. *Github-Link der Config.hpp von Sofdcar-HAL*. URL: <https://github.com/SQA-Robo-Lab/Sofdcar-HAL/blob/main/examples/SimpleHardwareController/Config.hpp> (zitiert auf S. 74).
- [Reid] G. Reißner. *New Motor Driver (Hardware abstraction libary)*. Commit 4616557 vom 2.11.2023. URL: <https://github.com/SQA-Robo-Lab/Sofdcar-HAL> (zitiert auf S. 2, 22).
- [Reie] G. Reißner. *Overtaking-Cars/arduino-containers_demo_hal/deployable-files-hal-test/*. URL: https://github.com/SQA-Robo-Lab/Overtaking-Cars/tree/hal_demo/arduino-containers_demo_hal/deployable-files-hal-test (zitiert auf S. 91).
- [Stü22] D. Stürner. „Generating Code for Distributed Deployments of Cyber-Physical Systems Using the MechatronicUML“. Magisterarb. Universitätsstraße 38, D-70569 Stuttgart: University of Stuttgart, Mai 2022 (zitiert auf S. 1–3, 6, 7, 13, 21, 22, 26, 27, 52–54, 61, 67, 79, 80, 93, 97).
- [Sun] SunFounder. *Bauanleitung für „PiCar-X AI Car Kit“*. URL: <file:///C:/Users/sebas/Downloads/a0000710-picar-x.pdf> (zitiert auf S. 105).
- [Tea] Y.L.D. Team. *YAML Ain't Markup Language (YAML (TM)) version 1.2*. URL: <https://yaml.org/spec/1.2.2/> (zitiert auf S. 29).
- [Tul] S. Tuli. *Learn How to Set Up a CI/CD Pipeline From Scratch*. URL: <https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch> (zitiert auf S. 29).
- [UBBM17] M. Usman, R. Britto, J. Börstler, E. Mendes. „Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method“. In: *Information and Software Technology* 85 (2017), S. 43–59. doi: <https://doi.org/10.1016/j.infsof.2017.01.006> (zitiert auf S. 17, 77).

Alle Links wurden am 29. Juli 2024 geprüft.

A. GitHub-Repository mit Sourcecode

Der gesamte erstellte oder modifizierte Code, also alle Eclipse-Plug-In-Projekte, das modifizierte MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“ und das angepasste MUML-Projekt „Overtaking-Cars“ können von dem GitHub-Repository „Baumfalk_MA_MUML_Pipeline“ abgerufen werden. Seine Web-Adresse lautet:

https://github.com/st114908/Baumfalk_MA_MUML_Pipeline

In diesem Repository sind weitere Diagramme, die für das Ausarbeitungsdokument zu groß sind, unter dem Verzeichnis „Diagramme“ abgelegt.

Unter „ArduinoCLI“ ist eine Kopie der verwendeten Version der Arduino-CLI beigelegt.

Unter „Installationsskripte“ sind Skripte hinterlegt, die beim Herunterladen und Installieren der verwendeten Software helfen sollen.

B. Ergänzungsmaterial Übersicht und Details zu den Roboterautos

Dieses Ergänzungsmaterial beschreibt den Aufbau der Version der Roboterautos, die während dieser Ausarbeitung als Plattform verwendet wurden. Für jedes Roboterauto wurde ein Auto aus einem „PiCar-X AI Car Kit“-Bausatz von „SunFounder“¹ modifiziert verwendet:

- Die Batteriebox (siehe Schritt 1 auf der Bauanleitung [Sun]) wurde nicht verwendet, sondern ein an einer anderen Stelle platziertes 12 V LiPo-Akku. An ihren Ort wurde ein L2980N-Motortreibermodul montiert.
- Das Kameramodul und seine gesamte Struktur, also alles, was auf der Bauanleitung (siehe Bauanleitung [Sun]) von den Schritten 11 bis 19 zusammengesetzt und montiert wird, wurde weggelassen.
- Die elektronische Steuereinheit „Robot HAT“ und seine Montageteile [Sun] wurden nicht integriert. Dies entspricht dem Weglassen der Schritte 2, 4, 5 und 6 sowie dem Verbinden aller Kabel an diese Steuereinheit. Stattdessen wurde eine Konstruktion aus Acrylplatten entworfen, um auf drei Etagen das 12V-Akku, das WiFi-Modul, einen Arduino Nano als Koordinator-AMK und einen Arduino Mega 2560 Rev3 als Fahrer-AMK zu integrieren. An dieser Struktur ist auch der hintere Abstandssensor hinten rechts nach rechts ausgerichtet befestigt.

Die Elektronik besteht aus den folgenden Komponenten:

- ESP8266-01S-WiFi-Modul inklusive eines Breadboard-Adapters für die Kommunikation per WLAN.
- Ein Arduino Mega 2560 Rev3, der als Fahrer-AMK das Lesen der Sensoren und Steuern der Elektromechanik, also der Motoren und des Lenk-Servos, als Aufgabe hat.
- Ein Arduino Nano, der mittels des WiFi-Moduls für die Kommunikation und Koordination verantwortlich ist.
- Zwei HC-SR04-Ultraschallsensoren für die Distanzmessungen. Der vordere ist nach der Bauanleitung des „PiCar-X AI Car Kit“-Bausatzes [Sun] montiert, um das Annähern an einem anderen Roboterauto zu erkennen, und der hintere ist hinten rechts nach rechts ausgerichtet, um für die Erkennung des Abschlusses des Überholvorganges das zu überholende Roboterauto abzutasten.

¹Shopseite des Bausatzes: https://www.sunfounder.com/products/raspberry-pi-robot-car-kit-picar-x?_pos=1&_sid=e4070601c&_ss=r

B. Ergänzungsmaterial Übersicht und Details zu den Roboterautos

- Drei KY-033-Infrarot-Sensoren für das Aufspüren der Linie, die eine Fahrbahn darstellt, und deren Verlauf. Im Fall des „PiCar-X AI Car Kit“-Bausatzes sind diese Sensoren zu einem Modul zusammengefasst, aber die Funktionsweise unterscheidet sich nicht von drei einzelnen Sensoren.
- Zwei einfache DC-Motoren bzw. Gleichstrommotoren als Antrieb.
- Ein 12V-Akku als 12V-Energiequelle.

Für die Energieversorgung des WiFi-Modules wurde ein einstellbares LM317-Spannungsregler-Modul verwendet, das die Spannung einer 9V-Blockbatterie auf 3,3V herunterregelt. Es kann beispielsweise unter dem Namen „LM317 DC-DC Einstellbarer Step-Down Spannungsregler Adjustable Konverter Modul“ bei dem Online-Versand „Eckstein“ bestellt werden². Prinzipiell sind andere Möglichkeiten für eine leistungsfähige 3,3V-Spannungsversorgung vorhanden, aber die Blockbatterien und die LM317-Module waren die verfügbaren Komponenten.

Der Schaltplan dieser Komponenten wird in Abbildung B.1 abgebildet.

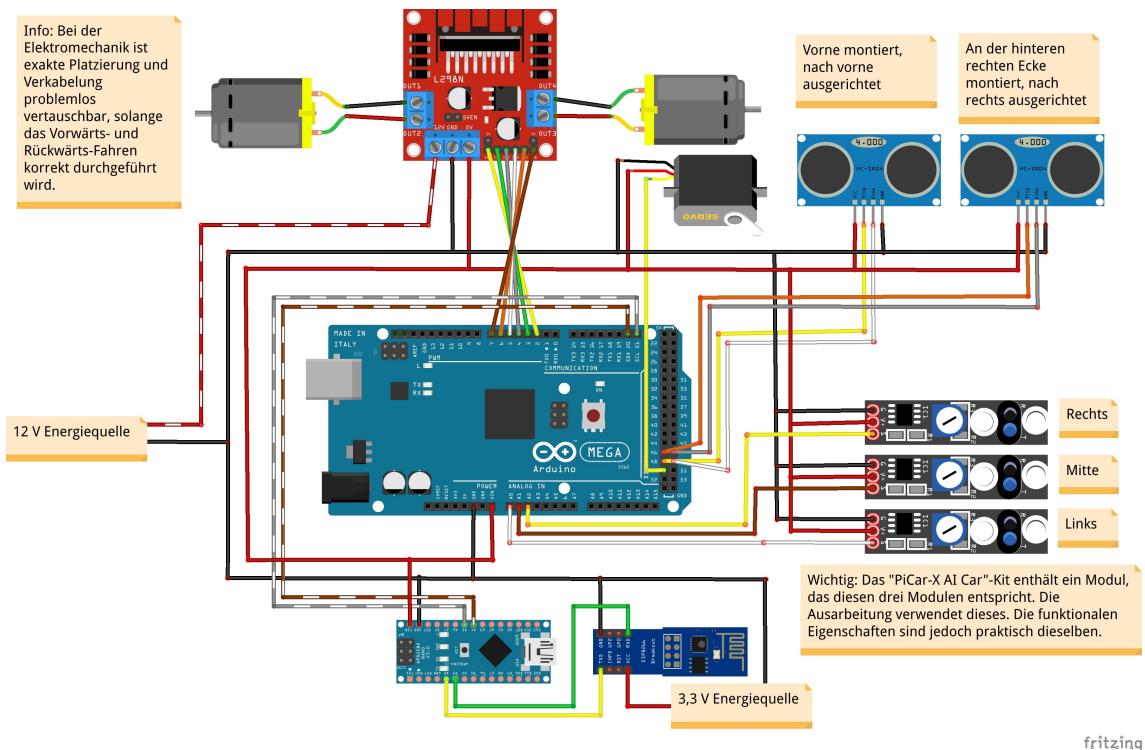


Abbildung B.1.: Schaltplan der elektrischen und elektromechanischen Komponenten eines während dieser Ausarbeitung verwendeten Roboterautos.

