5 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 [TODO: Quellenverweise auf Fowler, Redhat und Samarpit Tuli . Learn How to Set Up a CI/CD Pipeline From Scratch. https://dzone.com/articles/learn how to setup a cicd pipeline from scratch (((<--- Diese Quelle aus der Präsentation wird später noch ordentlich eingefügt werden.)))]. Die Konfigurationen werden im YAML-Format [TODO: Link] 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 „Container Code Generation“ und die „Component Code Generation“ 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 5.1: Konfigurationen erfolgen im YAML-Format
ANF 5.2: Frühes Feedback bei Fehlern
ANF 5.3: Automatische Tests der lokalen Arbeitskopie
ANF 5.4: Automatischer Build-Prozess
ANF 5.5: Schneller Build-Prozess
ANF 5.6: Automatisches Hochladen von der lokalen Arbeitskopie auf die Versionsverwaltung
ANF 5.7: Automatisches Herunterladen des aktuellen Codestandes von der Versionsverwaltung
ANF 5.8: Automatisches Deployment
ANF 5.9: Konfigurationsmöglichkeit für die einzelne automatische Ausführung von den MUML-Werkzeugen für die Container Transformation, die Container Code Generation und die Component Code Generation sowie dem Post-Processing

Die verworfenen Ansätze werden nicht beschrieben, sondern nur der umgesetzte Entwurf.
Die Beschreibung zu diesem ist wie folgt aufgeteilt:
In 5.1 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. [(((TODO: Evtl bessere Zusammenfassung.)))]
In 5.2 werden die nicht oder unzureichend erfüllbaren Anforderungen erläutert.
In 5.3 werden weitere Aspekte und deren zugrunde liegenden Entscheidungen erklärt.
In 5.4 werden beispielhaft sechs Pipelineschritte, die sich aus den Anforderungen ergeben, beschrieben.
In 5.5 wird der Aufbau der Konfigurationsdatei detaillierter erläutert.

5.1: 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 Kapitel 5.4 [TODO: Verweis]. Für den Arbeitsablauf mit der Vorbereitung und Nutzung der CI/CD-Pipeline sind folgende Arbeitsphasen und deren Schritte vorgesehen (siehe auch Diagramme [TODO: Verweise], wobei diese auf die Pipeline selbst begrenzt sind):

Phase 1.: Sofern notwendig, wird zuerst mit den MUML-Werkzeugen modelliert oder andere Arbeiten vorgenommen (siehe T1 [TODO: Verweis]).
Phase 2.: Anschließend wird mit den Konfigurationsdateien gearbeitet (Siehe T3 in [TODO: Verweise auf die beiden Diagramme Übersicht und Konfigurationsteil]), 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 ANF5.1 erfüllt.
Phase 2.1: Falls diese noch nicht vorhanden sind, so kann deren Generierung gestartet werden (siehe T2.1, T2.2 und T2.4 und T2.5 in [TODO: Verweis]). Hierbei entspricht die Beispielkonfiguration in „arduinoCLIUtilizerConfig.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 eingetragen werden (siehe T2.3 in [TODO: Verweis]). Im Fall der Datei „pipelineSettings.yaml“ sind die Beispieleinstellungen auf das in [TODO: Verweis] beschriebene Anwendungsszenario des Überholvorganges und dessen MUML-Modellen für die Roboterautos sowie den im Rahmen dieser Ausarbeitung vorgenommenen Änderungen (siehe [TODO: Verweis]) orientiert. Es kann aber auch eine Datei „Each_pipeline_step_by_an_example.txt“ generiert werden lassen, die eine Sammlung von allen Piepelineschritten 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 korrekt passen, sollten sie vorgenommen werden (siehe T2.3 in [TODO: Verweis]). Die Einstellungen der Pipeline (siehe T2.7) 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.
ANF 5.9 wird über das Konzipieren eines eigenen Bereiches für die Einstellungen der einzelnen automatischen Ausführungen von den MUML-Werkzeugen (Container Transformation, die Container Code Generation und die Component Code Generation) 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 durchführen zu müssen.
Für weitere Details hinsichtlich des Aufbaus und der Nutzung siehe 5.3.2 [TODO: Verweis].

Phase 3.: Dann wird die Ausführung der Pipeline gestartet (siehe T3 in [TODO: Verweis]). 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 [TODO: Verweis]). Falls sie fehlt, wird der Vorgang abgebrochen und eine entsprechende Fehlermeldung angezeigt (siehe T4.2 in [TODO: Verweis]).
Phase 3.2: Anschließend wird sie gelesen (siehe T4.3 in [TODO: Verweis]). Falls Syntaxfehler und Datenflussfehler gefunden werden (siehe T4.4 in [TODO: Verweis]), so findet auch hier ein Abbruch mit entsprechender Fehlermeldung statt (siehe T4.5 in [TODO: Verweis]).