Hier folgen zur Veranschaulichung eine Reihe von Kamerabildern, wobei bei diesen die Box für die 9V-Blockbatterie für die Bilder abgelöst wurde:

²Shopseite des Moduls:
DownSpannungsreglerAdjustableKonverterModul

<https://eckstein-shop.de/LM317DC-DCEinstellbarerStep->

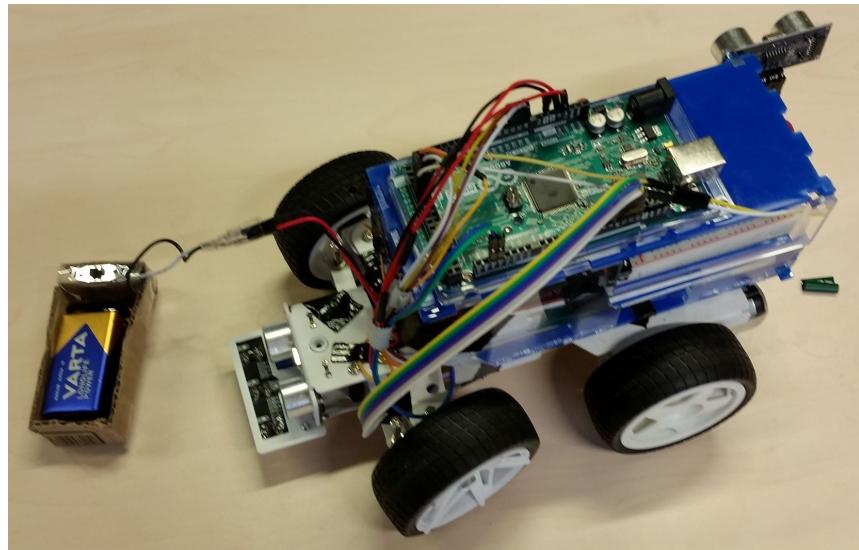


Abbildung B.2.: Das modifizierte Roboterauto von links vorne

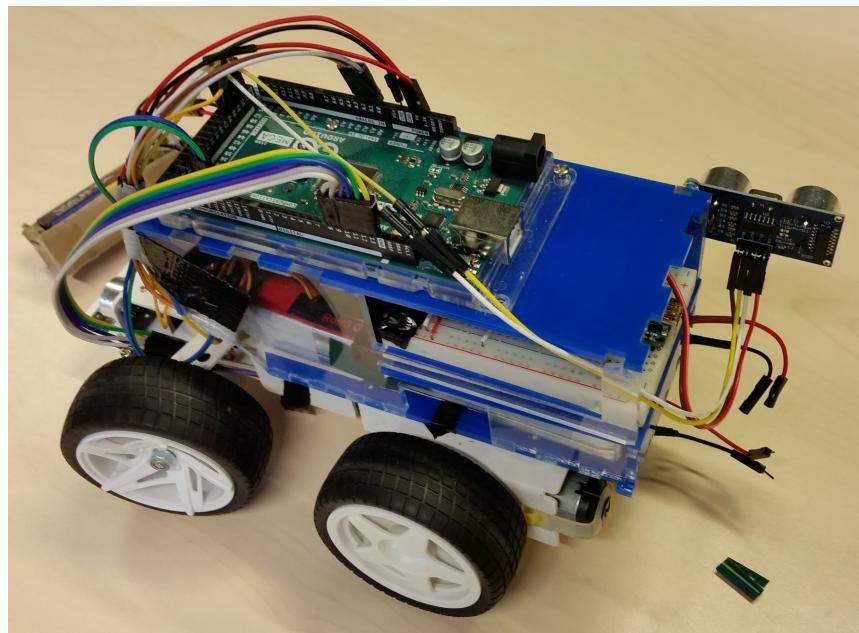


Abbildung B.3.: Das modifizierte Roboterauto von links hinten

B. Ergänzungsmaterial Übersicht und Details zu den Roboterautos

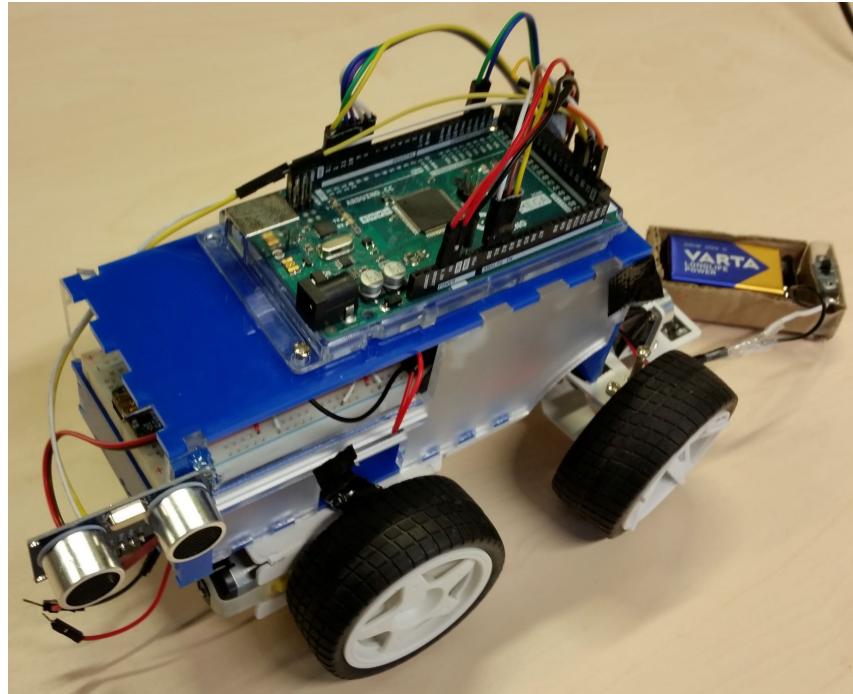


Abbildung B.4.: Das modifizierte Roboterauto von rechts hinten

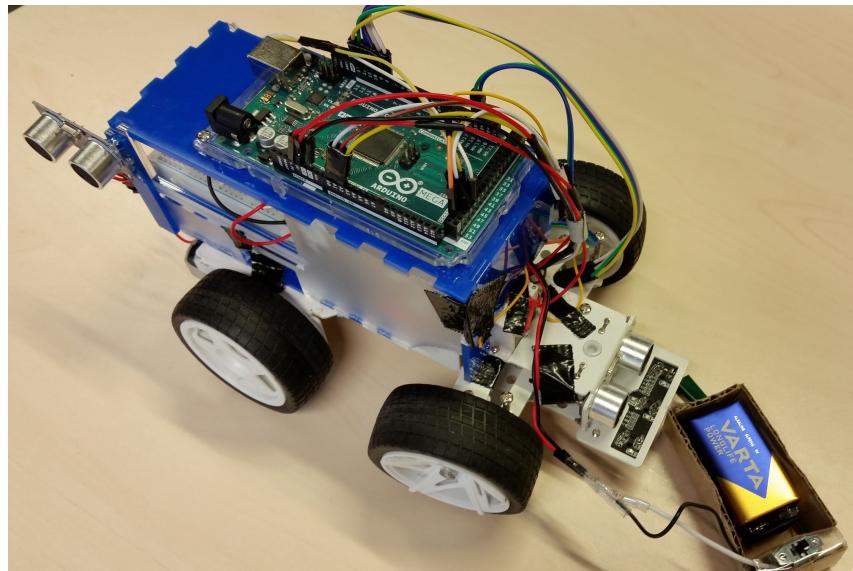


Abbildung B.5.: Das modifizierte Roboterauto von rechts vorne

Die Bibliothek „Sofdcar-HAL“ wird für die Steuerung der Hardware verwendet.
Die Roboterautos wurden von Georg Reißner konstruiert und während dieser Ausarbeitung wurde
der hintere Abstandssensor hinzugefügt.

C. Ergänzungsmaterial Änderungen an MUML-Modellen

In dieser Ergänzung werden die Details zu den Änderungen an den MUML-Modellen beschrieben. Dies erfolgt in Form eines Änderungsverlaufes an den Dateien und deren Inhalten.

Generell wurden bei Dateinamensänderungen in den xml-Informationen dieser umbenannten Dateien die Namensangabe in der jeweiligen Zeile 2 auf den entsprechenden neuen Namen angepasst. Dies wird in der folgenden Liste nicht explizit erwähnt.

Kardinalitäten werden hier immer als [(Untere Schranke), (Obere Schranke)] angegeben.

Nun die Änderungsliste selbst:

Model/protocol/roboCar.coordinationprotocol_diagram:

1. Role "driveControl" nach "courseControl" umbenannt.
2. MessageBuffer "driveControlMessageBuffer" nach "courseControlMessageBuffer" umbenannt.

Model/behavior/roboCar.operationrepository_diagram:

- OperationRepository "RobotCarPowerTrain":
1. Umbenannt nach "RobotCarDriveController".
 2. Alle Operationen bzw. deren Parameter velocity jeweils:
Data Type: Von "Primitive Data Type int32" nach "Primitive Data Type int16" geändert

Model/realtimestatechart:

1. driveControl.realtimestatechart_diagram:
 1. Umbenannt nach "courseControl.realtimestatechart_diagram".
 2. RealtimeStatechart "driveControlRole" nach "courseControlRole" umbenannt.

Model/component:

1. roboCar.component_diagram:
 1. StaticAtomicComponent "DriveControl" angepasst.
 1. Umbenannt nach "CourseControl".
 2. Ports:
 1. Hybrid Port "velocity" angepasst:
Data Type "Primitive Data Type int32" zu

C. Ergänzungsmaterial Änderungen an MUML-Modellen

- "Primitive Data Type int16" geändert.
- 2. Hybrid Port "angle" hinzugefügt
 - 1. Data Type: "Primitive Data Type int8"
 - 2. Kind: "OUT"
 - 3. Sampling Interval of Hybrid Port:
 - 1. Value: "30"
 - 2. Unit: "MILLISECONDS"
- 2. StaticAtomicComponent "PowerTrain"
 - 1. Nach "DriveController" umbenannt
 - 2. Ports:
 - 1. DiscretePort "driveControl" nach "courseControl" umbenannt.
 - 2. ContinuousPort "velocity" angelegt:
Data Type: "Primitive Data Type int32" zu "Primitive Data Type int16" geändert.
 - 3. ContinuousPort "angle" angelegt:
 - 1. Data Type: "Primitive Data Type int8"
 - 2. Kind: "IN"
 - 3. StaticStructuredComponent "RoboCar" erweitert:
 - 1. ComponentPart "driveControl" nach "courseControl" umbenannt.
 - 2. ComponentPart "powerTrain" nach "driveController" umbenannt.
 - 3. AssemblyConnector von "courseControl"/"angle" nach "driveController"/"angle" angelegt.
- 2. driveControlComponent.realtimestatechart_diagram:
 - 1. RealtimeStatechart "DriveControlComponent":
 - 1. Umbenannt nach "CourseControlComponent"
 - 2. State "DriveControl_main":
 - 1. Umbenannt nach "CourseControl_main".
 - 2. Embedded Regions:
RealtimeStatechart "driveControlPort" umbenannt nach "courseControlPort".
 - 2. Datei selbst nach "CourseControlComponent.realtimestatechart_diagram" umbenannt.

Model/hwresource:

- 1. roboCar.resource_diagram:
 - 1. CommunicationProtocolRepository "RoboCarProtocols":
 - 1. Communication Protocols: Link Protocol "ServoControl" hinzufügen.
 - 1. Link Protocol: OTHER
 - 2. Booleans: Keine ausgewählt.
 - 2. Device "DCMotor"/Communication Resources:
Communication Resource "inputSignal" zu "inputSignalMotor" umbenannt. Dies wurde propagiert.
 - 2. Änderungen an StructuredRessource "Arduino":
 - 1. CommunicationRessource "MotorDriver" zu "Motor" umbenannt.

-
- 2. CommunicationRessource "SteeringAxle" hinzugefügt.
 - 1. Protocol: "Link Protocol ServoControl"
 - 2. Cardinality: [0, 1]
 - 3. Device "DCServo" angelegt
 - 1. Device Kind "ACTUATOR".
 - 2. Communication Resources: "inputSignalServo"
 - 1. Protocol: "Link Protocol ServoControl"
 - 2. Cardinality: [1, 1]
 - 2. roboCar.resourceinstance_diagram:
 - 1. ActuatorInstance "SimpleDCServo" angelegt
 - 1. Ressource Type "Device DCServo".
 - 2. Automatisch: Hwports: HW Port "inputSignalServo" erhalten.
 - 1. Communication Resources: "inputSignalServo"
 - 1. Communication Resource:
 - "Communication Resource inputSignalServo"
 - 2. Cardinality: [1, 1]

Model/hwplatform:

- 1. DriveControlUnit.platform_diagram:
 - 1. HWPlatform "DriveControlUnit":
 - 1. Ressource Part "leftMotor" zu "fixedMotor" umbenannt.
 - 2. Ressource Part "rightMotor" entfernt.
 - 3. Ressource Part "steeringServo" hinzugefügt.
 - 1. Cardinality: [1, 1]
 - 2. Resource Type: "Actuator Instance SteeringDCServo"
 - 3. Hw Port Parts: HW Port Part "inputSignalServo"
 - 4. RessourcePart "ecu"/Hw Port Parts:
 - 1. HW Port Part "motorDriver" zu "motor" umbenannt.
 - 2. HW Port Part "steeringAxle" hinzugefügt.
 - 1. Hwport: "HW Port SteeringAxle"
 - 2. Comment: "steeringAxle"

(Nützlich für leichtere Auswahl und
für das Heraussuchen einer ID im nächsten Diagramm.)
 - 5. Network Connector von "ecu"/"steeringAxle" nach
"steeringServo"/"inputSignalServo" angelegt.
 - 6. Umbenannt nach "CourseControlUnit".
 - 2. Datei selbst nach "CourseControlUnit.platform_diagram" umbenannt.
- 2. roboCar.platformminstance_diagram:
 - 1. Unter HWPlatformInstance "slowCar" HWPlatformInstance "driver":
 - 1. In allen DCMotor-Instanzen:
 - Hwports: HW Port Instance "inputSignal" zu "inputSignalMotor"
 - 2. Actuator Instance "leftMotor.S" zu "fixedMotor.S" umbenannt.
 - 3. Actuator Instance "rightMotor.S" entfernt
 - 4. StructuredRessourceInstance "slowCarDriverECU"/ Hwports:
 - 1. HWPortInstance "MotorDriver":
 - Nach "Motor" umbenannt.

C. Ergänzungsmaterial Änderungen an MUML-Modellen

2. Comment: motor
2. HW Port Instance "SteeringAxe" hinzugefügt
 1. Type: "HW Port Part steeringAxe"
 2. Communication Ressource:
"Communication Resource SteeringAxe"
 3. Cardinality: [0, 1]
 4. Comment: steeringAxe
5. Actuator Instance "steeringServo.S" hingefügt
 1. Resource Type: "Device DCservo"
 2. Hwports: HW Port Instance "inputSignalServo"
 1. Type: "platform:/resource/Overtaking-Cars-Baumfalk/
model/roboCar.muuml"/"Root Node"/
"Model Element Category hwplatform"/
"HW Platform DriveControlUnit"
/"Ressource Part steeringServo"/
"HW Port Part "HW Port Part "inputSignalServo" "
 2. Communication Ressource: "Communication Ressource inputSignalServo"
 3. Cardinality: [1, 1]
6. Network Connector von slowCarDriverECU/SteeringAxe nach
steeringServo.S/inputSignalServo angelegt.
 1. Type: Network Connector "Network Connector von
ecu/SteeringAxe nach steeringServo/inputSignalServo"
(Benennung hier sehr nützlich!)
 2. Dasselbe analog mit HWPlatformInstance "fastCar",
aber "*F*" anstatt "*S*" und "fast*" anstatt "slow*".

Model/instance/roboCar2.componentinstanceconfiguration_diagram:

1. StructuredComponentInstance "slowCar" :
 1. AtomicComponentInstance "powerTrain.S" nach
"driveController.S" umbenennen.
 2. AssemblyConnectorInstance von
"driveControl.S"/"angle" nach
"steering.S"/"angle" angelegt.
 1. Type: "AssemblyContainer
(RoboCar.driveControl.angle, RoboCar.steering.angle)"
 3. AtomicComponentInstance "driveControl.S":
 1. Umbenannt nach "courseControl.S".
 2. DiscretePortInstance "driveControl1" umbenannt nach
"courseControl1".
2. Dasselbe analog mit StructuredComponentInstance "fastCar",
aber ".F" anstatt ".S".

Model/roboCarLibraries.osdsl:

1. MotorDriver:
 1. Zu DriveController umbenannt.
 2. "void setSpeed(int32 speed);;" zu
"void setSpeed(int16 speed);;" geändert.

3. Hinzugefügt:

```
void setAngle(int8 angle);
```

Model/roboCar.allocation_specification:

1. Überall "driveControl.F" und "driveControl.S" entsprechend nach "courseControl.F" und "courseControl.S" umbenannt.
2. Ersetzung von Segment

```
constraint collocation fastCarPowerTrain
{
    descriptors(first, second);
    ocl self.getSWInstance('powerTrain.F')->asSet()->product(
        self.getSWInstance('driveControl.F')->asSet()
    ->asSet());
}

constraint collocation slowCarPowerTrain
{
    descriptors(first, second);
    ocl self.getSWInstance('powerTrain.S')->asSet()->product(
        self.getSWInstance('driveControl.S')->asSet()
    ->asSet());
}
```

durch Segment

```
constraint collocation fastCarDriveController
{
    descriptors(first, second);
    ocl self.getSWInstance('driveController.F')->asSet()->product(
        self.getSWInstance('courseControl.F')->asSet()
    ->asSet());
}

constraint collocation slowCarDriveController
{
    descriptors(first, second);
    ocl self.getSWInstance('driveController.S')->asSet()->product(
        self.getSWInstance('courseControl.S')->asSet()
    ->asSet());
}
```


D. Ergänzungsmaterial Post-Processing-Ablauf

In dieser Ergänzung werden die Änderungen am Post-Processing-Ablauf sowie die entsprechenden Pipelineschritte beschrieben.

D.1. Änderungen am Post-Processing-Ablauf

Zum besseren Vergleich wird hierbei eine Tabelle verwendet. Leere Zellen stehen für einen unveränderten bzw. übernommenen Schritt.

D. Ergänzungsmaterial Post-Processing-Ablauf

Tabelle D.1.: Vergleichstabelle der Änderungen beim Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
1	Für alle Dateien in „fastAndSlowCar_v2“: Alle #include-Anweisungen für lokale Dateien anpassen.	
1.1	Die Pfade aller #include-Anweisungen für lokale Dateien anpassen, also z.B. #include „.. /lib/port.h“ zu #include "port.h" korrigieren.	
1.2	(Nicht vorhanden.)	Zusätzlich von Schritt 2: #include clock.h aus dem C++-Check hochziehen.
2	Unter „fastAndSlowCar_v2/components“:	
2.1	#include "clock.h" aus dem C++-Check hochziehen.	Zu Schritt 1 hinzugefügt.
2.2	„coordinatorComponent“-Dateien in die „*CarCoordinatorECU“-Ordner kopieren.	
2.3	Die anderen Dateien in die „*CarDriverECU“-Ordner kopieren.	
3	„fastAndSlowCar_v2/lib“: Alle Dateien außer „clock.h“ und „standardTypes.h“ in die „*ECU“-Ordner kopieren.	
4	„fastAndSlowCar_v2/messages“: „messages_types.h“ in die „*ECU“-Ordner kopieren.	
5	„fastAndSlowCar_v2/operations“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.	

D.1. Änderungen am Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
6	Unter „fastAndSlowCar_v2/RTSCs“:	
6.1	„coordinatorCoordinatorComponent“ „StateChart.c“ in die „*CarCoordinatorECU“-Ordner kopieren.	
6.2	„driveControlDriveControlComponent“ „StateChart.c“ in die „*CarDriverECU“-Ordner kopieren.	„courseControlDriveControlComponent“ „StateChart.c“ in die „*CarDriverECU“-Ordner kopieren.
7	„fastAndSlowCar_v2/types“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.	
8	„fastAndSlowCar_v2/types“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.	
9	Unter APImappings:	Auch unter APImappings, aber „POWERTRAIN“ statt „DRIVECONTROLLER“ und zusätzlich Variationen von CI_POWERTRAININPOWERTRAIN velocityPortaccessCommand für angle bzw. das Konfigurieren des maximalen Lenkwinkels.