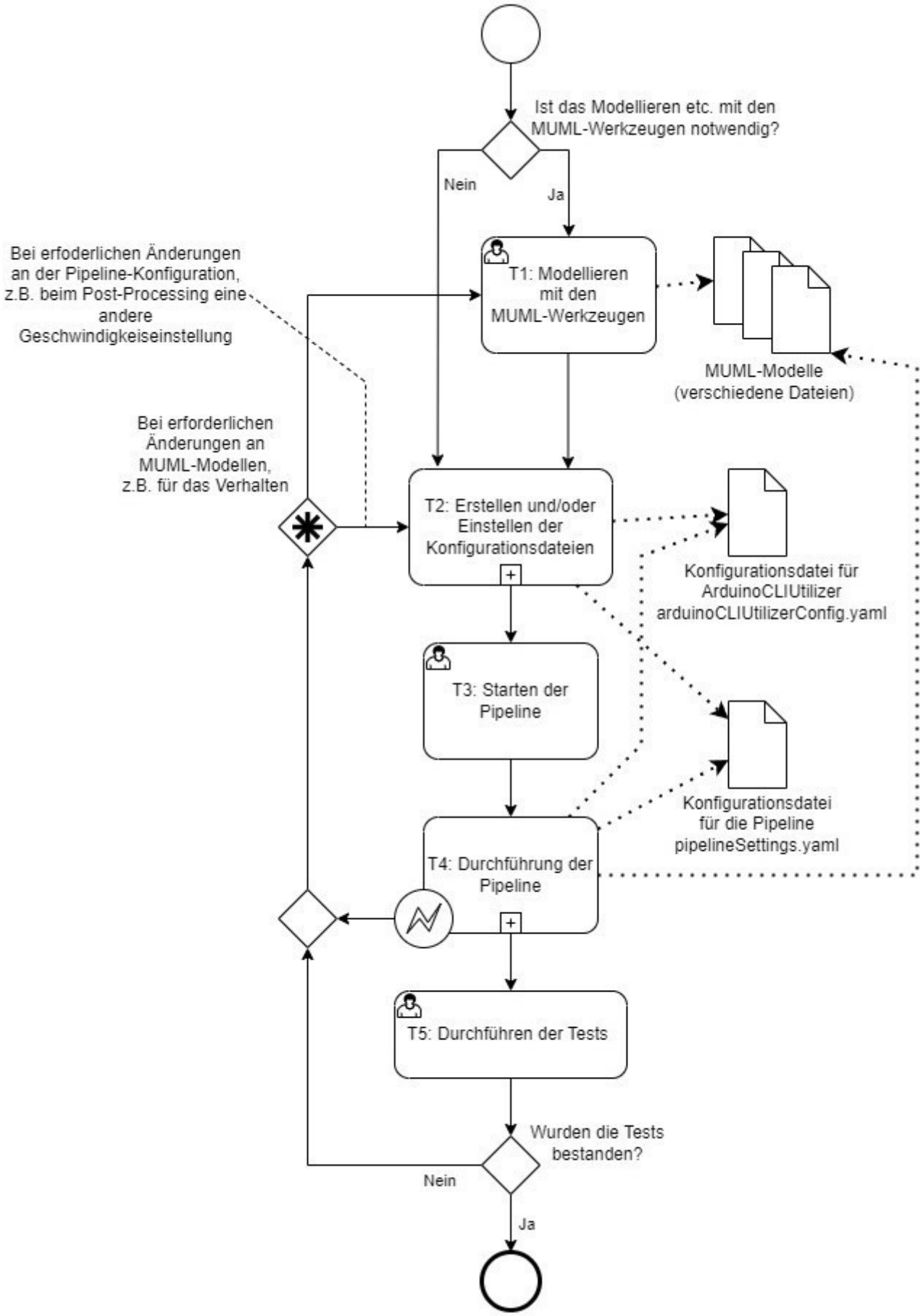
Phase 3.3: Falls der ArduinoCLIUtilizer durch mindestens einen ihn verwendenden Schritt verwendet wird, wird das Vorhandensein der Konfigurationsdatei für ihn überprüft (siehe T4.6 in [TODO: Verweis]). Falls sie fehlt, so erfolgt ebenfalls ein Abbruch mit Fehlermeldung (siehe T4.7 in [TODO: Verweis]).
Phase 3.4: Danach findet die eigentliche Ausführung der Pipelineschritte aus Pipelinesequenz bzw. der eigentlichen auszuführenden Pipeline statt (siehe T4.8 in [TODO: Verweis]). 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 5.4 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 5.2 erfüllt. ANF 5.8 wird durch den Schritt „Upload“ erfüllt, der das Hochladen auf einen unter dem angegebenen Port angeschlossenen Board repräsentiert. Zwecks Modularisierung soll die dafür verantwortliche Klasse in ArduinoCLIUtlizer dabei zwischen kompilierten Dateien bzw. .hex-Dateien und unkompiliertem Arduinocode bzw. .ino-Dateien unterscheiden und entsprechend einen internen Kompilliervorgang 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 6.4 „Vollständige Automatisierung per Pipeline“ [TODO: Verweis]) 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 der komplette Ablauf deutlich schneller erfolgt (siehe [TODO: Später Zeitvergleich]). Auf diese Weise wird ANF 5.5 erfüllt.

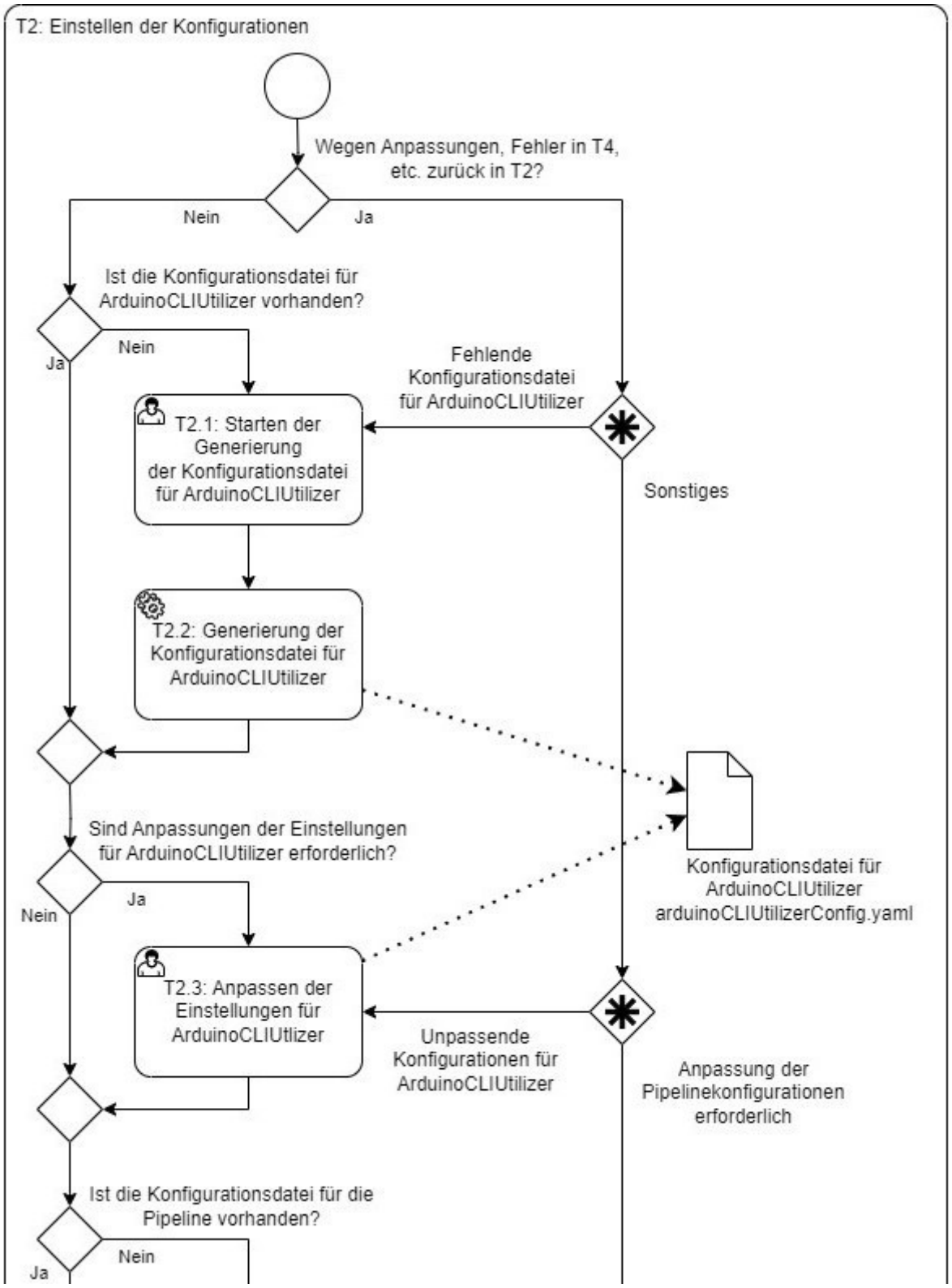
Phase 4.: Falls alles ausgeführt werden konnte, kann der Nutzer oder die Nutzerin Tests durchführen oder andere Handlungen durchführen.
Phase 4.1: Je nach Ergebnis der Tests ist die Durchführung entweder abgeschlossen oder sie geht auch hier zu einem früheren Schritt zurück.