D. Ergänzungsmaterial Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
9.1	<p>Alle Dateien mit „*F*“ in den Ordner fastCarDriverECU kopieren:</p> <p>CI_FRONTDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c CI_FRONTDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.h CI_POWERTRAINFPOWERTRAIN velocityPortaccessCommand.c CI_POWERTRAINFPOWERTRAIN velocityPortaccessCommand.h CI_REARDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c CI_REARDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.h</p>	<p>Alle Dateien mit „*F*“ in den Ordner fastCarDriverECU kopieren:</p> <p>CI_FRONTDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c CI_FRONTDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.h CI_DRIVECONTROLLERF DRIVECONTROLLERvelocity PortaccessCommand.c CI_DRIVECONTROLLERF DRIVECONTROLLERvelocity PortaccessCommand.h CI_DRIVECONTROLLERF DRIVECONTROLLERangle PortaccessCommand.c CI_DRIVECONTROLLERF DRIVECONTROLLERangle PortaccessCommand.h CI_REARDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c CI_REARDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.h</p> <p>Hierbei erfolgt das Kopieren der .c-Dateien standardmäßig zusammen mit den entsprechenden Unterschriften von 9.3 für das aufgelisteten Ausfüllen der API-Dateien.</p>

D.1. Änderungen am Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
9.2	<p>Alle Dateien mit „*S*“ in den Ordner slow-CarDriverECU kopieren:</p> <p>CI_FRONTDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.c CI_FRONTDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.h CI_POWERTRAINSPOWERTRAIN velocityPortaccessCommand.c CI_POWERTRAINSPOWERTRAIN velocityPortaccessCommand.h CI_REARDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.c CI_REARDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.h</p>	<p>Alle Dateien mit „*F*“ in den Ordner fastCarDriverECU kopieren:</p> <p>CI_FRONTDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.c CI_FRONTDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.h CI_DRIVECONTROLLERS DRIVECONTROLLERvelocity PortaccessCommand.c CI_DRIVECONTROLLERS DRIVECONTROLLERvelocity PortaccessCommand.h CI_DRIVECONTROLLERS DRIVECONTROLLERangle PortaccessCommand.c CI_DRIVECONTROLLERS DRIVECONTROLLERangle PortaccessCommand.h CI_REARDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.c CI_REARDISTANCESENSORS DISTANCESENSORdistance PortaccessCommand.h</p> <p>Hierbei erfolgt das Kopieren der .c-Dateien standardmäßig zusammen mit den entsprechenden Unterschriften von 9.3 für das aufgelisteten Ausfüllen der API-Dateien.</p>

D. Ergänzungsmaterial Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
9.3	(Nicht vorhanden.)	Methoden-Stubs in den API-Mapping-Dateien ausfüllen: (Die Arbeitsschritte hierfür sind für slowCarDriverECU und fastCarDriverECU analog zu einander, also im Dateinamen entsprechend mit einem S oder F.)
9.3.1	(Nicht vorhanden.)	In jeder API-Mapping (bevorzugt in den .c-Dateien) hinzufügen: <code>#include <SimpleHardwareController_Connector.h></code>
9.3.2	(Nicht vorhanden.)	Für CI_FRONTDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c in fastCarDriverECU: <code>*distance = SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(0);</code>
9.3.3	(Nicht vorhanden.)	Für CI_REARDISTANCESENSORF DISTANCESENSORdistance PortaccessCommand.c in fastCarDriverECU: <code>*distance = SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(1);</code>
9.3.4	(Nicht vorhanden.)	Für CI_DRIVECONTROLLERF DRIVECONTROLLERvelocity PortaccessCommand.c in fastCarDriverECU: <code>*velocity = SimpleHardwareController_DriveController_GetSpeed();</code>
9.3.5	(Nicht vorhanden.)	Für CI_DRIVECONTROLLERF DRIVECONTROLLERangle PortaccessCommand.c in fastCarDriverECU: <code>*angle = SimpleHardwareController_DriveController_GetAngle();</code>

D.1. Änderungen am Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
10	„ https://github.com/SQA-RoboLab/Sofdcar-HAL/blob/main/examples/SimpleHardwareController/Config.hpp “ in die „*CarDriverECU“-Ordner kopieren. Anpassungen vornehmen.	„ https://github.com/SQA-RoboLab/Sofdcar-HAL/blob/main/examples/SimpleHardwareController/Config.hpp “ in die „*CarDriverECU“-Ordner kopieren und anpassen, z.B. die jeweilige Geschwindigkeit und die Pinbelegung für den hinteren Abstandssensor, die durch den Kommentar „rearDistance“ erkannt werden kann, auf {46, 47}. Hierfür kann jedoch eine lokale Kopie generiert werden (siehe Abschnitt „7.3.3.7 Generierung einer lokalen SofdCar-Hal‘-Konfigurationsdatei“), deren Pinbelegungen zu dem in Ergänzungsmaterial B „Ergänzungsmaterial Übersicht und Details zu den Roboterautos“ beschriebenen Aufbau passen. Diese wird dann in die „*CarDriverECU“-Ordner kopiert und erfordert keine Anpassungen hinsichtlich der Pins.
11	Integrierung der HAL-Teile in den „*CarDriverECU.ino“-Dateien durchführen:	
11.1	Anweisungen <pre>#include <SimpleHardwareController.hpp> #include <SimpleHardwareController_Connector.h></pre> hinzufügen.	
11.2	Globale Variable SimpleHardwareController fastCarDriverController; in den .ino-Dateien hinzufügen und beliebig benennen. (Unter „arduino-containers_demo_hal/deployable-files-hal-test/[fast/slow]CarDriverECU/[fast/slow]CarDriverECU.ino“ wurde entsprechend der Name „[fast/slow]CarDriverController“ gewählt. Dies wird automatisiert beibehalten.)	
11.3	In der setup()-Funktion den Abschnitt für den Benutzercode der .ino-Datei die folgenden Zeilen für das initialisieren der HAL und dem bereitstellen des C-Connectors folgende Zeilen hinzufügen: <pre>initSofdcarHalConnectorFor(&fastCarDriverController); fastCarDriverController.initializeCar(config, lineConfig);</pre>	
11.4	Rufe die Schleifen-Funktion der HAL in der loop()-Function der .ino-Datei auf: <pre>fastCarDriverController.loop();</pre>	

D. Ergänzungsmaterial Post-Processing-Ablauf

Schritt	In Reißners Beschreibung [Reib]	Geänderter Ablauf
12	Methoden-Stubs in den API-Mapping-Dateien ausfüllen:	Zu Schritt 9 als Unterschritte 9.3.1 bis 9.3.4 hinzugefügt.
12.1	In jeder API-Mapping (bevorzugt in den .c-Dateien) hinzufügen: <pre>#include <SimpleHardwareController_Connector.h></pre>	Zu Schritt 9 als Unterschritt 9.3.1 hinzugefügt.
12.2	Für CI_FRONTDISTANCESENSORFDISTANCESENSOR distancePortaccessCommand.c in fastCar-DriverECU: *distance = SimpleHardwareController_DistanceSensor_ GetDistanceToClosestMm(0);	Zu Schritt 9 als Unterschritt 9.3.2 hinzugefügt.
12.3	Für CI_REARDISTANCESENSORFDISTANCESENSOR distancePortaccessCommand.c in fastCarDriverECU: *distance = SimpleHardwareController_DistanceSensor_ GetDistanceToClosestMm(1);	Zu Schritt 9 als Unterschritt 9.3.3 hinzugefügt.
12.4	Für CI_DRIVECONTROLLERFDIVECONTROLLER velocityPortaccessCommand.c in fastCarDriverECU: *velocity = SimpleHardwareController_DriveController_ GetSpeed();	Zu Schritt 9 als Unterschritt 9.3.4 hinzugefügt.
13	In den „robotCarPowerTrainOpRep.c“-Dateien die Methoden-Stubs ausfüllen:	For usere Modifikationen: In den „robotCarDriveControllerOpRep.c“ die Methoden-Stubs ausfüllen:
13.1	Binde „SimpleHardwareController_Connector.h“ am Anfang der Datei (auch hier bevorzugt in der .c-Datei) ein.	
13.2	Für das Wechseln in die linke Spur SimpleHardwareController_LineFollower_SetLineToFollow(0); verwenden. Für das Wechseln in die rechte Spur SimpleHardwareController_LineFollower_SetLineToFollow(1); verwenden.	
13.3	Für das Fahren entlang der Linie SimpleHardwareController_DriveController_SetSpeed(velocity); verwenden.	
14	In jedem „*CarCoordinatorECU“-Ordner jeweils: In driveControlDriveControl ComponentStateChart.c Werte eintragen für: desiredVelocity, slowVelocity, distance und lanedistance.	In jedem „*CarCoordinatorECU“-Ordner jeweils: In courseControlCourseControl ComponentStateChart.c Werte eintragen für: desiredVelocity, slowVelocity, distance und lanedistance.

D.2. Gegenüberstellung der Post-Processing-Schritte und der entsprechenden Pipelineschritte der generierbaren Beispielkonfiguration der Pipeline

Ein in den Anweisungen nicht erwähnter Schritt ist das Eintragen der Daten für die Verbindung zu dem zu verwendenden WLAN-Netzwerk und zu dem MQTT-Server. Es müssen also zusätzlich in den „*CarCoordinatorECU.ino“-Dateien jeweils die Structs `WiFiConfig wifiConfig` und `MqttConfig mqttConfig` konfiguriert werden.

D.2. Gegenüberstellung der Post-Processing-Schritte und der entsprechenden Pipelineschritte der generierbaren Beispielkonfiguration der Pipeline

In diesem Abschnitt wird der geänderte Post-Processing-Ablauf mit den entsprechenden Pipeline-schritten der generierbaren Beispielkonfiguration der Pipeline gegenüber gestellt.

Hierbei wird zu jeder Schrittnummer ein eingetragener Pipelineschritt oder, wenn mehrere Schritte vom selben Typ mit geringen Unterschieden verwendet werden, einer der eingetragenen Schritte als Beispiel gezeigt.

Schritt 1.1

Die Pfade aller `#include`-Anweisungen für lokale Dateien anpassen, also z.B. `#include "../lib/port.h"` zu `#include "port.h"` korrigieren.

Beispiel für „lib“:

```
PostProcessingAdjustIncludes:  
  in:  
    componentCodePath: from componentCodeFilesFolderPath  
    faultyInclude: direct "../lib/  
    correctInclude: direct "  
  out:  
    ifSuccessful: ifSuccessful
```

Schritt 1.2

`#include clock.h` aus dem C++-Check hochziehen.

```
PostProcessingMoveIncludeBefore_ifdef__cplusplus:  
  in:  
    componentsPath: from componentsFolderPath  
    targetInclude: direct "clock.h"  
  out:  
    ifSuccessful: ifSuccessful
```

D. Ergänzungsmaterial Post-Processing-Ablauf

Schritt 2.2

„coordinatorComponent“-Dateien in die „*CarCoordinatorECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsWhitelist:  
in:  
    sourceFolder: from componentsFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct CarCoordinatorECU  
    whitelist: direct coordinatorComponent_Interface.h, coordinatorComponent.c  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 2.3

Die anderen Dateien in die „*CarDriverECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
    sourceFolder: from componentsFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct CarDriverECU  
    except: direct coordinatorComponent_Interface.h, coordinatorComponent.c  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 3

„fastAndSlowCar_v2/lib“: Alle Dateien außer „clock.h“ und „standardTypes.h“ in die „*ECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
    sourceFolder: from libFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct ECU  
    except: direct clock.h, standardTypes.h  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 4

„fastAndSlowCar_v2/messages“: „messages_types.h“ in die „*ECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
    sourceFolder: from messagesFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct ECU
```

D.2. Gegenüberstellung der Post-Processing-Schritte und der entsprechenden Pipelineschritte der generierbaren Beispielkonfiguration der Pipeline

```
    except: direct none
out:
    ifSuccessful: ifSuccessful
```

Schritt 5

„fastAndSlowCar_v2/operations“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:
in:
    sourceFolder: from operationsFolderPath
    destinationFolder: from deployableFilesFolderPath
    ECUNameEnding: direct CarDriverECU
    except: direct none
out:
    ifSuccessful: ifSuccessful
```

Schritt 6.1

„coordinatorCoordinatorComponentStateChart.c“ in die „*CarCoordinatorECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsWhitelist:
in:
    sourceFolder: from RTSCsFolderPath
    destinationFolder: from deployableFilesFolderPath
    ECUNameEnding: direct CarCoordinatorECU
    whitelist: direct coordinatorCoordinatorComponentStateChart.c
out:
    ifSuccessful: ifSuccessful
```

Schritt 6.2

„courseControlDriveControlComponentStateChart.c“ in die „*CarDriverECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsWhitelist:
in:
    sourceFolder: from RTSCsFolderPath
    destinationFolder: from deployableFilesFolderPath
    ECUNameEnding: direct CarDriverECU
    whitelist: direct courseControlCourseControlComponentStateChart.c
out:
    ifSuccessful: ifSuccessful
```

D. Ergänzungsmaterial Post-Processing-Ablauf

Schritt 7

„fastAndSlowCar_v2/types“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
    sourceFolder: from typesFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct ECU  
    except: direct none  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 8

„fastAndSlowCar_v2/types“: Alle Dateien in die „*CarDriverECU“-Ordner kopieren.

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
    sourceFolder: from operationsFolderPath  
    destinationFolder: from deployableFilesFolderPath  
    ECUNameEnding: direct CarDriverECU  
    except: direct none  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 9.1 (.h-Dateien)

Alle Dateien mit „*F*“ in den Ordner fastCarDriverECU kopieren.

Hierbei erfolgt das Kopieren der .c-Dateien standardmäßig zusammen mit den entsprechenden Unterschriften von 9.3 für das aufgelistete Ausfüllen der API-Dateien.

Hier zunächst nur die .h-Dateien:

```
CopyFiles:  
in:  
    sourceFolder: from apiMappingsFolderPath  
    destinationFolder: from fastCarDriverECUFolderPath  
    files: direct CI_FRONTDISTANCESENSORDISTANCESENSORdistancePortaccessCommand.h,  
          CI_DRIVECONTROLLERFDRIVECONTROLLERvelocityPortaccessCommand.h,  
          CI_DRIVECONTROLLERFDRIVECONTROLLERanglePortaccessCommand.h,  
          CI_REARDDISTANCESENSORDISTANCESENSORdistancePortaccessCommand.h  
out:  
    ifSuccessful: ifSuccessful
```

Schritte 9.1 (Alle .c-Dateien) mit entsprechenden Unterschritten von 9.3

Alle Dateien mit „*F*“ in den Ordner fastCarDriverECU kopieren.

Hierbei erfolgt das Kopieren der .c-Dateien standardmäßig zusammen mit den entsprechenden Unterschritten von 9.3 für das aufgelistete Ausfüllen der API-Dateien.

Beispiel für CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c: Unterschritt
9.3.1: In jeder API-Mapping (bevorzugt in den .c-Dateien) hinzufügen: #include <
SimpleHardwareController_Connector.h>

(Die Einträge/Pipelineschritte für slowCarDriverECU hierfür sind analog zu denen für fastCarDriverECU mit einem S statt einem F.)

Unterschritt 9.3.2: Für CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c in
fastCarDriverECU: *distance = SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm
(0);

```
PostProcessingAdjustAPIMappingFile:  
in:  
    sourceFolder: from apiMappingsFolderPath  
    destinationFolder: from fastCarDriverECUFolderPath  
    fileName: direct CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c  
    library: direct SimpleHardwareController_Connector.h  
    instruction: direct *distance =  
        SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(0);  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 9.2 (.h-Dateien)

Alle Dateien mit „*S*“ in den Ordner slowCarDriverECU kopieren.

```
CopyFiles:  
in:  
    sourceFolder: from apiMappingsFolderPath  
    destinationFolder: from slowCarDriverECUFolderPath  
    files: direct CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.h,  
          CI_DRIVECONTROLLERSDRIVECONTROLLERvelocityPortaccessCommand.h,  
          CI_DRIVECONTROLLERSDRIVECONTROLLERanglePortaccessCommand.h,  
          CI_REARDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.h  
out:  
    ifSuccessful: ifSuccessful
```

Schritte 9.2 (Alle .c-Dateien) mit entsprechenden Unterschritten von 9.3

Alle Dateien mit „*S*“ in den Ordner slowCarDriverECU kopieren.

Hierbei erfolgt das Kopieren der .c-Dateien standardmäßig zusammen mit den entsprechenden Unterschritten von 9.3 für das aufgelisteten Ausfüllen der API-Dateien.

Beispiel für CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c: Unterschritt
9.3.1: In jeder API-Mapping (bevorzugt in den .c-Dateien) hinzufügen: #include <
SimpleHardwareController_Connector.h>

D. Ergänzungsmaterial Post-Processing-Ablauf

(Die Arbeitsschritte hierfür sind für slowCarDriverECU und fastCarDriverECU analog zu einander, also im Dateinamen entsprechend mit einem S oder F.)

Unterschritt 9.3.2: Für `CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c` in slowCarDriverECU: `*distance = SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(0);`

```
PostProcessingAdjustAPIMappingFile:  
in:  
    sourceFolder: from apiMappingsFolderPath  
    destinationFolder: from slowCarDriverECUFolderPath  
    fileName: direct CI_FRONTDISTANCESENSORDISTANCESENSORDistancePortaccessCommand.c  
    library: direct SimpleHardwareController_Connector.h  
    instruction: direct *distance =  
        SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(0);  
out:  
    ifSuccessful: ifSuccessful
```

Schritt 10

Die Umsetzung der Pipeline für das Anwendungsszenario beruht darauf, zunächst die lokale generierte „LocalSofdcarHalConfig.hpp“ (siehe Abschnitt „7.3.3.7 Generierung einer lokalen „SofdCar-Hal“-Konfigurationsdatei“) zu kopieren

```
PostProcessingCopyLocalConfig_hppToCarDeriverECUs:  
in:  
    arduinoContainersPath: from deployableFilesFolderPath  
out:  
    ifSuccessful: ifSuccessful
```

Danach werden die Werte für das Fahrverhalten angepasst: Hier für den Überholer (siehe Abschnitt 4.1 „Das „kooperatives Überholen“-Szenario“):

```
PostProcessingStateChartValues:  
in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ECUName: direct fastCarDriverECU  
    distanceLimit: direct 40  
    desiredVelocity: from fastCarDesiredVelocity  
    slowVelocity: direct 0  
    laneDistance: direct 70  
out:  
    ifSuccessful: ifSuccessful
```

Hier für den Partner (siehe Abschnitt 4.1 „Das „kooperatives Überholen“-Szenario“):

```
PostProcessingStateChartValues:  
in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ECUName: direct slowCarDriverECU  
    distanceLimit: direct 40  
    desiredVelocity: from slowCarDesiredVelocity  
    slowVelocity: direct 0
```

D.2. Gegenüberstellung der Post-Processing-Schritte und der entsprechenden Pipelineschritte der generierbaren Beispielkonfiguration der Pipeline

```
    laneDistance: direct 70
out:
    ifSuccessful: ifSuccessful
```

Schritt 11 mit Unterschritten 11.1 bis 11.4

Integrierung der HAL-Teile in den „*CarDriverECU.ino“-Dateien durchführen.