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

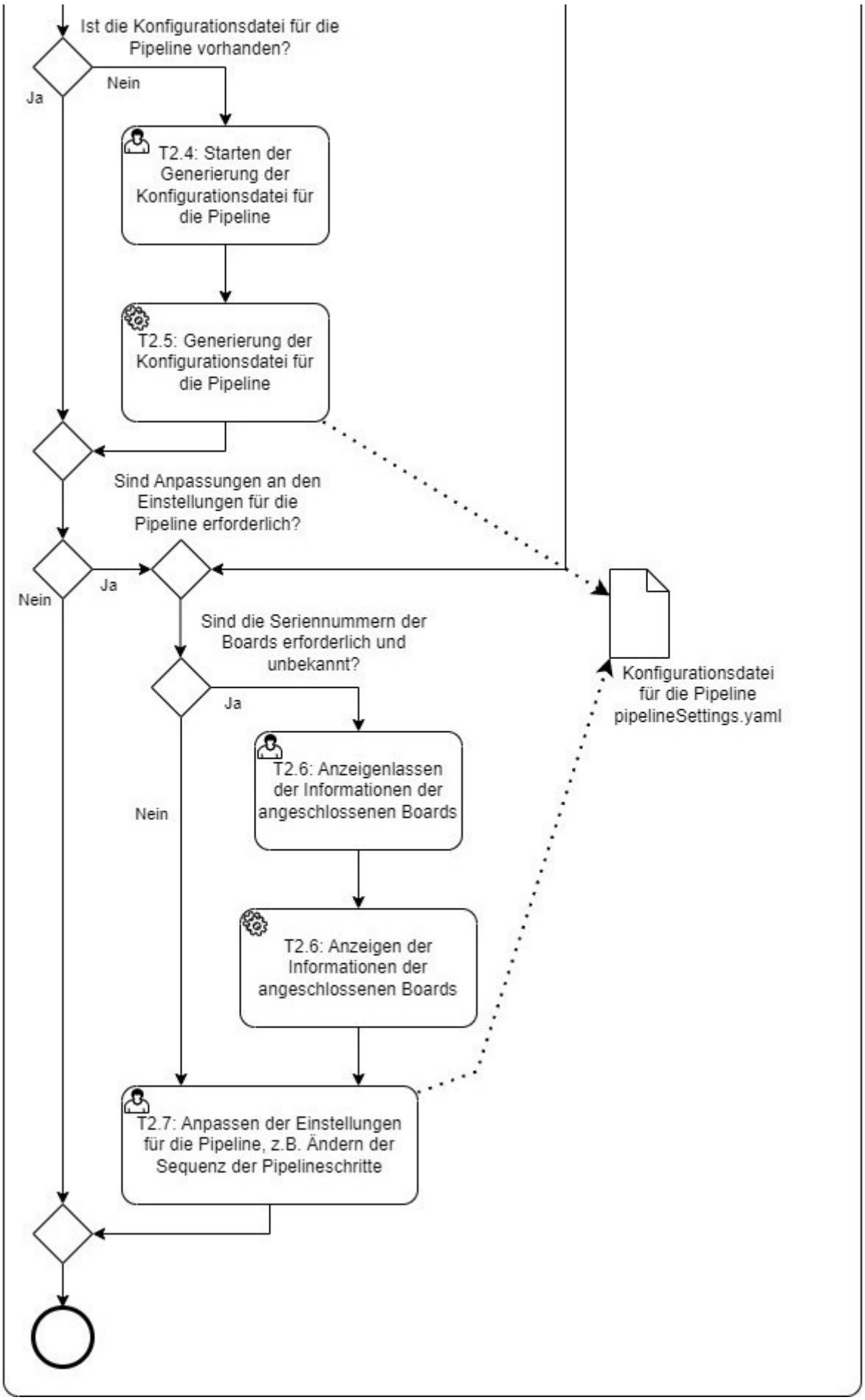
#


Ist das Modellieren etc. mit den MUML-Werkzeugen notwendig?
Nein
Ja
T1: Modellieren mit den MUML-Werkzeugen
MUML-Modelle (verschiedene Dateien)
Bei erfoderlichen Änderungen an der Pipeline-Konfiguration, z.B. beim Post-Processing eine andere Geschwindigkeiseinstellung
Bei erforderlichen Änderungen an MUML-Modellen, z.B. für das Verhalten
T2: Erstellen und/oder Einstellen der Konfigurationsdateien
Konfigurationsdatei für ArduinoCLIUtilizer arduinoCLIUtilizerConfig.yaml
T3: Starten der Pipeline
Konfigurationsdatei für die Pipeline pipelineSettings.yaml
T4: Durchführung der Pipeline
T5: Durchführen der Tests
Wurden die Tests bestanden?
Nein
Ja

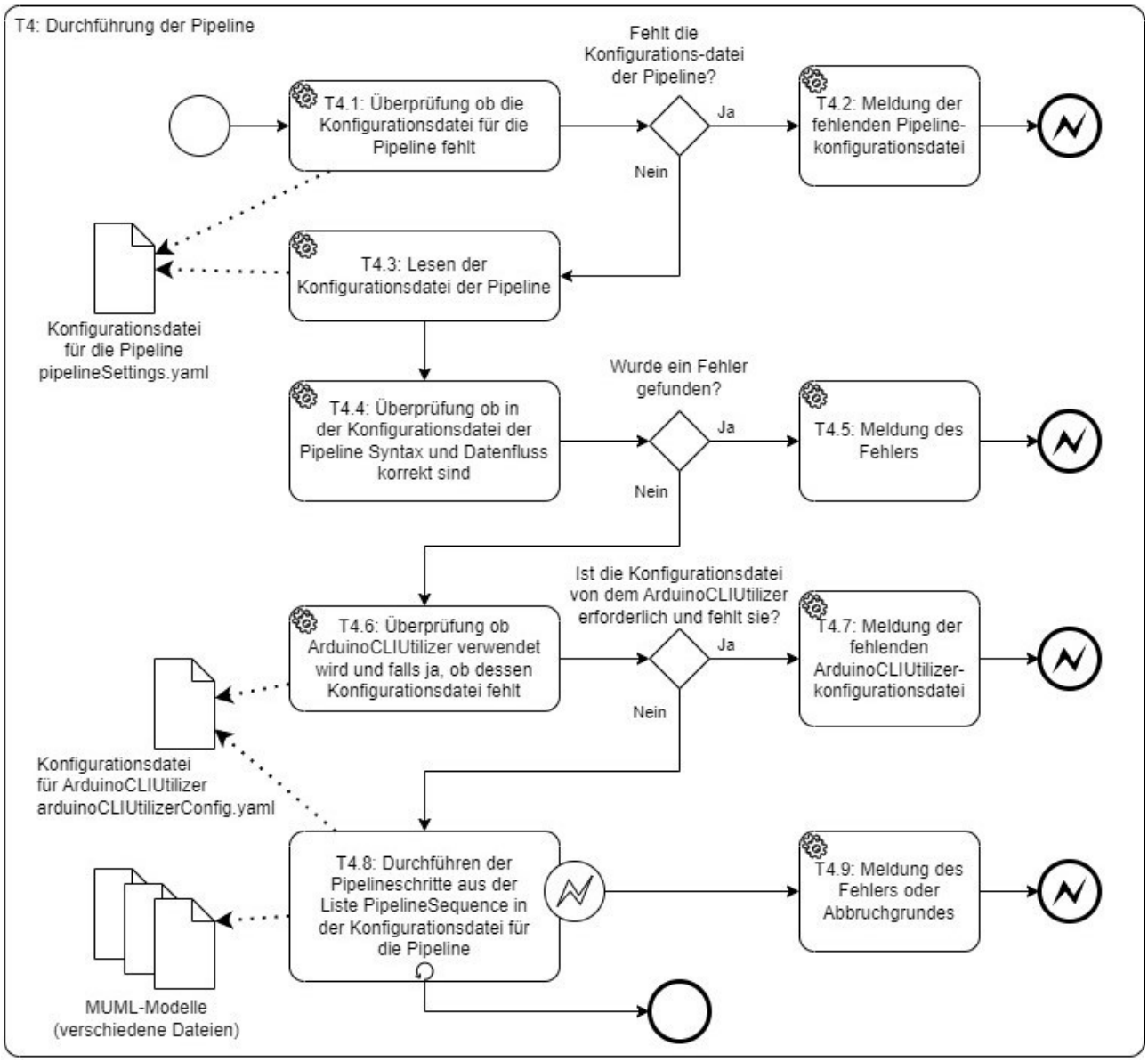
#


T2: Einstellen der Konfigurationen
Wegen Anpassungen, Fehler in T4, etc. zurück in T2?
Nein
Ja
Ist die Konfigurationsdatei für ArduinoCLIUtilizer vorhanden?
Ja
Nein
Fehlende Konfigurationsdatei für ArduinoCLIUtilizer
T2.1: Starten der Generierung der Konfigurationsdatei für ArduinoCLIUtilizer
Sonstiges
T2.2: Generierung der Konfigurationsdatei für ArduinoCLIUtilizer
Konfigurationsdatei für ArduinoCLIUtilizer arduinoCLIUtilizerConfig.yaml
Sind Anpassungen der Einstellungen für ArduinoCLIUtilizer erforderlich?
Nein
Ja
T2.3: Anpassen der Einstellungen für ArduinoCLIUtlizer
Unpassende Konfigurationen für ArduinoCLIUtilizer
Anpassung der Pipelinekonfigurationen erforderlich
Ist die Konfigurationsdatei für die Pipeline vorhanden?
Ja
Nein

#


Ist die Konfigurationsdatei für die Pipeline vorhanden?
Ja
Nein
T2.4: Starten der Generierung der Konfigurationsdatei für die Pipeline
T2.5: Generierung der Konfigurationsdatei für die Pipeline
Sind Anpassungen an den Einstellungen für die Pipeline erforderlich?
Nein
Ja
Sind die Seriennummern der Boards erforderlich und unbekannt?
Ja
Nein
T2.6: Anzeigenlassen der Informationen der angeschlossenen Boards
T2.6: Anzeigen der Informationen der angeschlossenen Boards
T2.7: Anpassen der Einstellungen für die Pipeline, z.B. Ändern der Sequenz der Pipelineschritte
Konfigurationsdatei für die Pipeline pipelineSettings.yaml

#



#

5.2: 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.

5.2.1: ANF 5.3: Automatische Tests der lokalen Arbeitskopie:
Diese Anforderung konnte nicht umgesetzt werden, weil keine Möglichkeit für das Testen von Arduinocode mit den verwendeten Roboterautos gefunden oder dafür angepasst werden konnte.
[(((TODO: Ich glaube, dass ich beim Evaluationsteil noch etwas hierzu hatte.)))]
Dies kann also nur manuell nach einem erfolgreichen Build-Prozess erfolgen.

5.2.2: 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 Versionsverwaltungsystemen empfohlen, hierfür etwas zu schreiben oder zu verwenden, das per Kommandozeilenschritt (siehe 5.2.1 [TODO: Verweis]) aufgerufen werden kann.

5.2.3: ANF 5.7: 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 MUML-Modelle) korrekt automatisch zusammengeführt werden können oder ob es z.B. bei den IDs zu Fehlern kommen kann.

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

5.3.1: Langlebige Flexibilität über mögliche Terminalaufrufe:
Es ist zum Zeitpunkt dieser Ausarbeitung unklar, was für Anforderungen später an die zu implementierende Pipeline auftreten könnten, also wurde ein eigener Schritt für das Nutzen des Terminals bzw. der Befehlszeile beschlossen. So können spätere Nutzer und Nutzerinnen fehlende Pipelinefähigkeiten z.B. durch eigene geschriebene Scripte und deren Aufrufe kompensieren.

5.3.2: 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 kompillierfähigen 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 5.5.3 [TODO: Verweis].

5.3.3: Datenfluss zwischen den eingetragenen Pipelineschritten:
Um die Übertragung von Daten von einem eingetragenen Pipelineschritt zu einem anderen zu ermöglichen, wurde beschlossen, dass die verschiedenen Pipelineschritttypen 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.

5.3.4: Vermeidung von Redundanz:
Um die Konfigurationen für die einzelne automatische Ausführung von den MUML-Werkzeugen für die Container Transformation, die Container Code Generation und die Component Code Generation 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.

5.3.5: 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 ArduinoCLIUtilizer und für die Pipeline selbst, generiert werden lassen können. Für die verschiedenen Schritte der Pipeline ist selbes auch in Form einer Textdatei mit allen Schritten und einer jeweiligen Beispielkonfiguration möglich.

[(((5.4 ist praktisch gesehen neu.)))]
5.4: Spezifische Pipelineschritte:
In diesem Unterkapitel werden aus dem Arbeitsablauf beispielhaft die sechs Pipelineschritte zu der Kontainer-Transformation, dem Eintragen der WLAN-Einstellungen, dem Kompilieren sowie den Hilfsschritten DeleteFolder, OnlyContinueIfFulfilledElseAbort und DialogMessage 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 5.2 , ANF 5.3 , ANF 5.4 , ANF 5.8 und ANF 5.9 aus der Perspektive der Nutzungsweise.

5.4.1: Modelltransformationen und Codegeneration: Beispiel: Kontainer-Transformation:

Zunächst kommt ein Beispiel aus den Arbeitsschritten und Pipelineschritten für die Codegeneration ausgehend von MUML-Modellen nach [https://github.com/SQA-Robo-Lab/MUML-CodeGen-Wiki/blob/main/user-documentation/main.md].

[TODO: Text und Screenshots zur manuellen Durchführung erst beim Tex-Zeug, weil es hier und jetzt noch keinen Sinn macht.]
Die Transformation von MUML-Modellen zu einer Modelldatei, die für einen Teil der Generation des unvollständigen Codes verwendet wird, erfolgt über die Kontainer-Transformation.
Diese erfolgt nach der Beschreibung aus [https://github.com/SQA-Robo-Lab/MUML-CodeGen-Wiki/blob/main/user-documentation/main.md, 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.

[Screenshot]

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

[Screenshot]

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.

[Screenshot]

Nach dem Klicken auf „Next >“ werden der Workspace als Dateisystem und der der Auswahl entsprechende Zielpfad angezeigt. Per Ordnerauswahl im Dateisystem oder per „Filesystem…“ kann der Generierungsort des Transformationsergebnisses angepasst werden. Für die Unterscheidung zwischen generiertem unvollständigem Code und verwendbarem Code wird als Zielort im Projektordner ein Ordner namens „generated-files“ angelegt und eingetragen.

[Screenshot]

Über den „Finish“-Knopf wird die Kontainer-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 Eingabe-Parameter „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 Eingabe-Parameter „middlewareOption“ repräsentiert die Middleware der Kommunikation bzw. den Ansatz für die Kommunikation. Hier wurde direkt die Zeichenkette für die automatische interne Auswahl von „MQTT and I2C Middleware Configuration“. 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 6.1 „Volle Unterstützung von neuer oder geänderter Hardware, Software und Bibliotheken“ [TODO: Verweis].
Der Eingabe-Parameter „muml_containerFileDestination“ dient für das Festlegen des Zielordes 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.

Der Ausgabe-Parameter „ ifSuccessful“ gibt in eine Variable aus, ob der Vorgang erfolgreich erfolgte. In diesem Fall wird die standardmäßig wiederholt überschriebene Variable „ifSuccessful“ verwendet.

5.4.2: 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:

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

Der Eingabe-Parameter „arduinoContainersPath“ steht für den den OrdnerPfad, in dem die Ordner für die Arduino-Code-Projekte für die verschiedenen Boards enthalten sind. In diesem Fall wird von der Variable „deployableFilesFolderPath“ der Pfad „deployable-files“ relativ zu dem Projektordner angegeben.
Im Eingabe-Parameter „ecuEnding“ wird der Namensfilter für die Ordnernamensenden eingetragen. Im Beispiel wird direkt per „CarCoordinatorECU“ die Auswahl auf „fastCarCoordinatorECU“ und „slowCarCoordinatorECU“ begrenzt.
Der Eingabe-Parameter „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 Eingabe-Parameter „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 Arduino-Sourcecodes das Passwort an öffentliche einsehbare Stellen gelangen kann.

Wie zuvor gibt der Ausgabe-Parameter „ ifSuccessful“ in eine Variable aus, ob der Vorgang erfolgreich erfolgte. Auch hier wird „ifSuccessful“ wiederverwendet.

5.4.3: 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 Ergebnisdateien 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 Eingabe-Parameter „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 Eingabe-Parameter „saveCompiledFilesNearby“ wird eingestellt, ob die beim Kompiliervorgang 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 5.4.3.3 [TODO: Verweis]).

Der Ausgabe-Parameter „ifSuccessful“ hat das typische Verhalten.
Über „resultMessage“ wird bei einem Fehlschlag mitunter auf den Ort der Datei mit dem Report über den Kompiliervorgang angegeben.

5.4.4: 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 zu des zu löschenden Ordners. Bei diesem Ausschnitt der Pipelinesequenz handelt es sich relativ zum Projektordner um „deployable-files/slowCarCoordinatorECU/slowCarCoordinatorECU.ino“.

5.4.5: 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 Eingabe-Parameter „condition“ übergebene Wahrheitswert „true“ ist, so wird der diesen liefernde Schritt als erfolgreich angesehen und die Pipelinedurchführung fährt fort. Ansonsten wird der im Eingabe-Parameter „message“ angegebene Text in einem Fenster angezeigt und die Pipelineausführung abgebrochen. Im Fall des Beispieles wird überprüft, ob dem 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.

5.4.6: 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.

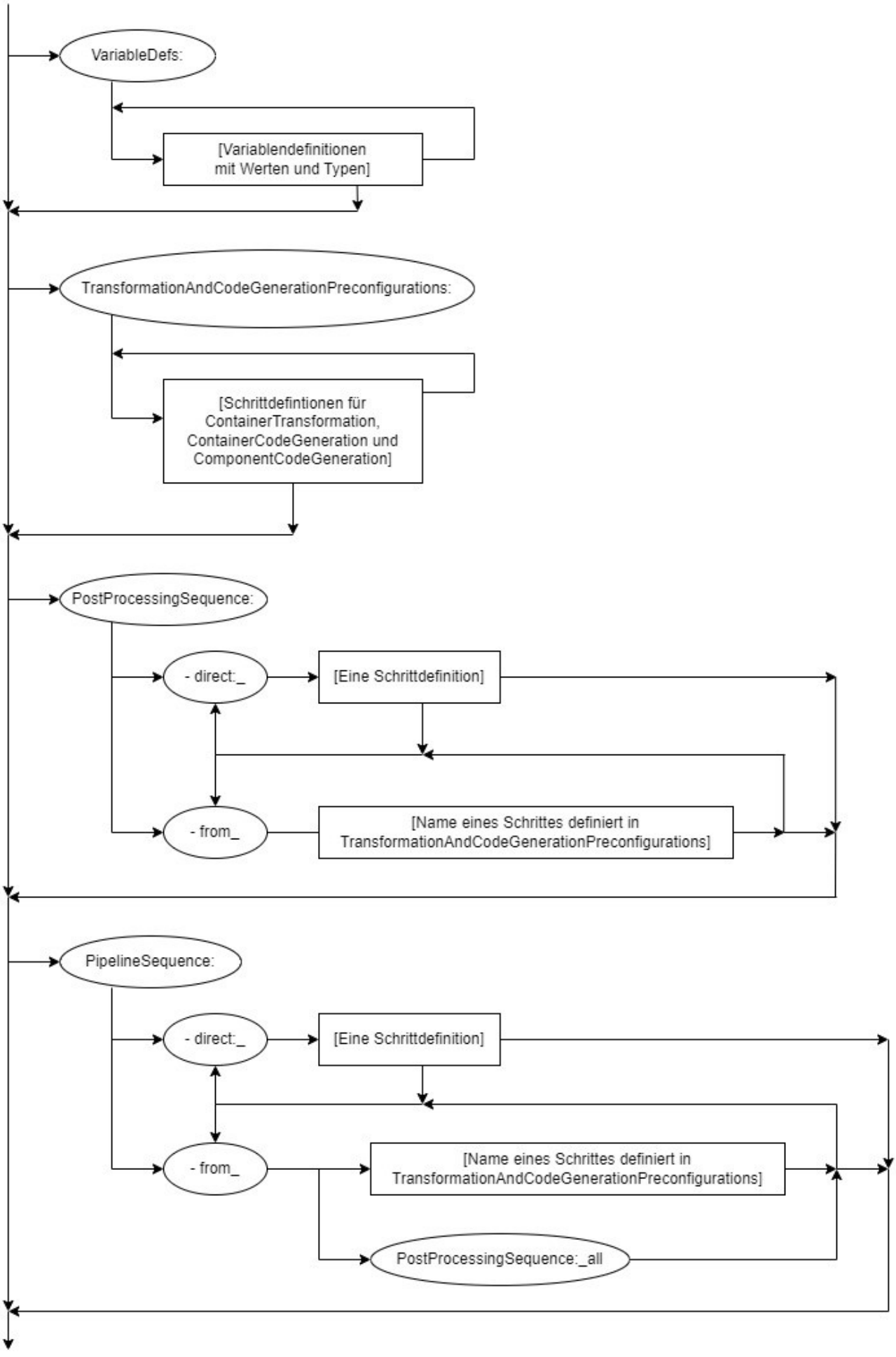
Als Pipelineschritt sieht der Eintrag hierfür beispielsweise wie folgt aus:

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

5.4.7: Pipelineschritte für Flexibilität und Langlebigkeit:
Es wurden für mehr Flexibilität und längerer Nützlichlichkeit 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 „Übersicht der Pipelineschritte“ [TODO: Verweis] aufgelistet. So ist beispielsweise der in 5.3.1 kurz umschriebene Pipelineschritt „TerminalCommand“ ein sehr nützlicher Befehl für die Einbundung der automatischenAusführung von Skripten und Programmen.

5.5: Aufbau der Konfigurationsdatei
Es wurden zunächst verschiedene Entwürfe für den Aufbau der Konfigurationsdatei „pipelineSettings.yaml“ aufgestellt, von denen im Rahmen einer Besprechung der im Folgenden beschriebene Ansatz ausgewählt wurde.

[TODO: Evtl. mehr Text]
Es sind in der Konfigurationsdatei vier Bereiche bzw. Mappings vorgesehen: VariableDefs für die anfänglichen Variablendefintionen, 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.
In den folgenden Unterabschnitten werden diese vier genannten Felder und ihre Konzepte der Reihe nach genauer beschrieben werden.


VariableDefs:
[Variablendefinitionen
mit Werten und Typen]
TransformationAndCodeGenerationPreconfigurations:
[Schrittdefintionen für
ContainerTransformation,
ContainerCodeGeneration und
ComponentCodeGeneration]
PostProcessingSequence:
- direct:_
[Eine Schrittdefinition]
- from_
[Name eines Schrittes definiert in
TransformationAndCodeGenerationPreconfigurations]
PipelineSequence:
- direct:_
[Eine Schrittdefinition]
- from_
[Name eines Schrittes definiert in
TransformationAndCodeGenerationPreconfigurations]
PostProcessingSequence:_all

5.5.1: 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 sind in dem Pipelinesystem die Typen „Number“, „String“, „Boolean“, „Path“, „FolderPath“, „FilePath“, „BoardSerialNumber“, „BoardIdentifierFQBN“, „ConnectionPort“, „WLANName“, „WLANPassword“, „ServerIPAddress“ und „ServerPort“ vorhanden. 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, nämlich für das dynamische Definieren und Initialisieren das Festlegen des Variablentyps und beim Überschreiben 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 Mappings, die jeweils die folgende Syntax befolgen:

<Variablentyp> <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 5.5.5: „Schrittdefinitionen“):

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

Hierbei erfolgt die Typenzuweisung intern und automatisch.
Zur Vereinfachung beim Schreiben der Konfigurationen wurde entschieden, dass das Überschreiben eines Variableninhalts auf dem selben 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.

5.5.2: 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 Container Code Generation und die Component Code Generation enthalten. Für das Vermeiden von Redundanz können innerhalb von PostProcessingSequence und PipelineSequence Bezüge auf diese eingetragen werden, die wiederum jeweils an ihrer Stelle das Ausführen des entsprechenden Werkzeugs mit den jeweiligen Einstellungen darstellt.

5.5.3: 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.

5.5.4: PipelineSequence - Die Sequenz für die eigentliche Pipeline:
Die Piepelinesequenz 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 Redundanz vermieden werden kann.

5.5.5: 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:

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

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 Ausgabe-Parametern 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 „direkt“, 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 Kontext keine Typenverwechslung passiert.
Syntaktisch beschrieben sehen die Eingabeparametersequenzeinträge jeweils folgendermaßen aus:

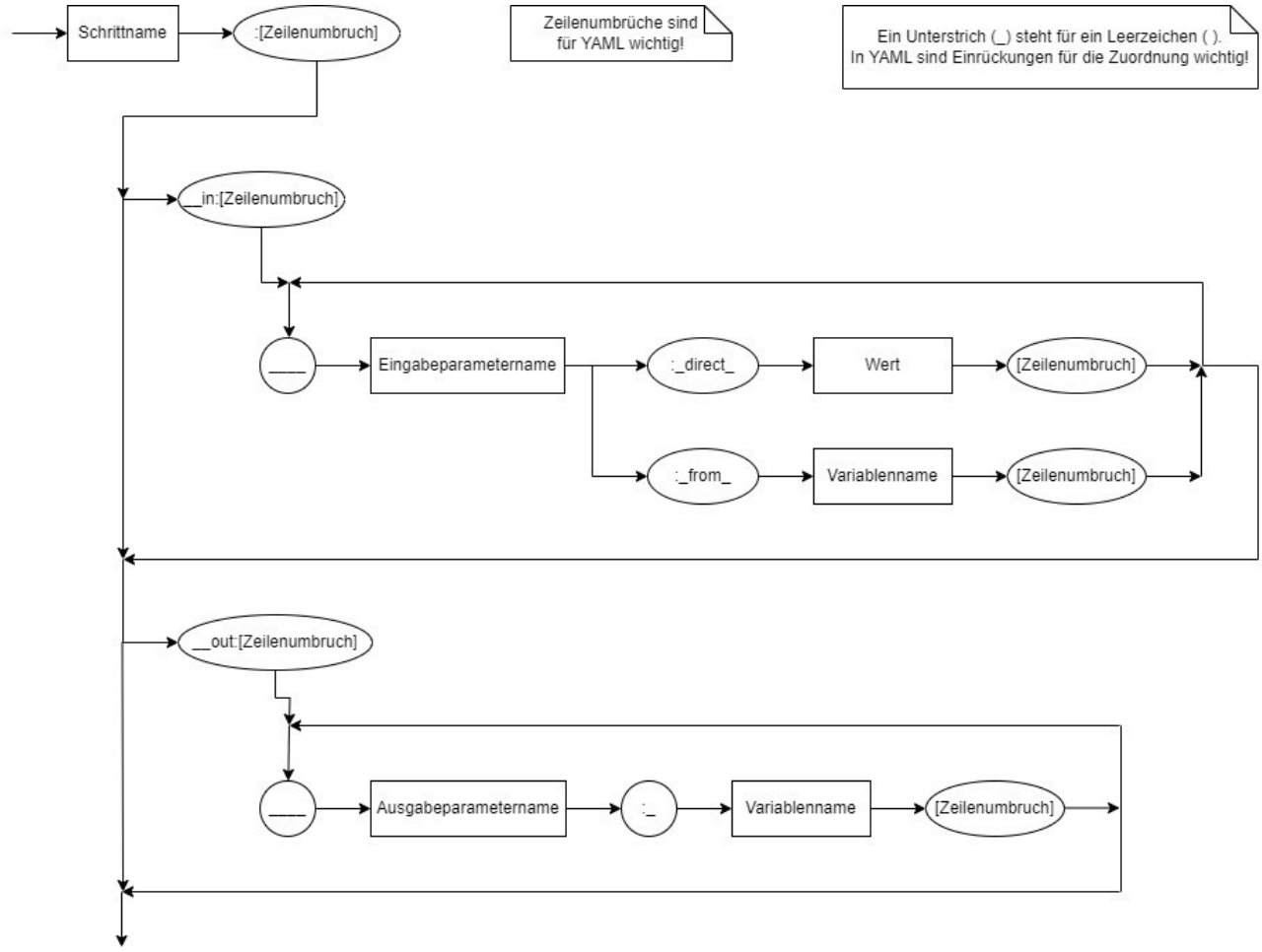
<Eingabeparametername>: direct: <VariableValue>

oder

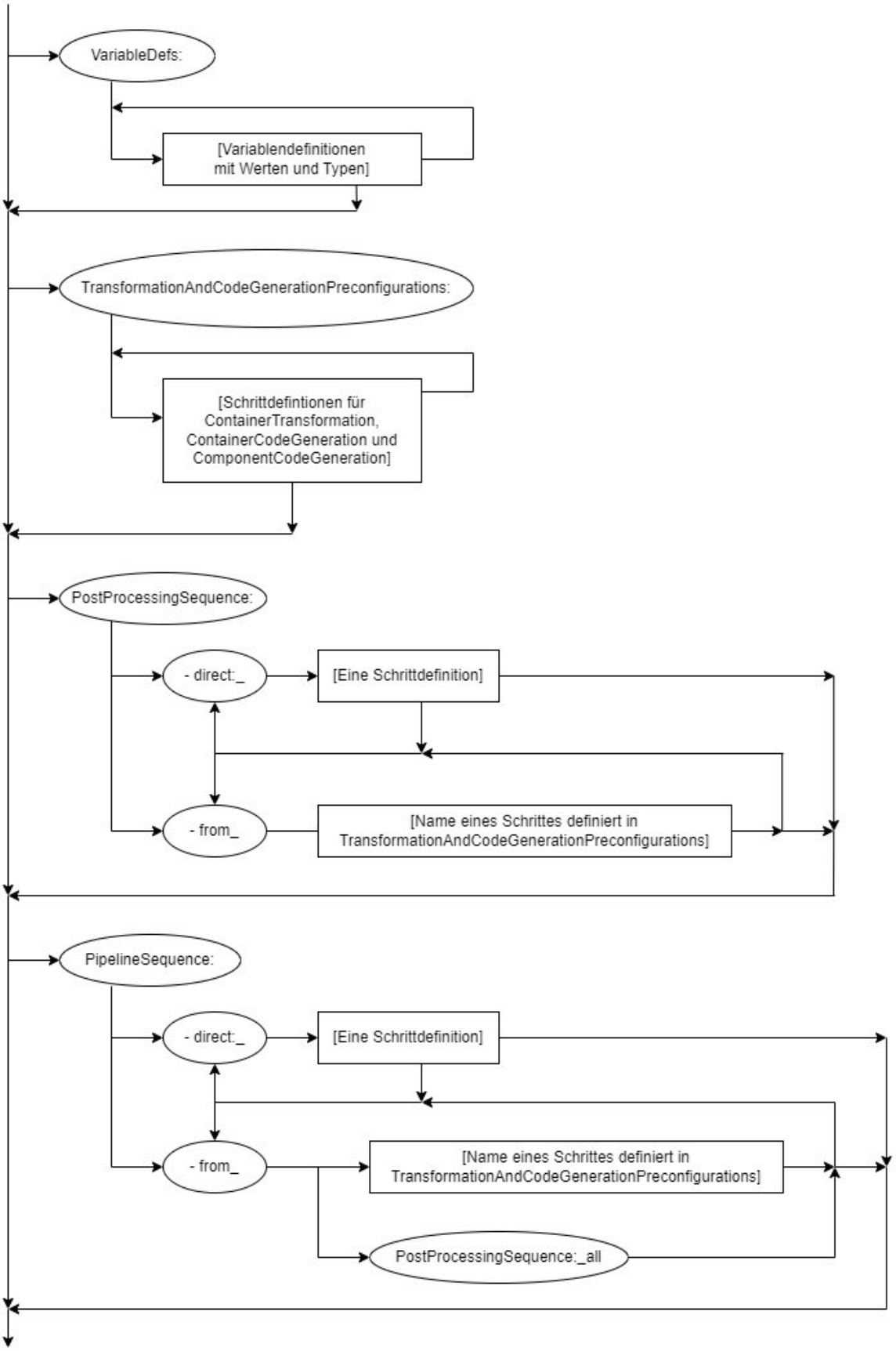
<Eingabeparametername>: from: <VariableName>

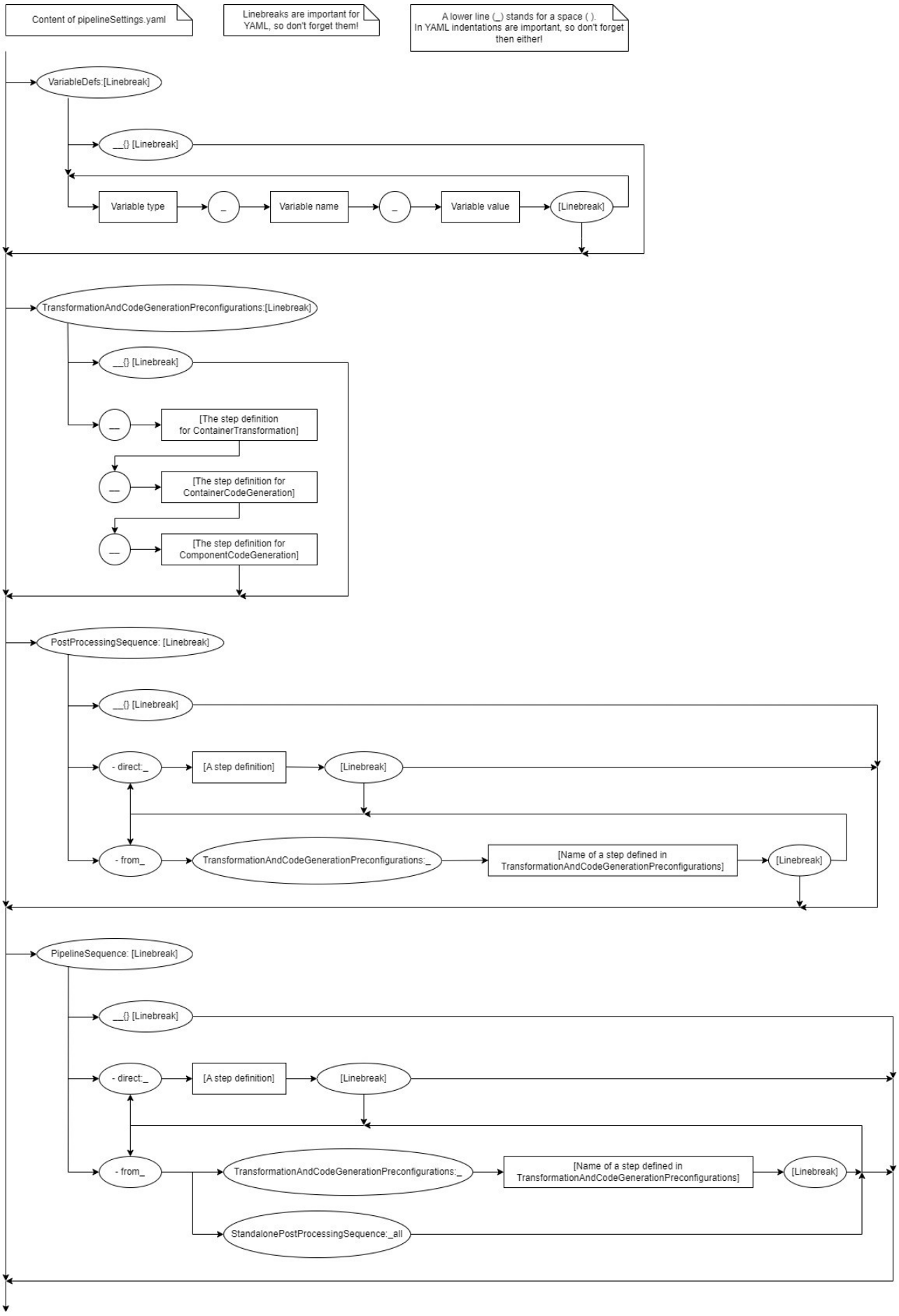
Die Ausgabeparameter werden ebenfalls als eine Sequenz aus einzelnen Mappings eingetragen. Diese bestehen jeweils aus dem entsprechenden Parameternamen und dem Namen einer Variablen, in der der jeweilige Ergebniswert gespeichert werden soll. Wie in [TODO: Nummer von 5.5.1: VariableDefs – Variablen] beschrieben wurde, wird hierbei automatisch je nach Fall eine neue Variable aufgenommen oder die Inhaltsüberschreibung einer bekannten Variable durchgeführt.

Grafisch dargestellt sieht es folgendermaßen aus:



Die Konfigurationsdatei sieht dabei wie folgt aus:


VariableDefs:
[Variablendefinitionen
mit Werten und Typen]
TransformationAndCodeGenerationPreconfigurations:
[Schrittdefintionen für
ContainerTransformation,
ContainerCodeGeneration und
ComponentCodeGeneration]
PostProcessingSequence:
- direct:_
[Eine Schrittdefinition]
- from_
[Name eines Schrittes definiert in
TransformationAndCodeGenerationPreconfigurations]
PipelineSequence:
- direct:_
[Eine Schrittdefinition]
- from_
[Name eines Schrittes definiert in
TransformationAndCodeGenerationPreconfigurations]
PostProcessingSequence:_all


Content of pipelineSettings.yaml
Linebreaks are important for YAML, so don't forget them!
A lower line (_) stands for a space ( ).
In YAML indentations are important, so don't forget then either!
VariableDefs:[Linebreak]
__{} [Linebreak]
Variable type
_
Variable name
_
Variable value
[Linebreak]
TransformationAndCodeGenerationPreconfigurations:[Linebreak]
__{} [Linebreak]
__
[The step definition for ContainerTransformation]
__
[The step definition for ContainerCodeGeneration]
__
[The step definition for ComponentCodeGeneration]
PostProcessingSequence: [Linebreak]
__{} [Linebreak]
- direct:_
[A step definition]
[Linebreak]
- from_
TransformationAndCodeGenerationPreconfigurations:_
[Name of a step defined in TransformationAndCodeGenerationPreconfigurations]
[Linebreak]
PipelineSequence: [Linebreak]
__{} [Linebreak]
- direct:_
[A step definition]
[Linebreak]
- from_
TransformationAndCodeGenerationPreconfigurations:_
[Name of a step defined in TransformationAndCodeGenerationPreconfigurations]
[Linebreak]
StandalonePostProcessingSequence:_all