```
PostProcessingAddHALPartsIntoCarDriverInoFiles:
in:
    arduinoContainersPath: from deployableFilesFolderPath
    ecuEnding: direct CarDriverECU
out:
    ifSuccessful: ifSuccessful
```

Schritt 13 mit Unterschritten 13.1 bis 13.3

For usere Modifikationen: In den „*robotCarDriveControllerOpRep.c“-Dateien Methoden-Stubs ausfüllen:

```
PostProcessingFillOutMethodStubs:
in:
    arduinoContainersPath: from deployableFilesFolderPath
out:
    ifSuccessful: ifSuccessful
```

Schritt 14

Für das schnelle Roboterauto bzw. den Überholer (siehe Abschnitt 4.1 „Das ‚kooperatives Überholen‘-Szenario“):

```
PostProcessingStateChartValues:
in:
    arduinoContainersPath: from deployableFilesFolderPath
    ECUName: direct fastCarDriverECU
    distanceLimit: direct 40
    desiredVelocity: from fastCarDesiredVelocity
    slowVelocity: direct 0
    laneDistance: direct 70
out:
    ifSuccessful: ifSuccessful
```

Für das langsame Roboterauto bzw. den Partner (siehe Abschnitt 4.1 „Das ‚kooperatives Überholen‘-Szenario“):

D. Ergänzungsmaterial Post-Processing-Ablauf

```
PostProcessingStateChartValues:  
  in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ECUName: direct slowCarDriverECU  
    distanceLimit: direct 40  
    desiredVelocity: from slowCarDesiredVelocity  
    slowVelocity: direct 0  
    laneDistance: direct 70  
  out:  
    ifSuccessful: ifSucce
```

Ergänzung 1: Eintragen der Kommunikationsdaten:

```
PostProcessingConfigureWLANSettings  
  in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ecuName: from CarCoordinatorECUName  
    nameOrSSID: from WLANNameOrSSID  
    password: from WLANPassword_MakeSureThisStaysSafe  
  out:  
    ifSuccessful: ifSuccessful  
  
PostProcessingConfigureMQTTSettings:  
  in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ecuName: from slowCarCoordinatorECUName  
    serverIPAddress: from MQTTServerIPAddress  
    serverPort: from MQTTServerPort  
    clientName: from slowCarCoordinatorECUName  
  out:  
    ifSuccessful: ifSuccessful
```

Ergänzung 2: Ändern der maximalen Größen der Topicnamen und Nachrichten in der seriellen Kommunikation zwischen den Mikrokontrollern eines Roboterautos

```
PostProcessingAdjustSerialCommunicationSizes:  
  in:  
    arduinoContainersPath: from deployableFilesFolderPath  
    ecuEnding: direct ECU  
    topicNameMaxSize: direct 50  
    messageMaxSize: direct 50  
  out:  
    ifSuccessful: ifSuccessful
```

E. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „ArduinoCLUtilizer“

Dieses Ergänzungsmaterial bietet weitere Informationen zu dem Eclipse-Plug-In-Projekt „ArduinoCLUtilizer“.

E.1. Verwendete Version

Die Versionsnummer der Arduino-CLI, die für diese Ausarbeitung verwendet wird, lautet 0.35.3 (Commit: 95cf654 Date: 2024-02-19T13:24:24Z). Vor dem Abschluss dieser Ausarbeitung wurden neuere Versionen veröffentlicht¹, von denen jedoch die aktuelleren die Ausgabe von Rückmeldungen im YAML-Format nicht mehr unterstützen. Zur Gewährleistung der erreichten Stabilität wurde während der Entwicklungsarbeiten kein Versionswechsel vorgenommen. Deshalb ist in dem Repository (siehe Ergänzungsmaterial A „GitHub-Repository mit Sourcecode“) unter dem Ordner „Used Arduino-CLI version“ eine Kopie der verwendeten Version beigelegt, um den Installationsprozess zu vereinfachen.

E.2. Die Konfigurationsdatei „arduinoCLUtilizerConfig.yaml“

Hier folgt ein Beispiel der Konfigurationsdatei:

```
arduinoCLIPathSetInPathEnvironment: false
arduinoCLIDirectory: /home/muml/ArduinoCLI
fallbackBoardIdentifierFQBN: arduino:avr:nano

# Info: The config file generation has detected, that the Arduino-CLI is already installed in
#        the folder set by arduinoCLIDirectory.
# Info: The generated settings should directly work.
```

In diesem Fall hatte der Generator, der sie erstellt hat, vor der eigentlichen Dateierstellung durch einen gescheiterten Testaufruf `arduino-cli config dump` festgestellt, dass in den Systempfaden kein Pfad zu einer funktionierenden Installation der Arduino-CLI vorhanden ist. Folglich wurde `arduinoCLIPathSetInPathEnvironment` auf `false` gesetzt.

Unabhängig vom Ergebnis des ersten Tests prüft der Generator dann, ob die Datei für die Arduino-CLI

¹<https://github.com/arduino/arduino-cli/releases>

E. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „ArduinoCLUtilizer“

an dem erwarteten Pfad („./home/muml/ArduinoCLI/arduino-cli“ mit „arduino-cli“ für die Arduino-CLI-Datei selbst) vorhanden ist. Falls dies nicht der Fall sein sollte, so wird indirekt geprüft, ob sie (die Datei für die Arduino-CLI) unter einem anderen Namen vorhanden ist, indem der Aufruf `arduino -cli config dump` in dem erwarteten Standardverzeichnis („./home/muml/ArduinoCLI“) durchgeführt wird. Wenn mindestens einer der beiden zuletzt genannten Tests erfolgreich ist, so wird die Arduino-CLI als vorhanden angesehen.

Die beiden Kommentarzeilen am Ende sind eine Auswertung des anhand der bisher genannten Tests ermittelten Systemzustands. Wenn beispielsweise alle Tests fehlgeschlagen, so wird stattdessen die folgende Auswertung angehängt:

```
# Info: The config file generation has detected, that the paths Arduino-CLI is neither known  
to the system nor in the directory of arduinoCLIDirectory.  
# Recommendation: Install the Arduino-CLI in the directory of arduinoCLIDirectory or adjust  
the settings to the installation path of the Arduino-CLI.
```

F. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „MUMLAGPPA“

Hier werden zusätzliche Informationen zu dem Eclipse-Plug-In-Projekt „MUMLAGPPA“ beschrieben.

F.1. Einschränkungen und Formate der Variablentypen

Beim Eintragen von Werten in den Variablendefinitionen am Anfang oder als direkte Angaben bei Eingabeparametern müssen Beschränkungen und Formate berücksichtigt werden.

Generell muss bei jedem Eintrag gleichzeitig das YAML-Format eingehalten werden, also wenn beispielsweise ein #-Zeichen in der Eingabe vorkommt, so muss der gesamte Wert in Anführungszeichen gesetzt werden, wobei das '-Zeichen der zu verwendende Anführungszeichentyp ist.

Die Tabelle F.1 listet die internen Typen zusammen mit den jeweiligen Beschränkungen und Formaten auf.

F. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „MUMLACGPPA“

Typ	Einschränkungen und Format
Any	Dieser Typ darf nur intern beim Lesen von Parametern verwendet werden, aber er ist nicht für Variablendefinitionen oder Ausgabeparameter erlaubt.
Number	Ganzzahlige Werte, die in Java als int-Typ verwaltet werden können.
String	Zeichenketten, die in Java als String-Typ verwaltet werden können.
Boolean	Wahrheitswerte true oder false.
(Generell für Pfade in FolderPath und FilePath)	Das Pfadformat, das von Ubuntu bzw. Linux verwendet wird. Wenn also am Anfang ein „\“-Zeichen (ohne Anführungszeichen) steht, wird ein absoluter Pfad angenommen. Wenn nicht, so wird der Pfad als relativ zum Ordnerpfad des Eclipse-Projektes, in dem gearbeitet wird, verwaltet.
FolderPath	Zusätzlich muss der angegebene Pfad ein Ordnerpfad sein.
FilePath	Zusätzlich muss der angegebene Pfad ein Dateipfad sein.
BoardSerialNumber	Die Seriennummer als Zeichenkette. Diese besteht wahrscheinlich aus Buchstaben und Zahlen.
BoardIdentifierFQBN	Der Boardtyp. Wegen der verwendeten Arduino-CLI vorausichtlich im Format [Produzent]:[Prozessortyp]:[Modelltyp] und in Kleinbuchstaben. Wenn ein Hersteller eine andere Angabenform verwendet, so ist diese auch in Ordnung, falls die Arduino-CLI diese akzeptiert.
ConnectionPort	Die Form der Portadressen hängt von dem verwendeten System ab. Falls eine virtuelle Maschine verwendet wird, so kann dies auch einen Einfluss auf die Form der Portadressen haben. Empfehlung: Nicht manuell hinschreiben, sondern per Pipelineschritt <code>LookupBoardBySerialNumber</code> ermitteln lassen.
WLANNName	Prinzipiell dieselben Beschränkungen, die für die Namensgebung von WLAN-Netzwerken gelten.
WLANPassword	Prinzipiell dieselben Beschränkungen, die für die Passwortwahl von WLAN-Netzwerken gelten.
ServerIPAddress	Adressen, wie man sie in Internetbrowsern eingibt, aber ohne „https://“ oder „http://“.
ServerPort	Den Standard für Serverports ¹ befolgend.

Tabelle F.1.: Die Einschränkungen und Formate, die seitens der VariablenTypen gefordert werden.

F.2. Beispiel der Reihenfolgeunterschiede

Für ein Beispiel der Reihenfolgenunterschiede siehe die beiden folgenden Codeauszüge der „ECU_Identifier.h“ in „fastCarDriverECU“.

Durchlauf 1:

```
#ifndef ECU_IDENTIFIER_H
#define ECU_IDENTIFIER_H
```

¹<https://www.wintotal.de/was-ist-ein-port/>

F.2. Beispiel der Reihenfolgeunterschiede

```
// code for ECU Config fastCarDriverECU_config
/**
*
* @brief Identifier for Messages used on fastCarDriverECU_config
* @details Identifier to Identy Local Messages
*/
//Identifier for Messages used on this ECU
#define
MESSAGE_OVERTAKINGPERMISSIONMESSAGESREQUESTPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 1 /**
< ECU Identifier: For the Message-Type: requestPermission */
#define
MESSAGE_OVERTAKINGPERMISSIONMESSAGESEXECUTEDOVERTAKINGOVERTAKINGPERMISSIONMESSAGESMESSAGE 2
/**< ECU Identifier: For the Message-Type: executedOvertaking */
#define
MESSAGE_OVERTAKINGPERMISSIONMESSAGESDENYPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 3 /**<
ECU Identifier: For the Message-Type: denyPermission */
#define
MESSAGE_OVERTAKINGPERMISSIONMESSAGESGRANTPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 4 /**<
ECU Identifier: For the Message-Type: grantPermission */

//Identifier for ComponentInstances
/**
*
* @brief Identifier to distinguish Component Instance on ECU fastCarDriverECU_config
* @details Used by a component container to identify component instances of the same
component type
*/
#define CI_FRONTDISTANCESENSORFDISTANCESENSOR 1 /**< Identifier for Component Instance
frontDistanceSensor.F */
#define CI_DRIVECONTROLLERFDRIVECONTROLLER 2 /**< Identifier for Component Instance
driveController.F */
#define CI_COURSECONTROLFCOURSECONTROL 3 /**< Identifier for Component Instance
courseControl.F */
#define CI_REARDISTANCESENSORFDISTANCESENSOR 4 /**< Identifier for Component Instance
rearDistanceSensor.F */

#endif /* ECU_IDENTIFIER_H */
```

Durchlauf 2:

```
#ifndef ECU_IDENTIFIER_H
#define ECU_IDENTIFIER_H

// code for ECU Config fastCarDriverECU_config
/**
*
* @brief Identifier for Messages used on fastCarDriverECU_config
* @details Identifier to Identy Local Messages
*/
//Identifier for Messages used on this ECU
#define
MESSAGE_OVERTAKINGPERMISSIONMESSAGESREQUESTPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 1 /**
< ECU Identifier: For the Message-Type: requestPermission */
```

F. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „MUMLAGPPA“

```
#define MESSAGE_OVERTAKINGPERMISSIONMESSAGESGRANTPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 2 /*<
ECU Identifier: For the Message-Type: grantPermission */
#define MESSAGE_OVERTAKINGPERMISSIONMESSAGESDENYPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE 3 /*<
ECU Identifier: For the Message-Type: denyPermission */
#define MESSAGE_OVERTAKINGPERMISSIONMESSAGESEXECUTEDOVERTAKINGOVERTAKINGPERMISSIONMESSAGESMESSAGE 4
/*< ECU Identifier: For the Message-Type: executedOvertaking */

//Identifier for ComponentInstances
/**
 *
 * @brief Identifier to distinguish Component Instance on ECU fastCarDriverECU_config
 * @details Used by a component container to identify component instances of the same
component type
 */
#define CI_FRONTDISTANCESENSORDISTANCESENSOR 1 /*< Identifier for Component Instance
frontDistanceSensor.F */
#define CI_REARDISTANCESENSORDISTANCESENSOR 2 /*< Identifier for Component Instance
rearDistanceSensor.F */
#define CI_COURSECONTROLCOURSECONTROL 3 /*< Identifier for Component Instance
courseControl.F */
#define CI_DRIVECONTROLLERFDRIVECONTROLLER 4 /*< Identifier for Component Instance
driveController.F */

#endif /* ECU_IDENTIFIER_H */
```

Unter den Identifizierten für die Nachrichten wurden in diesem Fall die Positionen von
MESSAGE_OVERTAKINGPERMISSIONMESSAGES-
EXECUTEDOVERTAKINGOVERTAKINGPERMISSIONMESSAGESMESSAGE
und
MESSAGE_OVERTAKINGPERMISSIONMESSAGES-
GRANTPERMISSIONOVERTAKINGPERMISSIONMESSAGESMESSAGE
(ohne Bindestriche) getauscht.

Unter den Identifizierten für die Instanzen der Komponenten wurden in diesem Fall die Positionen von CI_REARDISTANCESENSORDISTANCESENSOR und CI_DRIVECONTROLLERFDRIVECONTROLLER getauscht.

F.3. Typischer grober Ablauf der Pipelineschritte

In Abbildung F.1 wird dargestellt, wie der typische grobe Ablauf von Pipelineschritten aufgebaut ist. Am Anfang wird Ausgabeparameter ifSuccessful auf „false“ gesetzt, weil konzeptionell der Schritt noch nicht mit Erfolgsbestätigung fertig ausgeführt wurde. Somit ist es also eine Redundanz, falls aus irgendeinem Grund der Ablauf abbricht.

F.3. Typischer grober Ablauf der Pipelineschritte

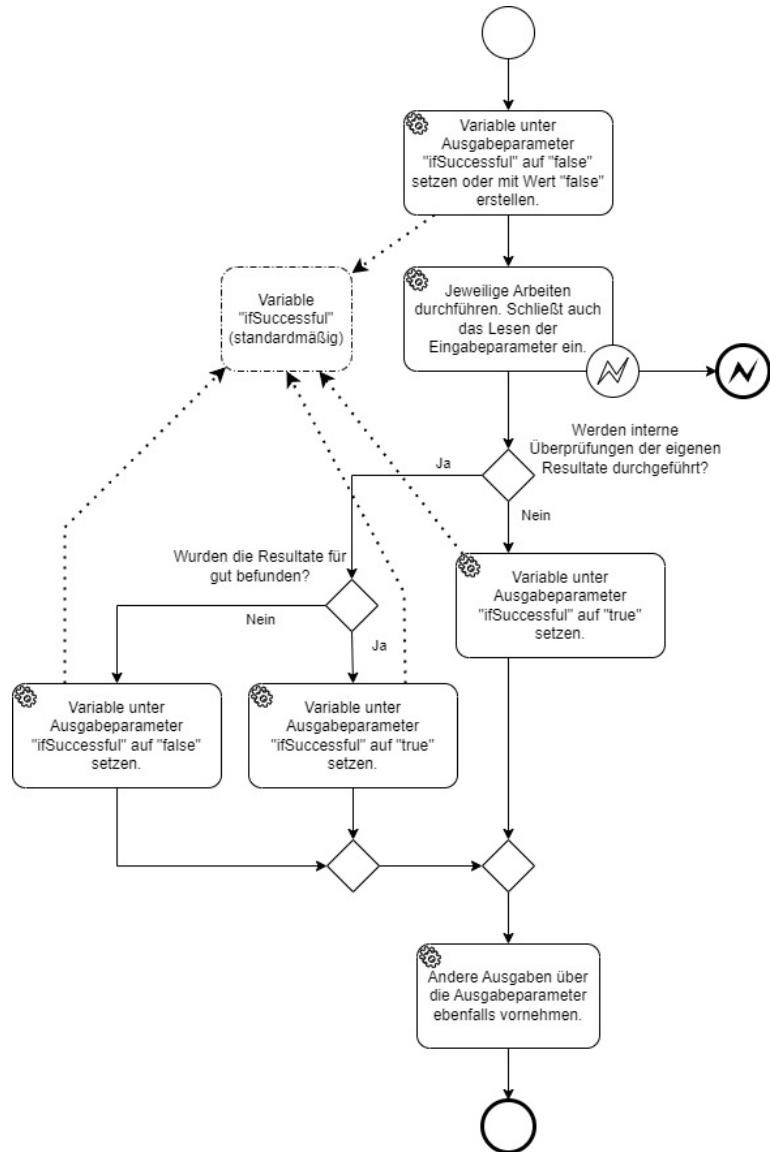


Abbildung F.1.: Typischer grober Ablauf der Pipelineschritte. Pipelineschritte wie DialogMessage und OnlyContinueIfFulfilledElseAbort werden hierbei ignoriert.

G. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „PipelineExecution“

In dieser Ergänzung werden weitere Details zum Eclipse-Plug-In-Projekt „PipelineExecution“ beschrieben.

G.1. Überprüfung der Schritte nach einem Aspekt und gegebenenfalls Anzeigen einer entsprechenden Meldung

Als Beispiel (siehe Listing G.1) dient der Suchdurchlauf auf die Schritte `Compile`, `Upload` und `LookupBoardBySerialNumber`, weil diese auf der Arduino-CLI basieren. Darauf folgt das Anzeigen einer Fehlermeldung, falls mindestens einer dieser Schritte gefunden wurde und die Konfigurationsdatei des Eclipse-Plug-In-Projekts „`ArduinoCLUtilizer`“ nicht gefunden werden konnte. Falls keiner der gesuchten Schritte aufzufinden war, wird deren Konfigurationsdatei nicht benötigt und kann auch fehlen, ohne eine Fehlermeldung zu verursachen.

Über `resetRespectiveSequence()`; wird nur der Befehl `PSRInstance.resetPipelineSequenceProgress()`; mit `PSRInstance` für eine Instanz der Klasse `PipelineSettingsReader` aufgerufen , damit von anderen Überprüfungen der Abfragefortschritt zurückgesetzt wird. Ähnlich hierzu leitet `hasNextStepRespectiveSequence()` zu `PSRInstance.hasNextPipelineSequenceStep()`; weiter und `getNextStepRespectiveSequence()` zu `PSRInstance.getNextPipelineSequenceStep()`;

Für die aufgerufenen Methoden der Instanz der Klasse `PipelineSettingsReader` siehe „7.3.3.5 Zugriff auf die interpretierte Pipeline“.

Als ein anderes Beispiel sucht `handleStepsWithWindowCheck()` nach den Schritten `DialogMessage`, `SelectableTextWindow` und `OnlyContinueIfFulfilledElseAbort`, aber die Informationsmeldung, dass deren Text auf die Konsole der startenden Workspace-Instanz ausgegeben wird, wird in diesem Fall zwangsläufig angezeigt.

G. Ergänzungsmaterial für Eclipse-Plug-In-Projekt „PipelineExecution“

Listing G.1 Suchdurchlauf auf Schritte, die auf der Arduino-CLI basieren sowie das bedingte Anzeigen einer Fehlermeldung.

```
protected boolean handleArduinoCLIUtilizerCheck(Path projectPath) {
    boolean checkForACLIUSettingsFile = false;
    resetRespectiveSequence();
    PipelineStep currentStep;
    while(hasNextStepRespectiveSequence()){
        currentStep = getNextStepRespectiveSequence();
        if ((currentStep.getClass().getSimpleName().equals(Compile.nameFlag))
            || (currentStep.getClass().getSimpleName().equals(Upload.nameFlag))
            || (currentStep.getClass().getSimpleName().equals(LookupBoardBySerialNumber.
nameFlag))) {
            checkForACLIUSettingsFile = true;
            break;
        }
    }

    if (checkForACLIUSettingsFile) {
        Path completeACLIUSettingsFilePath = projectPath
            .resolve(DefaultConfigDirectoryAndFilePath.CONFIG_DIRECTORY_FOLDER_NAME)
            .resolve(DefaultConfigDirectoryAndFilePath.CONFIG_FILE_NAME);
        if (!Files.exists(completeACLIUSettingsFilePath) && Files.isRegularFile(
            completeACLIUSettingsFilePath)) {
            InfoWindow arduinoCLISettingsMissingInfoWindow = new InfoWindow(
                "The ArduinoCLI settings file is missing!", "The ArduinoCLI settings file
is missing!",
                "The ArduinoCLI settings file is missing and at least step that requires
it has been found.\n"
                + "So if you are using a stet that relies on the
ArduinoCLIUtilizer project,\n"
                + "Generate one this way:\n" + "(Right click on a .muml file)/\""
MUMLACGPPA\"/\n"
                + "\"Generate ArduinoCLIUtilizer config file\"");
            addPage(arduinoCLISettingsMissingInfoWindow);
            return true;
        }
    }
    return false;
}
```

H. Ergänzungsmaterial Übersicht der Pipelineschritte

Hier folgt eine Liste über die Einträge aller Pipelineschritte.

AutoGitCommitAllAndPushCommand:

```
in:  
  comment: Triggered automatic adding of all files, committing and uploading.  
  remoteName: origin  
  branchName: master  
out:  
  ifSuccessful: ifSuccessful
```

Compile:

```
in:  
  boardTypeIdentifierFQBN: direct arduino:avr:uno  
  targetInoFile: direct arduino-containers/fastCarDriverECU/fastCarDriverECU.ino  
  saveCompiledFilesNearby: direct true  
out:  
  ifSuccessful: ifSuccessful  
  resultMessage: resultMessage
```

ComponentCodeGeneration:

```
in:  
  roboCar_mumlSourceFile: direct model/roboCar.muml  
  arduinoContainersDestinationFolder: direct arduino-containers  
out:  
  ifSuccessful: ifSuccessful
```

ContainerTransformation:

```
in:  
  roboCar_mumlSourceFile: direct model/roboCar.muml  
  middlewareOption: direct MQTT_I2C_CONFIG # Or DDS_CONFIG  
  muml_containerFileDestination: direct container-models/MUML_Container.muml_container  
out:  
  ifSuccessful: ifSuccessful
```

ContainerCodeGeneration:

```
in:  
  muml_containerSourceFile: direct container-models/MUML_Container.muml_container  
  arduinoContainersDestinationFolder: direct arduino-containers
```

H. Ergänzungsmaterial Übersicht der Pipelineschritte

```
out:  
    ifSuccessful: ifSuccessful  
  
CopyFolder:  
in:  
    sourcePath: direct exampleFolderPath1  
    destinationPath: direct exampleFolderPath2  
out:  
    ifSuccessful: ifSuccessful  
  
CopyFiles:  
in:  
    sourceFolder: direct exampleFilePath1  
    destinationFolder: direct exampleFilePath2  
    files: dummy1.txt, dummy2.txt  
out:  
    ifSuccessful: ifSuccessful  
  
CreateFolder:  
in:  
    path: direct exampleFolderPath  
    returnFailureIfTargetAlreadyExists: direct false  
out:  
    ifSuccessful: ifSuccessful  
  
DeleteFile:  
in:  
    path: direct exampleFilePath  
out:  
    ifSuccessful: ifSuccessful  
  
DeleteFolder:  
in:  
    path: direct exampleFolderPath  
out:  
    ifSuccessful: ifSuccessful  
  
DialogMessage:  
in:  
    condition: direct true  
    message: direct The condition is true, so this will be shown.  
  
LookupBoardBySerialNumber:  
in:  
    boardSerialNumber: direct 8593533337351A0B051  
    boardTypeIdentifierFQBN: direct arduino:avr:uno
```

```
out:
  ifSuccessful: ifSuccessful
  foundPortAddress: foundPortAddress

OnlyContinueIfFulfilledElseAbort:
in:
  condition: direct false
  message: |-  
  direct Pipeline aborted!
  The condition has been evaluated to false.

PostProcessingAddHALPartsIntoCarDriverInoFiles:
in:
  arduinoContainersPath: direct arduino-containers
  ecuEnding: direct CarDriverECU
out:
  ifSuccessful: ifSuccessful

PostProcessingAdjustAPIMappingFile:
in:
  sourceFolder: direct arduino-containers/APImappings
  destinationFolder: direct arduino-containers/APImappings
  fileName: direct CI_FRONTDISTANCESENSORDISTANCESENSORdistancePortaccessCommand.c
  library: direct SimpleHardwareController_Connector.h
  instruction: direct *distance =
SimpleHardwareController_DistanceSensor_GetDistanceToClosestMm(0);
out:
  ifSuccessful: ifSuccessful

PostProcessingAdjustIncludes:
in:
  componentCodePath: direct arduino-containers/fastAndSlowCar_v2
  faultyInclude: direct "../components/"
  correctInclude: direct "
out:
  ifSuccessful: ifSuccessful

PostProcessingAdjustSerialCommunicationSizes:
in:
  arduinoContainersPath: direct arduino-containers
  ecuEnding: direct ECU
  topicNameMaxSize: direct 50
  messageMaxSize: direct 50
out:
  ifSuccessful: ifSuccessful

PostProcessingConfigureMQTTSettings:
```

H. Ergänzungsmaterial Übersicht der Pipelineschritte

```
in:  
  arduinoContainersPath: direct arduino-containers  
  ecuName: direct CarCoordinatorECU  
  serverIPAddress: direct DummyServerIPAddress  
  serverPort: direct 1883  
  clientName: direct DummyCoordinatorECU_config  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingConfigureWLANSettings:  
in:  
  arduinoContainersPath: direct arduino-containers  
  ecuEnding: direct CarCoordinatorECU  
  nameOrSSID: direct DummyWLANNNameOrSSID  
  password: direct DummyWLANPassword  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingCopyFolderContentsToECUsAndExcept:  
in:  
  sourceFolder: direct arduino-containers/fastAndSlowCar_v2/lib  
  destinationFolder: direct arduino-containers  
  ECUNameEnding: direct ECU  
  except: direct clock.h, standardTypes.h  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingCopyFolderContentsToECUsWhitelist:  
in:  
  sourceFolder: direct arduino-containers/fastAndSlowCar_v2/lib  
  destinationFolder: direct arduino-containers  
  ECUNameEnding: direct ECU  
  whitelist: direct coordinatorComponent_Interface.h, coordinatorComponent.c  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingCopyLocalConfig_hppToCarDeriverECUs:  
in:  
  arduinoContainersPath: direct arduino-containers  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingDownloadConfig_hpp:  
in:  
  arduinoContainersPath: direct arduino-containers  
out:  
  ifSuccessful: ifSuccessful
```

```
PostProcessingFillOutMethodStubs:
  in:
    arduinoContainersPath: direct arduino-containers
  out:
    ifSuccessful: ifSuccessful

PostProcessingInsertAtIndex:
  in:
    filePath: direct arduino-containers/fastCarDriverECU/
courseControlCourseControlComponentStateChart.c
    insertIndex: direct 0
    lineToInsert: direct // Comment as example.
  out:
    ifSuccessful: ifSuccessful

PostProcessingMoveIncludeBefore_ifdef__cplusplus:
  in:
    componentsPath: direct arduino-containers/fastAndSlowCar_v2/components
    targetInclude: direct "clock.h"
  out:
    ifSuccessful: ifSuccessful

PostProcessingStateChartValues:
  in:
    arduinoContainersPath: direct arduino-containers
    ECUName: direct fastCarCoordinatorECU
    distanceLimit: direct 40
    desiredVelocity: direct 65
    slowVelocity: direct 0
    laneDistance: direct 70
  out:
    ifSuccessful: ifSuccessful

PostProcessingStateChartValuesFlexible:
  in:
    arduinoContainersPath: direct arduino-containers
    ECUName: direct FastCarDriverECU
    fileName: direct courseControlCourseControlComponentStateChart.c
    targetStateChartValueName: direct desiredVelocity
    valueToSet: direct 12
  out:
    ifSuccessful: ifSuccessful

ReplaceLineContent:
  in:
    filePath: direct arduino-containers/fastCarDriverECU/
courseControlCourseControlComponentStateChart.c
```

H. Ergänzungsmaterial Übersicht der Pipelineschritte

```
targetLineContent: direct stateChart->distanceLimit = stateChart->distanceLimit  
= 1;  
contentReplacement: direct stateChart->distanceLimit = stateChart->distanceLimit  
= 0;  
out:  
    ifSuccessful: ifSuccessful  
  
SaveToFile:  
in:  
    path: direct example.txt  
    text: direct Example text.  
out:  
    ifSuccessful: ifSuccessful  
  
SelectableTextWindow:  
in:  
    condition: direct true  
    message: direct The condition is true, so this will be shown.  
  
TerminalCommand:  
in:  
    terminalCommand: direct echo example  
    exitCodeNumberForSuccessfulExecution: direct 0  
out:  
    ifSuccessful: ifSuccessful  
    exitCode: exitCode  
    normalFeedback: normalFeedback  
    errorFeedback: errorFeedback  
  
Upload:  
in:  
    portAddress: direct COM0  
    boardTypeIdentifierFQBN: direct arduino:avr:uno  
    targetInoOrHexFile: direct arduino-containers/fastCarDriverECU/CompiledFiles/  
fastCarDriverECU.ino.hex  
out:  
    ifSuccessful: ifSuccessful  
    resultMessage: resultMessage
```

I. Verwendete Software und Projekte

Im Folgenden werden die verwendete Software und die verwendeten Projekte aufgelistet.

- Die VirtualBox von Oracle (<https://www.virtualbox.org/>).
 - MechatronicUML <https://www.mechatronicuml.org/de/index.html>, genauer gesagt das vorbereitete Lubuntu-Image <https://www.mechatronicuml.org/en/download.html>
 - Die MUML-Plug-Ins vom Fraunhofer und in manchen Fällen davon der Branch „stuerner_ma“ (siehe https://github.com/SQA-Robo-Lab/MUML_1_0-win32-x86_64):
 - <https://github.com/fraunhofer-iem/mechatronicuml-core> (branch: master)
 - <https://github.com/fraunhofer-iem/mechatronicuml-pim> (branch: master)
 - <https://github.com/fraunhofer-iem/mechatronicuml-pm> (branch: master)
 - https://github.com/fraunhofer-iem/mechatronicuml-psm/tree/stuerner_ma (branch: stuerner_ma)
 - https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-container/tree/stuerner_ma (branch: stuerner_ma)
 - * In diesem Fall wurden Modifikationen vorgenommen und eine Kopie dieser modifizierten Variante ist in dem Repository „Baumfalk_MA_MUML_Pipeline“ (siehe Ergänzungsmaterial A „GitHub-Repository mit Sourcecode“).
 - https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-type/tree/stuerner_ma (branch: stuerner_ma)
- MUML-Projekt „Overtaking-Cars“, Branch „hal_demo“ https://github.com/SQA-Robo-Lab/Overtaking-Cars/tree/hal_demo

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift