Hochschule Neu-Ulm
University of Applied Sciences

Technische Hochschule Ulm

Bachelorarbeit

im Bachelorstudiengang

Wirtschaftsingenieurwesen

an der Hochschule für angewandte Wissenschaften Neu-Ulm

Konzept und Implementierung eines Orchestration Agents in Node-RED

Erstkorrektorin: Prof. Dr.-Ing. Lisa Ollinger

Zweitkorrektor: Prof. Dr.-Ing. Hartwig Baumgärtel

Verfasser: Jakob Weber (Matrikel-Nr.: 275668)

Babenhauserstraße 30, 89294 Oberroth

Thema erhalten: 01.04.2023

Arbeit abgegeben: 28.07.2023

Inhalt

Eigenständigkeitserklärung

Diese Arbeit wurde von mir selbständig verfasst. Es wurden nur die angegebenen Quellen und Hilfsmittel verwendet. Alle wörtlichen und sinngemäßen Zitate sind in dieser Arbeit als solche kenntlich gemacht.

28.07.2023 

Datum **Jakob Weber**

Abkürzungsverzeichnis

SkE	Skill-based Engineering
OA	Orchestration Agent
IoT	Internet of Things
RFID	Radio frequenzy identification
SoA	Service-oriented Architecture
OPC UA	Open Platform Communications Unified Architecture
MQTT	Message Queuing Telemetry Transport
QoS	Quality of Service
ISO	International Standardization Organization
OSI	Open System Interconnection
JSON	JavaScript Object Notation
MAS	Multiagentensystem
BDI	Belief, Desire, Intentions
SPS	Speicherprogrammierbare Steuerung
FUP	Funktionsplan
PLC	Programable Logic Controller
TCP	Transmission Control Protocol
IP	Internet Protocol
SCL	Superordinate Control Logic
DT	Digital Twin
GUI	Graphical User Interface

1. Einleitung

1.1 Motivation

Mit fortschreitender Globalisierung lag für Produktionsanalagen in den letzten Jahren der Fokus vor allem auf der Massenproduktion. Produkte sollten möglichst effizient in großer Stückzahl produziert werden. Mit der aktuellen Tendenz zu einem Anstieg an individuellen Kundenspezifikationen für Produkte und geringeren Produktlebenszyklen entstehen neue Herausforderungen.[1] Die Steigerung der Variantenvielfalt eines Produkts erhöht signifikant den Produktionsaufwand. Konventionelle automatisierte Systeme auf Basis von Schrittketten sind nicht in der Lage flexibel auf derartige Veränderungen zu reagieren. Unter dem Begriff Industrie 4.0 werden Ansätze konzipiert, um derartige Herausforderungen zu lösen. Ein wichtiger Aspekt im Rahmen der Industrie 4.0 ist die Konzeption modularer Produktionseinheiten, diese können anforderungsspezifisch kombiniert werden.[2] Die Module verfügen über definierte Fähigkeiten, die abgerufen werden können. Ein wichtiger Ansatz zur Erstellung dieser Module ist das Skill-based Engineering (SkE). Das Konzept basiert auf der herstellerunabhängigen Definition der Fähigkeiten einer Ressource als Skill.[3] Für die Erstellung von Produktionsabläufen ist es erforderlich, dass die zusammenhangslosen Skills durch eine höhere Instanz angeordnet werden. Diese Instanz wird als Orchestration Agent (OA) bezeichnet. Ein OA ermöglicht die flexible und individuelle Orchestrierung der Skills für jedes Produkt. Somit bilden OAs einen Ansatz zur signifikanten Steigerung der Flexibilität und Modularität eines Systems. Im Rahmen dieser Arbeit soll ein OA über Node-RED umgesetzt werden, der in der Lage ist erstellte Skills zu orchestrieren und somit individuelle Prozesse an einem fischertechnik Demonstrator auszuführen.

1.2 Ziele

Die Zielsetzung dieser Arbeit ist es über Node-RED einen OA zu erstellen, der fähig ist Skills zu orchestrieren. Die Implementierung eines OAs soll in Kombination mit der Generierung des SPS-Programms nach dem Ansatz des SkE die konventionelle Schrittkette zur Ablaufsteuerung von Prozessen ersetzen und somit den Grad der Modularität und Flexibilität steigern. Die Arbeit umfasst die Erstellung verschiedener Konzepte, die zum Orchestrierungsprozess beitragen. Die geeigneten Konzepte werden

[1] Vgl. Lasi et al. 2014, S. 261
[2] Vgl. Bundesministerium für Wirtschaft und Klimaschutz 2023
[3] Vgl. Jungbluth et al. 2023, S. 163

anschließend am fischertechnik Industrie 4.0 Demonstrator umgesetzt. Die Arbeit beinhaltet die folgenden drei Hauptaspekte bei der Erstellung eines OAs:

1. Wie werden die Skills des SPS-Programms in Node-RED registriert?
2. Wie wird die Steuerungslogik des OAs abgelegt?
3. Wie erfolgt die Kommunikation und welche Schnittstellen sind erforderlich?

2. Stand der Technik

2.1 Industrie 4.0

Industrie 4.0 bildet die vierte industrielle Revolution und gilt häufig als Oberbegriff für fortschrittliche Technologien wie Internet of Things (IoT), Big Data, Cloud Computing oder künstliche Intelligenz.[4] Ein wichtige Rolle spielt die steigende Modularität und Flexibilität der Anlagen und Systeme. Die extreme Form davon bildet die Losgröße eins, die die Fertigung von nur einem Teil einer Produktart in einem Zyklus beschreibt. Die Vision ist es, Fertigungssysteme zu konzipieren, die in der Lage sind, Produkte in der Losgröße eins effizient fertigen zu können. Hierfür ist entlang der gesamten Wertschöpfungskette eine vertikale und horizontale Vernetzung der Systeme erforderlich.[5] Kernaspekt der Industrie 4.0 ist die intelligente Vernetzung der Komponenten. Der verbesserte Informationsaustausch ermöglicht die Umsetzung neuer Ansätze in der Produktion. Durch eine bessere Vernetzung entlang der Wertschöpfungskette eines Produktes ist eine bessere Planung der Maschinenauslastung möglich. Zudem verwirklichen die Ansätze der Industrie 4.0 den Aufbau modularer Fabriken. Die Module können nach Art der Anforderung der Prozesse, spezifisch angeordnet werden. Die steigende Flexibilität und Modularität ermöglichen die Befriedigung individueller Kundenforderungen bereits in geringen Stückzahlen.[6]

2.2 Internet of Things

Internet of Things beschreibt ein Konzept bei dem sogenannte smart objects in einem Netzwerk untereinander kommunizieren, auf Veränderungen in der Umwelt reagieren und selbstständig agieren.[7] Smart objects werden als physische Objekte, die durch zusätzliche Konnektivität in einem digitalen Netzwerk kommunizieren können, definiert. Sie sind in der Lage über Sensoren und/oder Aktoren mit der physischen Welt zu interagieren und diese Daten anschließend in einem digitalen Netzwerk auszutauschen. Ein triviales Beispiel sind radio frequenzy identification (RFID) Chips, die dauerhaft in ein physisches Objekt integriert werden. Somit kann das Objekt von Lesegeräten über einen digitalen Identifikationscode erfasst werden.[8] IoT hat in vielen Bereichen Anwendungsmöglichkeiten: Im Alltag sind smart objects wie Smartphones und Smartwatches bereits äußerst populär. Im industriellen Kontext können durch das Konzept Produktionsprozesse tiefgreifender automatisiert und

[4] Vgl. Suleiman et al. 2022, S. 2
[5] Vgl. Lasi et al. 2014, S. 261
[6] Vgl. Bundesministerium für Wirtschaft und Klimaschutz 2023
[7] Vgl. Madakam/Ramaswamy/Tripathi 2015, S. 165
[8] Vgl. González García et al. 2017, S. 8

prozesssicherer gestaltet werden. Durch eine Verknüpfung der zu bearbeitenden Produkte und der Werkzeugsätze mit individuellen digitalen Kennungen kann an jeder Bearbeitungsstation das Soll-Werkzeug mit dem eingelegten Werkzeug automatisch abgeglichen werden. Dadurch können Fehler bei manuellen Umrüstvorgängen eliminiert werden.[9]

2.3 Service-oriented Architecture

Service-oriented Architecture (SoA) ist eine Architektur, die aus der Nutzung und Bereitstellung von Services besteht. In diesem Kontext kann der Begriff Service als eine Art Dienstleistung oder Fähigkeit, eine bestimmte Aufgabe zu lösen, verstanden werden. Im Rahmen einer SoA sind zwei Entitäten verankert: Teilnehmer, die Services bereitstellen, sog. Service Provider und diejenigen, die Services nutzen, sog. Service User. Diese Kategorisierung ist nicht exklusiv, ein Teilnehmer kann gleichzeitig Service Provider und Service User sein.[10] Eine SoA verknüpft die Provider und User über ein Netzwerk. Eine weitere Komponente ist das Interface bzw. Verzeichnis, das die Aufgabe hat, verfügbare Services zu registrieren und diese für die User sichtbar und benutzbar zu machen. Die Kombination aus dem Verzeichnis und der Netzwerkverknüpfung ermöglicht die Nutzung der Services nicht nur am physischen Ort des Service Providers, sondern im gesamten Netzwerk.[11] Die Verwendung von SoA ermöglicht eine größere Modularität des Systems: User können flexibel die Services benutzen, die zum aktuellen Zeitpunkt benötigt werden. Bei Veränderungen des Anwendungsfalls können weitere Services beansprucht werden oder bisherige Services ausgelassen werden.[12] Im Kontext der Automatisierung von Fertigungssystemen ermöglicht das SoA-Konzept die Implementierung von plug-and-produce Funktionalitäten unter den einzelnen Komponenten.[13]

2.4 Skill-based Engineering

SkE ist ein Ansatz, der mit den Prinzipien der Modularität und Flexibilität von Produktionssystemen vereinbar ist. Ein Skill bezeichnet die Fähigkeit von Systemkomponenten auf der untersten Ebene.[14] Skills existieren in unterschiedlichen Komplexitätsniveaus. Die einfachste Form sind basic skills, die z.B. die Aktivierung eines Aktors ermöglichen. Composite Skills sind komplexer aufgebaut und bestehen aus zwei

[9] Vgl. 2015, S. 7–12
[10] Vgl. Blanco/Kotermanski/Merson 2007, S. 1
[11] Vgl. Blanco/Kotermanski/Merson 2007, S. 3
[12] Vgl. Laskey/Laskey 2009, S. 101–102
[13] Vgl. Dorofeev 2020, S. 159
[14] Vgl. Dorofeev 2020, S. 159

oder mehreren basic Skills, die zusammen eine Fähigkeit bilden. Durch Kombination der basic Skills für zwei Lichtschranken und eines Motors kann beispielsweise ein composite Skill zur Bewegung eines Förderbands generiert werden. Die composite Skills können über die Reihenfolge und Prozessparameter angeordnet werden, dass gewünschte Services realisiert werden.[15] Der Hintergrund von SkE ist die Modularisierung der starren Schrittketten. Die Untergliederung in mehrere Skills ermöglicht die flexible Orchestrierung von Skills zu kompletten Prozessabläufen. Dieser Aspekt ermöglicht die Wiederverwendbarkeit von Programmbestandteilen, in der initialen Entwicklungsphase und bei späteren Erweiterungen des Programms.[16] Durch die Möglichkeit der Orchestrierung von Skills durch eine höhere Instanz können Teile der Steuerungen aus der untersten Prozessebene ausgelagert werden. Die Ansteuerung der Komponenten durch Kommunikation Standards wie Open Platform Communications Unified Architecture (OPC UA) vereinfacht somit die vertikale Kommunikation.[17] Ein wichtiger Bestandteil des SkE ist die Herstellerunabhängigkeit. Die modulare Kombination verschiedener Ressourcen zu einem System basiert auf der Grundlage eines herstellerunabhängigen Modells. Die Fähigkeiten einer Ressource werden als Skill modelliert und der Ressource zugeordnet. Jede Ressource des Systems verfügt über Skills, die angeboten und über Kommunikationsprotokolle wie MQTT oder OPC UA aufgerufen werden können.[18]

2.5 OPC UA / MQTT

OPC UA ist ein Datenaustausch-Standard, der eine plattformunabhängige Kommunikation zwischen Geräten verschiedener Hersteller ermöglicht. OPC UA basiert auf einer SoA und bietet durch seine Plattformunabhängigkeit die Möglichkeit zum Informationsaustausch komplexer Daten von Mikrocontrollern bis hin zu Cloud-services. OPC UA bietet neben dem klassischen Server–Client Modell die Möglichkeit der Verwendung des Publish-Subscribe Modells. Während beim ersten Modell versucht wird die Information direkt an einen Client zuzustellen, wird beim Publish-Subscribe Modell die Information an einen Broker veröffentlicht. Das Server-Client Modell basiert auf dem Zusammenspiel von Requests (Anfragen) und Responses (Antworten), somit wird ein Austausch nur durch das Senden einer Request initialisiert. Im Gegensatz dazu erfolgt die Bereitstellung von Informationen beim Publish-Subscribe Modell durchgehend. Allerdings erfolgt der Austausch nicht direkt mit dem Teilnehmer, der an der Information interessiert ist, sondern wird an einen Broker veröffentlicht. Berechtigte Subscriber können anschließend den Broker abonnieren und

[15] Vgl. Jungbluth et al. 2023, S. 164
[16] Vgl. Dorofeev 2020, S. 159–160
[17] Vgl. Zimmermann et al., S. 1101–1103
[18] Vgl. Jungbluth et al. 2023, S. 163–164

somit die individuell relevanten Informationen erhalten. Die Publisher veröffentlichen somit konstant Informationen, unabhängig davon, ob eine Request vorliegt. Die Subscriber können konstant auf die relevanten Daten zugreifen, ohne auf eine Response der Subscriber warten zu müssen.[19] Eine visuelle Veranschaulichung der Modelle kann der Abb.1 entnommen werden.

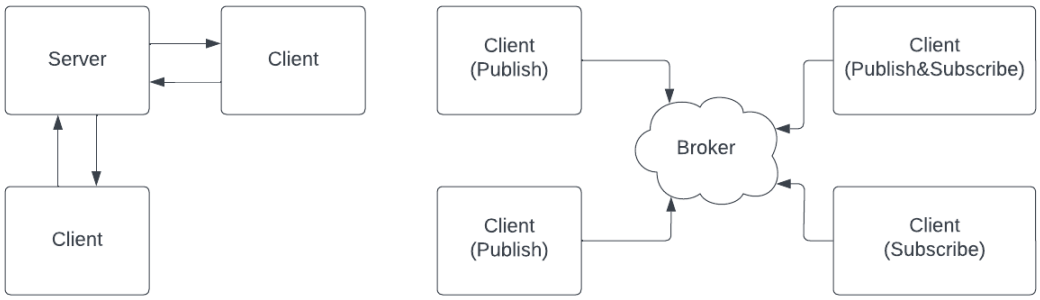


Abbildung 1: Schematische Funktionsweise von Server/Client und Publish/Subscribe Modellen

Das MQTT-Protokoll verwendet ausschließlich das Publish-Subscribe Modell. MQTT steht ursprünglich für Message Queuing Telemetry Transport. MQTT ist ausgelegt für Netzwerke mit begrenzten Ressourcen, wie geringe Bandweite und Energieverbrauch. Es zeichnet sich durch seine einfache Implementierung aus.[20] MQTT bietet die Möglichkeit Daten bei der Kommunikation an eine Topic zu koppeln. Somit können Daten kategorisiert werden und einfacher über die Topic abgerufen werden. Die Kommunikation über MQTT ist in drei Level unterteilt. Hierbei handelt es sich um die Quality of Service (QoS), die vorgibt, wie wahrscheinlich Nachrichten empfangen werden. Die qualitativ niederwertigste Variante ist QoS0. Hierbei wird die Nachricht einmal abgeschickt und es erfolgt keine Empfangsbestätigung des Empfängers. Bei dieser Variante ist die Zustellung nicht garantiert und Nachrichten können unbemerkt verloren gehen. QoS1 garantiert über Empfangsbestätigungen, dass die die Nachricht mindestens einmal zugestellt wird. Bei dieser Variante kann es, auf Grund einer fehlenden Empfangsbestätigung, durch erneut gesendete Nachrichten zu Duplikaten beim Empfänger führen. QoS1 bietet einen Kompromiss aus Effizienz und garantierter Zustellung. QoS2 ergänzt das QoS1-Modell um eine zweite Rückmeldeschleife und garantiert damit, dass jede Nachricht nur genau einmal empfangen wird. Diese Garantie erfordert mehr Daten bei den Transaktionen und ist langsamer als die anderen Varianten.[21]

[19] Vgl. OPC Foundation
[20] Vgl. Mishra/Kertesz 2020, S. 201074–201075
[21] Vgl. HiveMQ 2015

Während beide Kommunikationsprotokolle für den Einsatz im Bereich IoT geeignet sind, grenzt sich OPC UA durch einen komplexeren und umfassenderen Aufbau ab. Faktoren wie Sicherheit und Skalierbarkeit qualifizieren OPC UA für komplexere und umfangreiche Systeme, während MQTT sich durch simple Implementierung vor allem für den Einsatz in kleineren Systemen bzw. Testsysteme eignet. Der unterschiedliche Aufbau beider Kommunikationsprotokolle wird im ISO/OSI Schichtenmodell der International Standardization Organization (ISO) deutlich (siehe Abb. 2 und 3.). Das Open System Interconnection (OSI) Modell unterteilt den Kommunikationsablauf in sieben Schichten. Den einzelnen Schichten werden hierbei spezifische Aufgaben des Kommunikationsablaufs zugeordnet. Bei den Schichten handelt es sich um die Bitübertragungs-, Sicherungs-, Vermittlungs-, Transport-, Sitzungs-, Darstellungs-, und Anwendungsschicht.

Das OSI-Modell beginnt mit der Bitübertragungsschicht als Schicht 1. In der ersten Schicht werden grundlegende Elemente definiert, um eine physische Verbindung herzustellen, wie z.B. der Kabel Typ. Für beide Protokolle wird als Übertragungsmedium Ethernet verwendet. Darauf aufbauend erfolgt die Sicherungsschicht, die den Zugriff auf das Netzmedium bzw. die Fehlererkennung und Fehlerkorrektur zur Aufgabe hat. Die Vermittlungsschicht beinhaltet unter anderem die Bereitstellung von netzwerkübergreifenden Adressen bzw. das Routing. Die Bereitstellung einer gesicherten Prozess-zu-Prozess Verbindung auf den Endknoten wird in der Transportschicht erreicht. Mit Ausnahme der Sicherungsschicht ist der Aufbau der Protokolle inklusive der Transportschicht identisch. Darauf aufbauend sorgt die Sitzungsschicht für die Prozesskommunikation zwischen zwei Systemen. In der Darstellungsschicht werden die Datenstrukturen und Datenformate festgelegt. Während OPC UA auf diesen Ebenen über verschiedene Varianten verfügt, werden die Schichten für das MQTT-Protokoll nicht benötigt (siehe Abb.2 und 3.) Über die Abbildungen wird der Unterschied im Umfang und der Komplexität der Protokolle deutlich. Die finale Anwendungsschicht bietet eine Input/Output Schnittstelle für die gekoppelte Anwendung. Bei der Kommunikation zwischen zwei Anwendungen erfolgt der Durchlauf über alle Schichten. Ausgangspunkt ist Schicht sieben des Senders. Anschließend werden die Schichten abwärts, bis Schicht eins durchlaufen, wo die physische Übertragung zum Empfänger stattfindet. Dort ist die Ablaufsequenz der Schichten entgegengesetzt.[22]

[22] Vgl. Küveler/Schwoch 2003, S. 446–448

ISO/OSI Modell			
7	Anwendungsschicht	UA Application	
6	Darstellungsschicht	UA Binary	UA XML
5	Sitzungsschicht	UA TCP, UA Secure Conversation	SOAP/HTTP, WS-SecureConversation
4	Transportschicht	TCP (RFC 793)	
3	Vermittlungsschicht	IP (RFC 791)	
2	Sicherungsschicht	MAC (IEEE 802.3)	
1	Bitübertragungsschicht	Ethernet (IEEE 802.3)	

Abbildung 2:OPC UA ISO/OSI Modell

ISO/OSI Modell		
7	Anwendungsschicht	MQTT Protocol
6	Darstellungsschicht	N/A
5	Sitzungsschicht	N/A
4	Transportschicht	TCP (RFC 793)
3	Vermittlungsschicht	IP (RFC 791)
2	Sicherungsschicht	IP (RFC 894)
1	Bitübertragungsschicht	Ethernet (IEEE 802.3)

Abbildung 3: MQTT ISO/OSI Modell

2.6 Node-RED / JSON

Node-RED ist ein open-source Programmierwerkzeug der OpenJS Foundation. Die Anwendung basiert auf der Verwendung von sog. *flows*. Das bedeutet, dass Informationen entsprechend der Verbindungen zwischen Objekten im Netzwerk fließen. Diese Objekte werden in Node-RED als *nodes* bezeichnet.[23] Node-RED basiert auf der Programmiersprache JavaScript und speichert Informationen in Form von JSON objects, das für JavaScript Object Notation steht. JSON ist ein kompaktes Datenformat, das Daten über key-value Paare speichert. Der Key ist hierbei immer als string notiert, während value als string, number, boolean, null, object oder array vorliegen kann. Die Notation ist kompakt, da in einem JSON object mehrere Objekte beschrieben werden können, die eine Vielzahl an key-value Paaren aufweisen können. Beispielsweise können so alle relevanten Daten, in Form von key-value Paaren, für eine Wettervorhersage in ein JSON object integriert werden. Ein key-value Paar ist hier beispielsweise: “weather“: “cloudy“, wobei “weather“ der

[23] Vgl. OpenJS Foundation & Contributors

key ist und “cloudy“ entrspricht der value.[24] Node-RED verfügt in der Standardversion über eine Palette an *nodes* für bestimmte Anwendungen, darunter fallen *Input-* und *Debug-nodes*. Spezifische Fähigkeiten können individuell in *function nodes* programmiert werden. Zudem gibt es durch das open-source Prinzip *node* Vorlagen, die von der Community erstellt und publiziert wurden. Mithilfe von Node-RED können somit beispielsweise Inputs aus IoT-fähigen Sensoren über MQTT verarbeitet werden und Anweisungen an IoT-fähige Aktoren ausgegeben werden. Node-RED wird im Browser ausgeführt und alle Informationen können über das integrierte Dashboard ausgegeben werden. [25] Die Abb. 4 zeigt exemplarisch ein simples Anwendungsbeispiel von Node-RED: Die *nodes* sind seriell und parallel von links nach rechts verbunden. Die *node* „5“ setzt bei der Aktivierung durch Klicken *„msg.payload“* gleich 5. Die nachfolgende *node* „function 1“ bearbeitet den *payload-Wert* entsprechend des darin notierten JavaScript Codes des Erstellers. Während die grüne *Debug node* den *payload-Wert* im Debug-Fenster ausgibt, erwirkt die *node* „Anzeige“ das Erscheinen des *payload-Werts* auf dem Dashboard.

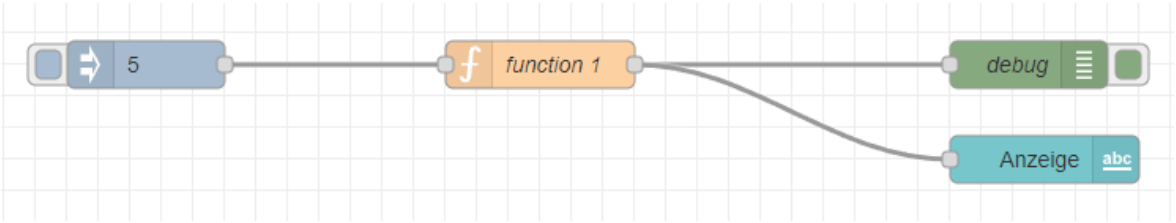


Abbildung 4: Beispiel eines Node-RED flows

Standardmäßig geben alle *nodes* ihren Output als *„msg.payload“* aus. Die *Message* ist die Variable, die entlang des *flows* sequenziell gelesen und verarbeitet wird. Werden mehrere *Messages* in einem *flow* verwendet, werden diese standardmäßig identisch im *flow* verarbeitet, ansonsten können *Messages* durch *filter nodes* herausgefiltert werden. Neben der Verwendung von *Messages* kann der Status individueller *nodes* gelesen und verwendet werden, um eine ereignisbasierte Steuerung zu generieren. Eine weitere Möglichkeit des Node-RED-internen Datenaustauschs sind globale Variablen. Diese können über Methoden deklariert werden und in allen *flows* gelesen werden. Relevant ist der Einsatz globaler Variablen vor allem bei einem Informationsaustausch abweichend der Richtung des *flows*.

[24] Vgl. Ooms 2014, S. 1–3
[25] Vgl. Clerissi et al. 2018, S. 1

2.7 Agenten

In der Softwaretechnik werden Agenten als Einheiten bezeichnet, die durch Inputs und Outputs die Umgebung wahrnehmen und darauf Einfluss nehmen können. Häufig werden Agenten in Form von Multiagentensystemen (MAS) eingesetzt. Agenten werden als eigenständige Software implementiert und sollen autonom vorgegebene Ziele erreichen bzw. definierte Aufgaben ausführen.[26] Agenten sollen folgende Eigenschaften besitzen: Sie sollen in deren zugewiesenen Aufgabenbereichen autonom handeln. Des Weiteren sollen sie nicht nur in der Lage sein auf Veränderungen in der Umwelt entsprechend zu reagieren, sondern pro-aktiv zu handeln, um deren vorgegebenen Aufgaben korrekt zu erfüllen. Zuletzt sollen Agenten, vor allem in MAS, in der Lage sein zu kommunizieren.[27] Agenten werden in drei Klassen aufgegliedert und unterscheiden sich diesbezüglich in der Komplexität, der Breite und Tiefe der Fähigkeiten. In diesem Kontext wird unterschieden in Reaktive Agenten, Adaptive Agenten und Kognitive Agenten. Reaktive Agenten sind vergleichsweise simpel aufgebaut. Sie erkennen Veränderungen in der Umgebung und reagieren darauf mit den vorgegebenen Regeln. Dieser Agententyp verfügt somit nur über die Fähigkeit auf Situationen zu reagieren. Adaptive Agenten unterscheiden sich davon insofern, dass sie über interne Zustände verfügen und somit die Reaktion anpassen können, um einen optimalen Output zu generieren. Kognitive Agenten verfügen über mehr Informationen und Kernstrukturen. Hier können beispielsweise Ziele definiert werden. Kognitive Agententypen sind in der Lage den Umweltzustand zu bewerten und anschließend abzuwägen welche Aktionen erforderlich sind, um das definierte Ziel bestmöglich zu erfüllen.[28]

Insbesondere die Kombination mehrerer Agenten zu MAS erweist sich als geeigneter Ansatz für die Lösung komplexer Automatisierungssysteme. Relevant ist hier die Fähigkeit komplexe Systeme in kleinere Teilgebiete aufzuteilen. Die zugewiesenen Agenten sind dann in der Lage zu kommunizieren und kooperativ die Prozesse zu gestalten.[29] Im Kontext dieser Arbeit soll der Agent hauptsächlich die Aufgabe der Orchestration von Skills erfüllen. Diese Art von Agenten wird als OA bezeichnet. Unter Orchestration wird die Steuerung und Anordnung von Prozessen eines Systems in einer zentralen Instanz verstanden. Das Konzept der zentralen Orchestrierung durch einen OA hat den Vorteil, dass das System dadurch flexibler, modular erweiterbar und hardwareunabhängiger wird. Bei der

[26] Vgl. Bommer, S. 4–5
[27] Vgl. Wooldridge/Ciancarini 2001, S. 2–3
[28] Vgl. Bommer, S. 7–10
[29] Vgl. Jennings 2000, S. 282–283

Verwendung eines OAs werden ein oder mehrere komplexere Prozesse in simplere Aufgaben aufgegliedert.[30]

Unter dem Begriff OA werden zwei verschiedene Ansätze von Agenten verfolgt. Die Unterschiede liegen in den verschiedenen Leveln der Orchestrierung. Im Kontext von MAS wird ein OA als Agent definiert, der andere teilnehmenden Agenten des Systems orchestriert. Ein OA gleicht die Fähigkeit anderer Agenten mit den Anforderungen der zu bewältigenden Aufgabe ab und weist entsprechend die korrekten Agenten der Aufgabe zu.[31]

In einem anderen Ansatz ist das Level der Orchestration abweichend definiert. Hierbei wird der OA als Instanz definiert, die Skills zu einem Prozess anordnet. Es erfolgt ein Abgleich welche Ressourcen eine Aufgabe erfüllen können bzw. welche Ressourcen über die erforderlichen Skills verfügen. Entsprechend der vorgegebenen Prozesse vergleicht der OA die verfügbaren Skills der Ressourcen und ordnet diese optimal dem Zielprozess zu.[32] Der wesentliche Unterschied liegt somit in der Art der Instanzen, die Subjekt des Orchestrationsprozesses sind. Für den Anwendungsfall wird der OA nach dem zweiten Ansatz implementiert und soll Skills zu Prozessen orchestrieren.

[30] Vgl. Viroli/Denti/Ricci 2007, S. 227–228
[31] Vgl. Pisarić et al. 2020, S. 471–475
[32] Vgl. Lober et al., S. 4–5

3. Bestandsaufnahme

3.1 TIA-Portal / e!Cockpit

Bei dem für die Arbeit verwendeten Demonstrator handelt es sich um die fischertechnik Lernfabrik 4.0. Das System ist in zwei separate Bereiche, auf zwei physisch abgetrennten Tischen, unterteilt. Besonders ist der unterschiedliche Aufbau beider Teilsysteme hinsichtlich der Hardware und Software. Ein Teilsystem basiert auf der Verwendung einer Wago Speicherprogrammierbaren Steuerung (SPS). Aufgrund der Hardware erfolgt die Programmierung des System über, das auf Codesys basierende, e!Cockpit von Wago. Das zweite Teilsystem verwendet eine Siemens S7-1500 SPS. Die Programmierung dieses Systems erfolgt über das TIA-Portal von Siemens. Aufgrund dieser Unterschiede ist die Erstellung zweier Programme notwendig. Beide Systeme können im aktuellen Zustand nur separat Prozesse ausführen und nicht kooperieren. Für die Zusammenfügung beider Systeme, trotz Hardware- und Software-Unterschiede, ist eine herstellerunabhängige Instanz erforderlich. Die Erstellung der Programme hinsichtlich der Skills und Betriebsarten erfolgt universell über das Prinzip des Funktionsplans (FUP). Im Rahmen der Arbeit wird die SPS-Programmierung für das Wago-Teilsystem behandelt.

3.2 Bestandteile

Wie bereits im vorherigen Abschnitt erwähnt, ist der Demonstrator in zwei Bereiche untergliedert. Das über die Wago SPS gesteuerte Teilsystem ist gliederbar in 5 Komponenten: Das Subsystem verfügt über zwei separate Förderbänder, die über je eine integrierte Stempeleinheit verfügen. Eine weitere Komponente ist die U-Straße, die eine Komposition aus folgenden Bestandteilen ist: Drei Förderbänder mit einer Lichtschranke, ein Förderband mit zwei Lichtschranken, zwei Abschieber und zwei Bearbeitungsstationen. Zusätzlich stehen dem Subsystem zwei Kräne zur Verfügung. Die Reichweite des ersten Krans beinhaltet jeweils einen Übergabepunkt der beiden Förderbänder und einen Übergabepunkt mit dem anderen Kran. Der zweite Kran ist fähig, neben dem Übergabepunkt mit dem ersten Kran, die beiden Übergabepunkte der U-Straße anzusteuern. Die Vakuum-Sauggreifer beider Kräne werden jeweils durch einen Kompressor und ein Magnetventil gesteuert. Die fünf Komponenten werden in drei Subsysteme untergliedert. Die Subsysteme eins und zwei verfügen über je ein Förderband mit integrierter Stempelmaschine und einen Kran. Das Subsystem drei besteht aus der U-Straße. Die Untergliederung in Subsysteme verbessert die Übersichtlichkeit des Systems, da z.B. die Subsysteme zwei und drei aufgrund der selben Komponenten schwer

unterscheidbar sind. Die Abb. 5 zeigt den Aufbau des Wago Teilsystems und die Zuordnung der Komponenten in die Subsysteme.

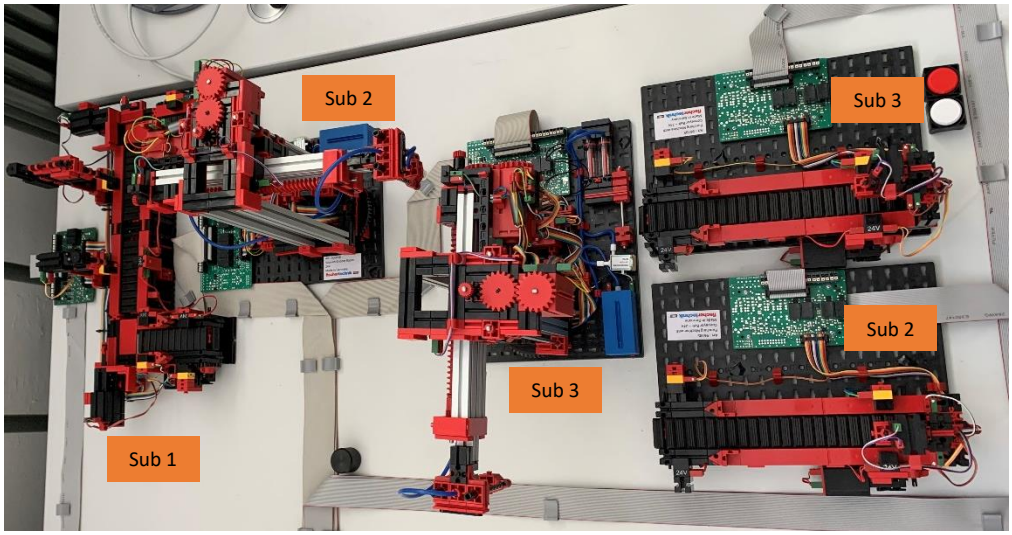


Abbildung 5: Aufbau des Wago-Teilsystems des Demonstrators

3.3 Sensorik / Aktorik

Der Bereich des Demonstrators, der über die Wago SPS angesteuert wird, wird weiter untergliedert. Im e!Cockpit ist das System in drei Subsysteme aufgeteilt. Subsystem 1 steuert die Bestandteile der U-Straße und Subsystem 2 steuert einen der Kräne und eine Kombination aus Förderband und Stempelmaschine. Das dritte Subsystem beinhaltet den anderen Kran und die zweite Förderband-Stempelmaschine Kombination. Dementsprechend sind die Sensoren und Aktoren den drei Subsystemen untergeordnet. Grundsätzlich werden zwei verschiedene Varianten von Sensoren verwendet: Lichtschranken und Taster. Die verwendeten Lichtschranken bestehen aus jeweils einer LED und einem Phototransistor. Der Output der Lichtschranken ist im ununterbrochenen Zustand auf 1 und im unterbrochenen Zustand auf 0. Die verwendeten Taster sind so eingestellt, dass diese im aktivierten Zustand den Output 1 haben und im deaktivierten Zustand den Output 0 ausgeben. Unter den Aktoren sind die häufigsten Komponenten Motoren. Während diese in den Förderbändern der Subsysteme 2 und 3 bidirektional agieren, werden in den Förderbändern der U-Straße unidirektionale Motoren eingesetzt. Die restlichen Motoren können in beiden Drehrichtungen verwendet werden. Zwei Komponenten, die nur bei den Kränen verwendet werden, sind der Kompressor und das Magnetventil. Diese werden benötigt, um das für den Sauggreifer notwendige Vakuum zu erzeugen. Die Funktionsweise wird in der Abb. 6 abgebildet. Eine Übersicht über alle Inputs und Outputs der Subsysteme 1 bis 3 kann dem Anhang 1 im Anhang entnommen werden.

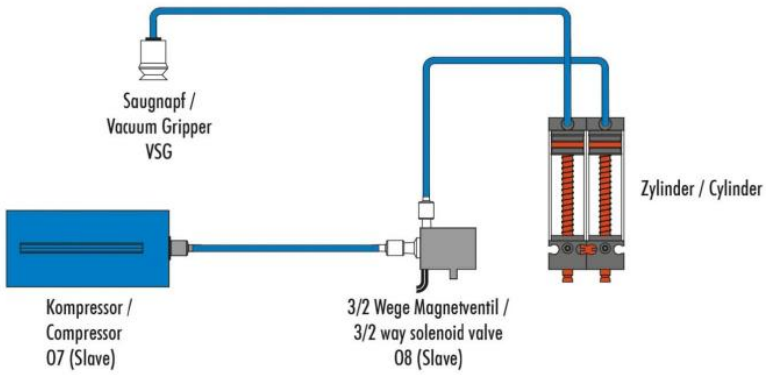


Abbildung 6: Pneumatik Schaltplan des Vakuum Greifers

3.4 Bekannte Services

Die einzelnen Bestandteile des Demonstrators sind in der Lage verschiedene Services auszuführen. Ein Service wird in diesem Kontext als eine Komposition aus einem oder mehreren Skills definiert. Ein Service kann aus einer einfachen Translation entlang einer Achse bestehen oder aus einer Bewegung im dreidimensionalen Raum mit variablen Streckenparameter. Aus vorherigen Arbeiten am verwendeten Demonstrator sind bereits Services definiert (siehe Tabelle 1).

Tabelle 1: Übersicht der bestehenden Services

Transport Conveyor	Sub2/Sub3
Transport Conveyor + Stamping	Sub2/Sub3
Transport Crane clockwise	Sub2/Sub3
Transport Crane counterclockwise	Sub2/Sub3
Transport Conveyor	Sub1
Transport Conveyor + Milling	Sub1
Transport Conveyor + Drilling	Sub1
Transport Conveypr + Milling + Drilling	Sub1

Das Modul, bestehend aus einem Förderband und einer Stempelmaschine (siehe Abb. 7), verfügt über zwei grundlegende Services. Der erste Service beinhaltet nur den Transport von der Übergabestelle an das Förderbandende und den anschließenden Rücktransport zum Ausgangspunkt. Der zweite Service ergänzt den ersten Service um die Durchführung des Stempelprozesses. Hierbei wird zwischen den beiden Transport-skills, durch Auf- und Ab- Bewegungen des Stempels der Stempelprozess durchgeführt. Dieser kann beliebig oft durchgeführt werden. Das eben angesprochene Modul ist in Subsystem 2 und in

Subsystem 3 vorzufinden. Die Module sind identisch und verfügen somit über die gleichen Services.

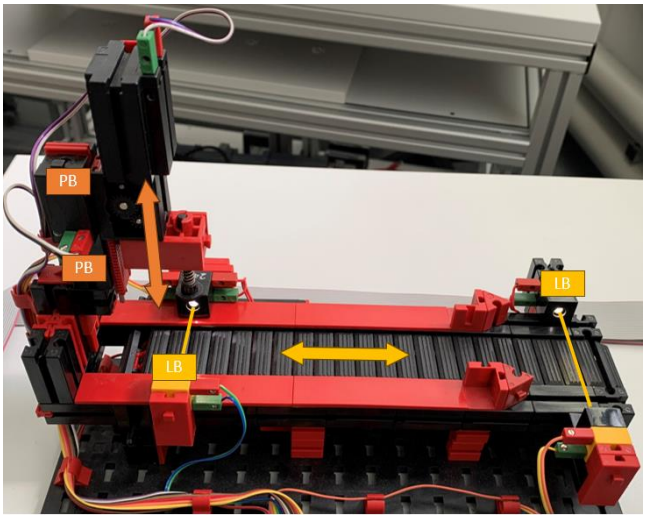


Abbildung 7: Übersicht über die Bestandteile der Ressource Conveyor

In Subsystem 2 und 3 ist ein weiteres Modul identisch. Der Kran verfügt in beiden Systemen über den grundlegend gleichen Aufbau (siehe Abb. 8). Die einzigen Abweichungen sind verschiedene Drehrichtungen, Fahrzeiten und die Position eines Tasters. In vorhergehender Arbeit wurden zwei grundlegende Services für die Kräne definiert: Der Transport eines Objekts in Drehrichtung des Uhrzeigers und der Transport entgegen dem Uhrzeigersinn. Diese Services sind deutlich komplexer, da Bewegungen entlang der X-Achse, Y-Achse und Z-Achse stattfinden. Zusätzlich wird die Erzeugung eines Vakuums für den Sauggreifer benötigt. Die Kategorisierung in zwei Services ist ausreichend, da alle anderen Transportwege mit abweichenden Punkten sich nur hinsichtlich der Fahrzeiten unterscheiden.

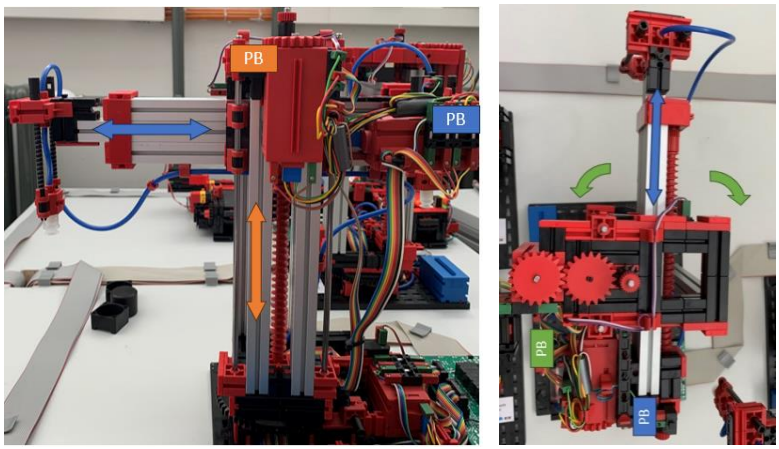


Abbildung 8: Übersicht über die Bestandteile der Ressource Crane

Für andere Fähigkeiten, wie beispielsweise Puffervorgänge, müssen andere Services implementiert werden, da diese aber für diese Arbeit nicht beachtet werden, sind die

existierenden Services ausreichend. Das Subsystem 1 besteht aus dem Modul der U-Straße (siehe Abb.9). Die U-Straße bildet eine unidirektionale Angliederung von Förderbändern ab. Mithilfe von Abschiebern können Objekte vom Startpunkt bis zum Endpunkt transportiert werden. In das System sind zwei Bearbeitungsstationen integriert. Die verfügbaren Services unterscheiden sich grundsätzlich aus der Komposition der Bearbeitungsstationen. Somit ergeben sich vier Varianten für die Abläufe: Dier erste Option ist der reine Transport des Objekts vom Startpunkt zum Endpunkt, ohne Einsatz der Bearbeitungsstationen. Die zweite Variante besteht aus dem Einsatz beider Bearbeitungsstationen, zusätzlich zum Transport. Aufgrund der unidirektionalen Verwendung der Förderbänder sind nur zwei weitere Möglichkeiten verfügbar. Entweder kann zusätzlich zum Transport nur die erste Bearbeitungsstation oder nur die zweite Bearbeitungsstation verwendet werden.

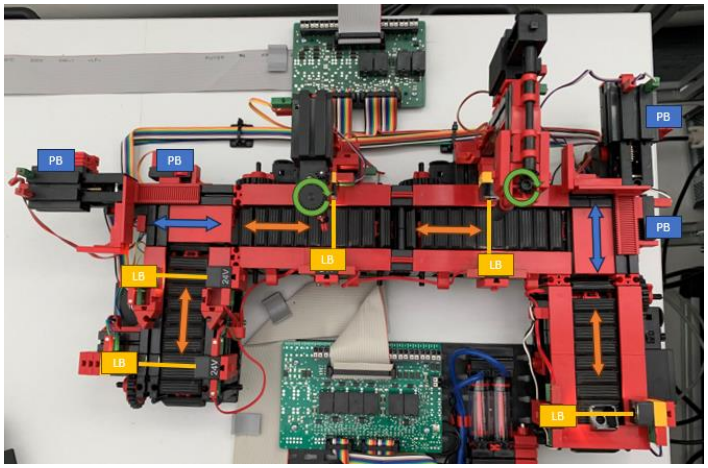


Abbildung 9: Übersicht über die Bestandteile der Ressource U-Straße

Die aufgelisteten Services wurden in den vorhergehenden Programmen starr als Schrittketten definiert. So konnte durch das manuelle Abrufen mehrerer Services ein Prozessablauf abgebildet werden. Im Kontext dieser Arbeit werden die Services allerdings in dieser Form nicht mehr existieren. Im Rahmen der Arbeit werden die Prozessabläufe durch Anreihung der Skills gebildet und nicht durch die übergeordneten Services. Die Services existieren somit nur in Form der vom OA zusammengestellten Skills und nicht mehr als eigenständige Komponenten. Unter einem Service wird folgend nicht die starre Aneinanderreihung von Skills verstanden, sondern der übergeordnete Aufruf an das System, beispielsweise ein Teil zu produzieren. Der Unterschied liegt hierbei darin, dass zum Zeitpunkt des Service Calls, der Service noch nicht zusammengestellt wurde, sondern noch aus losen Skills besteht, die anschließend orchestriert werden. Der Service Call impliziert einen definierten Endzustand bzw. Ziel, der eigentliche Ablauf wird allerdings nicht absolut durch den Service Call bestimmt.

3.5 Netzwerk Architektur

Die aktuelle Netzwerkarchitektur besteht aus drei Elementen. Hierzu gehört der Programable Logic Controller (PLC). Der PLC beinhaltet zum aktuellen Stand des Programms die einzelnen Skills, die das System ausführen kann, die Services, die die Skills zu Prozessen kombinieren und die Zuweisung der Skills zu den Inputs und Outputs. Die weiteren Bestandteile sind das Bedienpanel und das e!Cockpit. Beide Komponenten dienen zur Steuerung des Systems. Hierüber werden die Betriebsarten gewechselt und die gewünschten Services aktiviert. Eine schematische Übersicht über die aktuelle Architektur kann der Abb. 10 entnommen werden. Erkennbar ist, dass der Großteil des Systems innerhalb der PLC stattfindet. Die Aspekte des SkE sind umgesetzt, allerdings können ohne Auslagerung der Service-Kompositionen die Vorteile des SkE nicht vollständig genutzt werden: Die Services sind starr in das Programm als Schrittkette implementiert und müssen einzeln aktiviert werden. Sollen weitere Abläufe abgebildet werden, müssen diese wieder in das Programm programmiert werden.

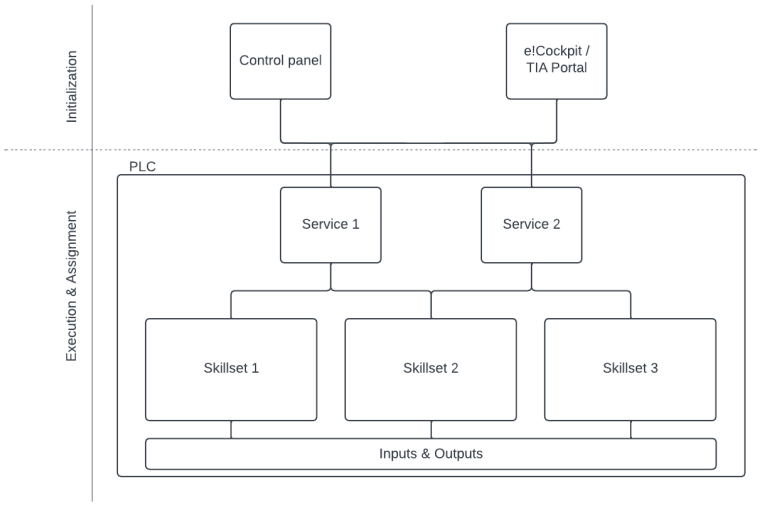


Abbildung 10: Übersicht über die aktuelle Architektur des Systems

Über eine Ethernet Verbindung wird das SPS-Programm von einem PC auf die SPS geladen. Die Kommunikation erfolgt über Modbus Transmission Control Protocol (TCP) nach der IEC 61158. Das TCP/IP Modell ist eine Alternative des ISO/OSI Modells und besteht aus vier Schichten. Die unterste Schicht, Netzzugang entspricht dem Ethernet-Kabel. Das Internet Protocol (IP) wird für die Internet-Schicht verwendet, die im ISO/OSI Modell der Vermittlungsschicht entspricht. Die Transportschicht verwendet das Protokoll TCP. Die höchste Schicht entspricht der Modbus Anwendungsschicht.

4. Konzeption des Systems

4.1 Konzept eines Orchestration Agent

Ein OA ist eine Kategorie der Softwareagenten, die die spezifische Aufgabe der Orchestrierung von Aufgaben und Prozessen hat. Die unter Kapitel 2.6. (Agenten) angesprochenen Eigenschaften eines Softwareagenten sollen, wie im Folgenden beschrieben, umgesetzt werden. Ein OA soll fähig sein auf seine Umgebung reagieren zu können. Inputs aus der Umgebung sollen nach korrekter interner Verarbeitung Reaktionen über die Effektoren auslösen. Die Proaktivität orientiert sich bei Agenten an der Aufgabe, die diesen zugewiesen wird.[33] Bei OAs handelt es sich um die optimale Zusammenstellung von Subprozessen, um den Ablauf für das vorgegebene Ziel bestmöglich abzubilden.[34] Proaktivität entspricht in diesem Kontext die bereitstehenden Ressourcen respektive der Aufgabe des OA zu analysieren und anschließend abzuwägen, welche Variante der Auswahlmöglichkeiten die Aufgabe optimal erfüllt. Der Einsatz des OAs in einem Multi-Agenten System erfordert die Befähigung zur Kommunikation. Die Zuweisung von anderen Agenten kann nur dann optimal ausgeführt werden, wenn ein Austausch über die Daten aller Agenten stattfindet. Im Kontext der „social ability“ kann der OA seine Aufgabe bestmöglich erfüllen, wenn alle Teilnehmer ihre Informationen bezüglich der aktuellen Verfügbarkeit, Durchlaufzeit, Kapazität, etc. einheitlich veröffentlichen. Für die Klassifizierung als Softwareagent müssen die genannten Aspekte autonom ablaufen. Das bedeutet, dass der OA im normalen Betriebsmodus kein menschliches Einwirken benötigt, um seine Aufgabe zu erfüllen.[35]

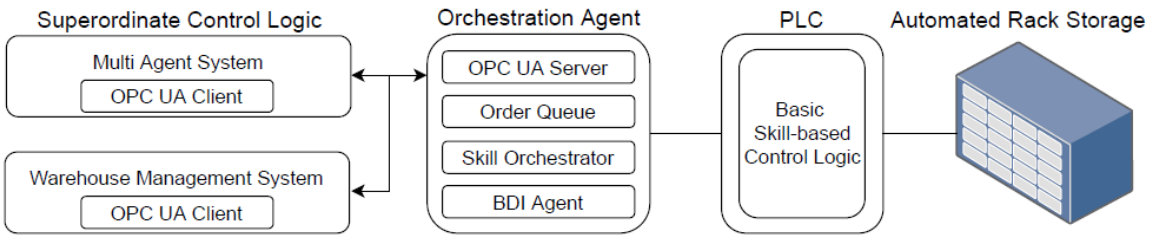


Abbildung 11: Implementierung eines OAs in ein automatisiertes Lager

Der beispielhafte Aufbau eines OAs bzw. dessen Implementierung kann der Abb. 11 entnommen werden. Nach Lober et al. wird die Implementierung eines OAs in ein automatisiertes Kleinteilelager betrachtet. Die PLC wurde in diesem Kontext so aufbereitet, dass diese über Skills, gemäß dem SkE, verfügt. Zentrale Einheit zwischen der PLC und

[33] Vgl. Wooldridge, S. 26–27
[34] Vgl. Pisarić et al. 2020, S. 469–470
[35] Vgl. Wooldridge, S. 26–29

der Superordinate Control Logic (SCL) ist der OA, der sich aus folgenden Komponenten zusammensetzt. Zur Kommunikation wird ein Server-Client Modell nach OPC UA verwendet, wobei der OA als OPC UA Server implementiert wird. Zur Pufferung von eingehenden Calls verfügt der OA über eine Order Queue, dadurch können eingehende Service Calls nach der internen Logik abgearbeitet werden. Der Skill Orchestrator verfügt über die Information, welche Skills für die entsprechenden Service Callls notwendig sind und in welcher Reihenfolge diese anzuordnen sind. In diesem Beispiel verfügt der Orchestrator über einen Belief, Desire and Intentions (BDI) Agenten oder zumindest agentenähnlichem Verhalten. Dieser Softwareagent ermöglicht dem Orchestrator in einem MAS zu verhandeln.[36] Der Durchlauf eines Prozesses in diesem System kann vereinfacht folgendermaßen beschrieben werden (siehe Abb. 12). Der Durchlauf wird initiiert durch den Call eines Services durch die SCL an den OA. Anschließend wird der Service in die Order Queue eingegliedert, da die Möglichkeit besteht, dass zum aktuellen Zeitpunkt noch ein Service ausgeführt wird. Im nächsten Schritt, ordnet der OA die notwendigen Skills mit den entsprechenden Parametern so an, dass der gewünschte Service entsteht. Sobald die PLC meldet, dass der letzte Skill des vorhergehenden Service beendet wurde, kann der erste Skill des neu orchestrierten Services gestartet werden. Hierbei werden die entsprechenden Parameter bezüglich beispielsweise Fahrzeiten an die PLC weitergegeben. Nachfolgend werden sequenziell alle Skills, die dem aktuellen Service zugeordnet werden, ausgeführt. Die Ausführung ist allerdings nicht auf eine rein sequenzielle Reihenfolge eingeschränkt, sondern kann in Teilen parallele Exekutionen beinhalten.[37]

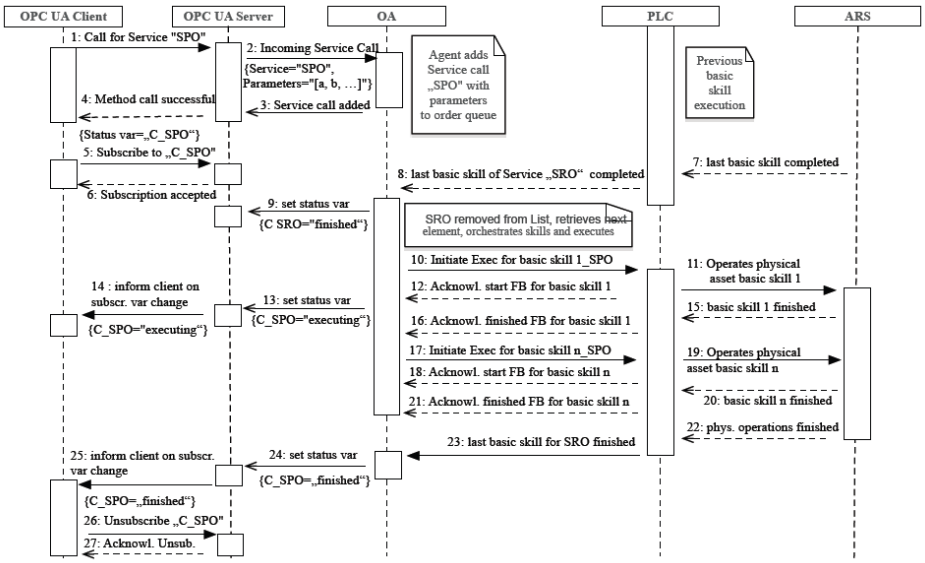


Abbildung 12: Übersicht über die Abläufe zwischen den Teilnehmern während eines Orchestrierungsprozesses

[36] Vgl. Lober et al., S. 4–5
[37] Vgl. Lober et al., S. 5

Dieses Konzept kann um die Integration eines Digital Twins (DT) erweitert werden. Der DT ist in diesem Fall ein digitales Abbild der PLC in Echtzeit. Somit können darüber alle Informationen der SPS wie die Zustände aller Aktoren und Sensoren visualisiert und verarbeitet werden. Zudem können die Aktionen des OAs über die Ausgänge des DTs an die SPS kommuniziert werden. Der DT beinhaltet zusätzlich eine Knowledge Base mit den Informationen, welche Ressourcen über welche Skills verfügen. Eine beispielhafte Visualisierung dieser Erweiterung kann der Abb. 13 entnommen werden.

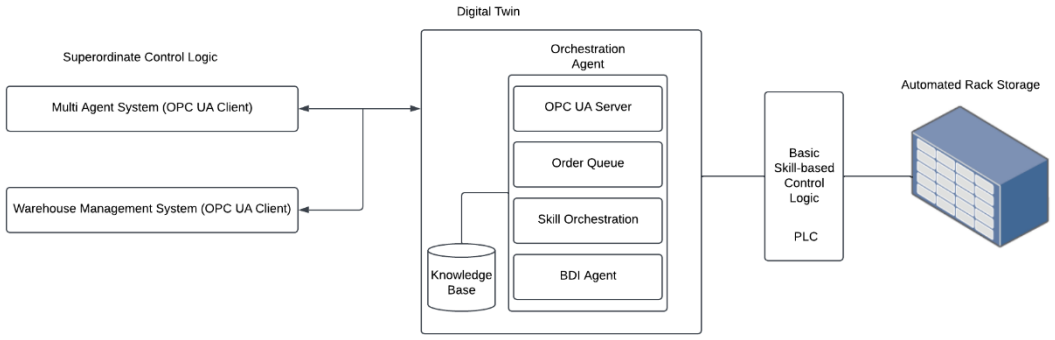


Abbildung 13: Erweiterung des Konzepts nach Lober et al. um einen Digital Twin

4.2 Konzept zur Umsetzung eines OAs im Anwendungsfall

4.2.1 Konzepte zur Registrierung von Skills in Node-RED

Im Rahmen dieser Arbeit wird der OA nicht im vollen Umfang, wie im vorherigen Abschnitt, erstellt. Der Fokus der Arbeit liegt auf der Orchestrierung der Skills und nicht auf der Kommunikation und der Verhandlung in einem MAS. Die Orchestration-Funktionalität beruht auf der Bereitstellung der Informationen über einen Knowledge-Graph. In diesem Anwendungsfall soll der OA auf Grundlage eines Knowledge-Graph, die Skills für definierte Produkte zu Prozessen zusammenstellen. Das System soll über eine graphische Oberfläche den aktuellen Status und die aktuellen Abläufe widerspiegeln. Zudem soll über die selbige Oberfläche das System manipuliert werden. Die grundlegende Funktionalität des OAs ist die Anordnung und die Aktivierung von Skills. In diesem Anwendungsfall handelt es sich um die im SPS-Programm definierten Skills für den fischertechnik Demonstrator. Um diese Funktionalität zu erfüllen, gibt es drei generelle Herangehensweisen: Die einfachste Möglichkeit ist, für jeden Skill einen simplen *flow* zu erstellen. Die *flows* bestehen aus einer *Input node*, einer *function node* und einer *MQTT out node*. Über die *Input node* werden die erforderlichen Daten für die Durchführung des Skills übergeben. Dazu gehört der Aufruf des Skills über den Skill Call und abweichend je nach Skill, Variablen zur Definition der Laufzeit von zeitbasierten Abläufen. Die *function node* hat

die Aufgabe, die eingegebenen Daten in das korrekte Datenformat, in dem Fall JSON, umzuwandeln. Am Ende des *flows* erfolgt die Datenübertragung an die SPS über die *MQTT node*. Hierbei werden für die Kommunikation erforderliche Informationen wie die Topic oder QoS über die *node* automatisch angehängt. Für die Steuerung der Inputs gibt es zwei Möglichkeiten: Die Variablen können direkt in der *Input node* in der Programmieroberfläche von Node-RED geschrieben werden. Bei der zweiten Möglichkeit wird die *Input node* durch mehrere *Dashboard nodes* ersetzt. Diese Änderung hat den Vorteil, dass die Eingaben nicht über das Programmier-Interface von Node-RED erfolgen, sondern über das Node-RED Dashboard. Dadurch können die Variablen über *text-input nodes* eingegeben werden und über den Skill Call der Skill aktiviert werden. Beide Varianten ähneln dem Handbetrieb einer Maschine, da die *flows* keine Verbindungen untereinander haben und die gesamte Steuerung durch die manuelle Auswahl der Skills erfolgt. Zudem wird bei diesem Konzept nicht das Statusmodell der Skills verwendet. Es kann der aktuelle Status eines Skills im Dashboard visualisiert werden, allerdings gibt es keine Funktion, die verhindert, das Skills ungewollt gleichzeitig aktiviert werden. Somit werden alle Skills, die im Dashboard aufgerufen werden, sofort aktiviert und können folglich kritische Zustände des Systems verursachen. Eine derartige Konzeption ist vorteilhaft in der Testphase, um die Ansteuerbarkeit des SPS-Programms zu testen oder um in einzelnen Situationen individuelle Skills separat anzusteuern. Für den dauerhaften Einsatz zur Orchestration der Skills ist dieses Konzept jedoch nicht geeignet, aufgrund folgender Eigenschaften: Manuelle Aktivierung der Skills; manuelle Eingabe der Prozessparameter; keine Abläufe möglich, sondern nur einzelne Skills; fehlende Prozesssicherheit.

Ein fortgeschrittener Ansatz ist die Konzipierung eines modularen Systems. Die Vision ist das Prinzip des SkE im SPS-Programm auf Node-RED zu übertragen. Es wird jedem Skill auf der SPS eine Kopie auf Node-RED zugewiesen. Diese Skill-Bausteine in Node-RED können manuell zu einem Prozessablauf angeordnet werden. Im Vergleich zum vorhergehenden Ansatz wird so ermöglicht, dass Skills zu gesamten Prozessabläufen angeordnet werden. Die Abb. 14 veranschaulicht den konzeptionellen Ablauf eines Skill-Bausteins in Node-RED. Dieser besteht aus drei Hauptkomponenten bzw. Hauptprozessen: Der Skill Call inklusive Prozessparameter an die SPS, das Warten auf die Beendigung des Skills und das Zurücksetzen des Skill Calls und der zugehörigen Parameter. Die Übermittlung des Skill Calls muss so konzipiert werden, dass die richtige Reihenfolge eingehalten wird und die Skills nacheinander aktiviert werden. Durch einen Loop kann anschließend überprüft werden, ob der aktuelle Skill noch aktiv ist oder bereits durchgelaufen ist. Zuletzt müssen alle gesetzten Parameter wieder zurückgesetzt werden.

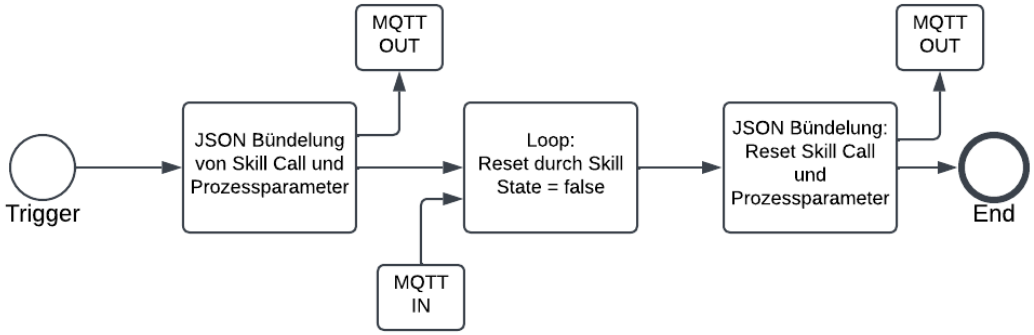


Abbildung 14: Schematischer Ablauf eines Skill-Bausteins

Für die Kombination von *nodes* zu den Skill Bausteinen sind eine Vielzahl an *nodes* notwendig. Die manuelle Anordnung in der Node-RED Programmieroberfläche ist bei wachsender Anzahl an *nodes* unübersichtlich und fehleranfällig. Um die User Experience zu verbessern, können die *nodes* zu *Subflows* zusammengefasst werden. Die *Subflows* werden anschließend als reguläre *nodes* definiert und können beliebig eingesetzt werden. Die Abb. 15 veranschaulicht, wie die Anordnung mithilfe von *Subflows* vereinfacht werden kann.

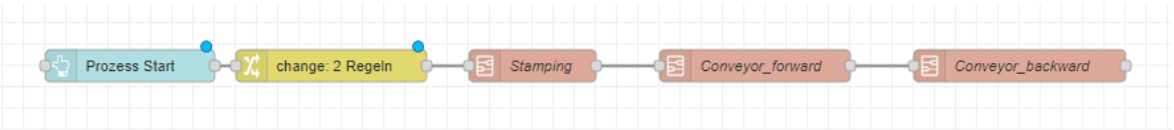


Abbildung 15: Konzept einer starren Schrittkette mit Verwendung von Subflows

Die *Subflows* bzw. Skills sind im Aufbau identisch, unterscheiden sich allerdings in den Schnittstellen zur SPS, da die einzelnen Skills unterschiedliche Inputs und Outputs haben. Für einen geordneten Ablauf beginnt jeder Prozess gleich mit der Übergabe aller relevanten Daten. Sobald alle Daten eingetragen sind, kann der Prozess getriggert werden. Anschließend laufen die zuvor angeordneten Skills sequenziell ab. Um zu erreichen, dass die Skill-Bausteine nacheinander und nicht gleichzeitig ablaufen, werden die Statusangaben der SPS-Skills verwendet. Somit lässt sich in Node-RED bestimmen, ob der gestartete Skill noch aktiv ist oder bereits beendet wurde. Die Herausforderung ist allerdings, dass bei der Anordnung zu einem Prozess, der nachfolgende Skill die exakte Bezeichnung der rücksetzenden Statusvariable des Vorgängers wissen muss. Da die rücksetzenden Statusvariablen für jeden Skill unterschiedlich sind, kann auf diese Weise keine Ablaufsequenz erreicht werden. Um diese Sequenz umsetzen zu können, wird eine Variable benötigt, die wie ein Token funktioniert. Ein Token funktioniert als eine Berechtigung, um senden zu dürfen: Der erste Skill-Baustein erhält den Token durch das Starten des Prozesses. Nachdem der erste Skill ausgeführt wurde, wird der Token an den nächsten Skill-Baustein weitergegeben, was diesem die Berechtigung gibt, den Skill Call

an die SPS zu senden. Dieser Ansatz kann durch das *flow-Prinzip* in Node-RED vereinfacht durch eine spezielle Variable umgesetzt werden. Zu Beginn des *flows* wird die Variable definiert und beim Start des Prozesses auf true gesetzt. Der erste Skill-Baustein besitzt, wie jeder andere Skill-Baustein, eine Bedingung am Anfang. Diese Bedingung lautet, dass die Token-Variable true sein muss, um den eigentlichen Skill-Baustein zu starten. Wenn diese Bedingung erfüllt ist, wird der Skill Call im Baustein durchgeführt und parallel die Token-Variable auf false gesetzt. Anschließend folgt ein Loop, der so lange aktiv ist, bis der Skill in der SPS beendet wurde. Am Ende des Bausteins wird die Token-Variable wieder auf true gestellt und somit dem beliebigen nachfolgenden Skill-Baustein signalisiert, dass dieser beginnen kann. Bei diesem Konzept wird eine universelle Weiterschaltbedingung durch individuelle Statusmeldungen so beeinflusst, dass eine sequenzielle Schrittkette entsteht.

Dieses Konzept ermöglicht ein System, das ohne viel manuellen Aufwand die Skills in der gewünschten Reihenfolge aufruft. Für die Umsetzung werden mit dem Prozess Start die Angaben bezüglich der Stempel-Wiederholungen und die Zeitangaben mitgegeben. Hieraus entwickelt sich bei umfangreicheren Prozessen ein Problem: Es kann zwar jeder Skill in den Prozess integriert werden und das öfter als einmal, allerdings kann einer Kategorie von Skill-Baustein nur eine Zeitvariable zugeordnet werden. Das bedeutet, wenn der Skill-Baustein Stamping z.B. zweimal instanziiert wird, ist die Anzahl der Wiederholungen bei beiden Aktivierungen gleich und kann nicht unterschiedlich sein. Um zu erreichen, dass beide Stamping Skills unterschiedliche Wiederholungszahlen haben, müssten die Skill-Bausteine individuell identifiziert werden. Durch eine solche Maßnahme wird allerdings die Drag and Drop Funktionalität verhindert, die die einfache und schnelle Anordnung ermöglicht. Die angesprochene Einschränkung bezüglich der unflexiblen Einstellung der Prozessparameter für einen Skill-Baustein, der mehrmals instanziiert ist, ist in dieser Form für die industrielle Anwendung ein Ausschlusskriterium.

Die bisher beschriebenen Konzepte ermöglichen die Ansteuerung und Anordnung von Skills der SPS. Beide Konzepte haben allerdings den entscheidenden Nachteil, dass sie manuell gesteuert werden müssen. Vor allem die zweite Variante mit den einzelnen Skill-Modulen ist in der Orchestrierung bereits einfach gestaltet, dennoch erfolgt die Anordnung manuell und im Programmierinterface. Ziel ist es, dass alle Inputs, die das System benötigt, über das Node-RED Dashboard eingegeben werden können. Dies beinhaltet den Prozess-Trigger und alle relevanten Prozessparameter. Dieses Konzept fokussiert die automatische Anordnung bzw. die automatische Ansteuerung der korrekten Skills. Das Konzept der Skill-Module kann als Basis berücksichtigt und erweitert werden. Um einen automatischen Aufruf der richtigen Skills zu erreichen, müssen vier Teilkomponenten implementiert werden. Dazu

gehören die Skill-Bausteine in modifizierter Form, der Orchestrator, das Graphical User Interface (GUI) und eine Knowledge Base mit Informationen über den aktuellen Prozess. Das GUI ist die Schnittstelle zwischen dem System und dem Bediener und ermöglicht die Veränderung und das Auslesen Prozessparameter. Das GUI kann über das Node-RED Dashboard implementiert werden. Dort können Schaltflächen integriert werden, die den Ablauf beeinflussen. Es werden Schaltflächen zur Definition der Reihenfolge der Skills bzw. zur Auswahl des Produktes und zur Aktivierung des Prozesses benötigt. Informationen über den aktuell aktiven Skill und die Zustände aller Aktoren, Sensoren und der Betriebsarten können über das Dashboard entnommen werden.

Ziel des Konzeptes ist es, dass verschiedene Grade an Flexibilität integriert werden: So soll es möglich sein, dass über eine Schaltfläche mit einem Button starr fixierte Abläufe abgerufen werden können. Denkbar sind hier beispielsweise Testabläufe, um die Funktionsfähigkeit aller Aktoren zu überprüfen. In einer anderen Kategorie soll eine Auswahl an verschiedenen Produkten definiert sein. Aus dieser Auswahl kann der Bediener das gewünschte Produkt auswählen und der Prozess wird entsprechend zusammengestellt. An dieser Stelle ist es möglich einen automatischen Ansatz zu verfolgen. Anstatt der gezielten Auswahl durch den Bediener ist es möglich, dass das System über z.B. Farbsensoren das aktuelle Produkt erkennt und dementsprechend die richtigen Prozesse anordnet. Die Umstellung der Aktivierung über die GUI zur automatischen Aktivierung durch Sensoren ist nicht komplex. Die Schaltflächen der GUI müssen durch die individuellen Sensorsignale des Farbsensors ersetzt werden. Der fischertechnik Demonstrator verfügt prinzipiell über einen Farbsensor, dieser ist jedoch an einer anderen Stelle verbaut und ist im betrachteten Teilabschnitt nicht sinnvoll platzierbar.

Die letzte Kategorie zum Abruf von Prozessen soll die individuelle Zusammenstellung von Skills zu einem Prozess sein. Dies ist der höchste Grad an Flexibilität, die das System in diesem Rahmen erreichen kann. Über Schaltflächen soll es möglich sein, Skills an die gewünschte Position im Prozessablauf zu setzen und dabei gleichzeitig die erforderlichen Parameter zu integrieren. Im Rahmen der vorgegebenen Möglichkeiten des Demonstrators kann so jede Ablaufkombination individuell zusammengestellt werden. Durch dieses Prinzip lässt sich beispielsweise die Losgröße 1 umsetzen, sodass kein Teil im Prozessablauf dem Vorgänger und Nachfolger gleicht. Wichtig ist, dass die Schaltflächen entweder starr identifiziert werden, welche Position sie im Ablauf bestimmen können, oder über die Schaltflächen zusätzlich bei der Eingabe des Skills, die gewünschte Position im Ablauf eingegeben wird. Auf die genaue Konzipierung und Design des Dashboards wird im Kapitel 4.6 eingegangen.

4.2.2 Knowledge-Base

Die Knowledge-Base hat entsprechend der beschriebenen Varianten an Flexibilität ebenfalls Teilabschnitte. Für die starren Abläufe sind die Reihenfolge und die Prozessparameter definiert. Bei der Auswahl der Produkte sind diese Variablen zwar vordefiniert aber es müssen alle verfügbar gemacht werden, wovon ein Produkt ausgewählt wird. Es muss ein Prozess der Anordnung stattfinden, entsprechend der Produktauswahl des GUI. Für die individuelle Zusammenstellung der Skills werden weder Reihenfolge noch Parameter im Voraus definiert. In der Knowledge-Base wird hierfür ein Platzhalter definiert. Die über die GUI festgelegte Reihenfolge und Parameter werden über eine Methode in der richtigen Reihenfolge in die Knowledge-Base gepusht. Somit wird für jeden individuellen Prozess vor Beginn die Knowledge-Base mit den gewünschten Skills beschrieben und kann anschließend vom Orchestrator abgerufen werden. Um den nächsten Prozess erneut individuell gestalten zu können, muss dieser Teil der Knowledge-Base entweder nach Durchlauf des Prozesses oder durch einen Reset-Button gelöscht werden. Die Variante des Reset-Buttons hat den Vorteil, dass falls der Prozess ein zweites Mal ablaufen soll, dieser nicht erneut zusammengestellt werden muss.

4.2.3 Konzepte zur Orchestrierung

Der Orchestrator ist zuständig für die Varianten mit den auswählbaren Produkten und den individuellen Prozessabläufen. Beide Teile der Knowledge Base können integriert werden, da in beiden Fällen die Art der Speicherung nach dem gleichen Schema erfolgt. Für die Umsetzung des Orchestrators gibt es verschiedene Möglichkeiten. Eine Variante besteht aus der Implementierung von einer Vielzahl an Platzhalter-Modulen, die für jeden Prozess neu definiert werden können. Diese Platzhalter-Module rufen bei Aktivierung, entsprechend der Eingabe den korrekten Skill-Baustein auf. Ein modellhafter Ablauf davon ist in der Abb. 16 abgebildet. Die Abbildung ist auf die Systemgrenzen von Node-RED begrenzt.

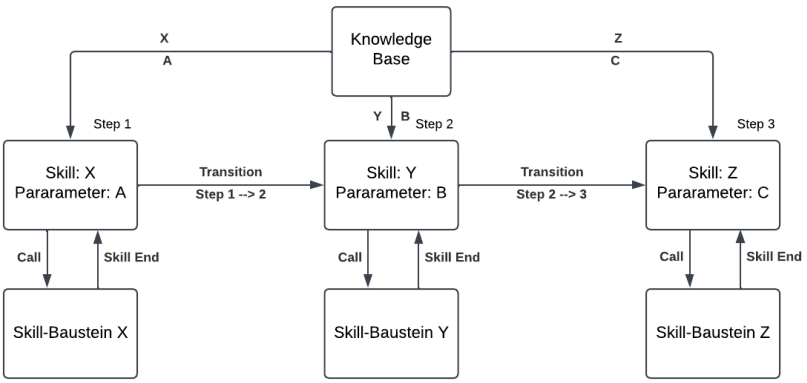


Abbildung 16: Modell einer Orchestrierung über definierte Platzhalter-Module

Der Vorteil dieses Ansatzes ist, dass die Transitionen zwischen den Schritten eindeutig sind, da die Module Step 1, 2 und 3 konstant sind. Somit ist die Transition zwischen Step 1 und Step 2 immer identisch, selbst wenn den Steps unterschiedliche Skills zugewiesen werden. Dieses Konzept hat allerdings den entscheidenden Nachteil, dass die Anzahl der Skills in einem Prozess begrenzt ist. Die Einschränkungen sind die Platzhaltermodule. Abhängig davon, wie viele Platzhaltermodule implementiert werden, kann der Prozess maximal so viele Skills beinhalten. Somit sind die Prozesse durch die initiale Implementierung eingeschränkt. Es besteht die Möglichkeit, eine Vielzahl an Platzhalter vorab einzufügen, allerdings besteht dennoch die Chance, dass es für unvorhergesehene Prozesse zu wenige Platzhalter sind. Zudem ist diese Art der Implementierung nicht effizient und damit nicht für die Umsetzung geeignet.

Um einen OA zu gestalten, der beliebig viele Skills bzw. Steps abarbeiten kann, muss ein anderer Ansatz gewählt werden. Ein etablierter Ansatz, um Elemente nur einmal zu erstellen aber öfter aufzurufen, ist der Einsatz von Schleifen bzw. Loops. Loops ermöglichen die mehrmalige Aktivierung eines Bausteins, bis eine bestimmte Bedingung erfüllt ist. Somit ist es möglich nur ein Platzhalter-Modul zu erstellen, das dann beliebig oft durch den Loop aktiviert wird. Ein entscheidendes Kriterium ist, in welcher Form die Daten über die Skills und deren Reihenfolge an den Orchestrator weitergegeben werden. Eine Möglichkeit dafür sind Arrays. Diese speichern Informationen an einer definierten Stelle im Array. Die erste Information wird auf den Index 0 geschrieben und alle anderen Werte auf den nächsten ganzzahligen Index in aufsteigender Reihenfolge. Diese Art der Speicherung ermöglicht es Elemente, anhand des Index abzurufen ohne weitere Informationen über den Inhalt zu haben. Zudem ist es unkompliziert Arrays zu bearbeiten. Die Methode *„shift()“* ermöglicht es das Element auf dem Index 0 zu löschen, sodass folgend das Element auf Index 1 auf Index 0 vorrückt. Analog dazu verhalten sich die restlichen Elemente. In Kombination mit Loops ist es möglich zu Beginn einen Array zu erstellen, der in der

korrekten Reihenfolge die Skill Calls enthält. Sobald der Array an das Platzhalter-Modul übertragen wurde, wird der Skill Call auf dem Index 0 ausgeführt und an den entsprechenden Skill-Baustein gesendet. Anschließend erfolgt eine Schleife, in der auf die Beendigung des Skills gewartet wird. Diese ist vergleichbar mit der Schleife in den einzelnen Skill-Bausteinen. Sobald der Skill beendet ist, wird der Skill Call auf dem Index 0 gelöscht und der nachfolgende Skill Call rückt auf den Index 0 vor. Nachfolgend wird durch einen übergreifenden Loop der Array wieder an das Platzhalter-Modul als Input weitergegeben. Der Prozess wird wiederholt, mit dem Unterschied, dass der Skill Call sich geändert hat. Der übergreifende Loop ist so lange aktiv, bis der Inhalt des Arrays leer ist. Wenn der Array komplett durchgelaufen ist, sind alle aufgelisteten Skills abgearbeitet worden und der Prozess ist beendet. Anschließend kann ein neuer Prozess gestartet werden. Während der Skill Call für jeden Call individuell sein muss, kann die Statusmeldung für den beendeten Skill universal gleich gestaltet werden. Das bedeutet, dass jeder Skill-Baustein bei Beendigung des Skills, die identische Statusmeldung versendet. Abb. 17 visualisiert das beschriebene Konzept.

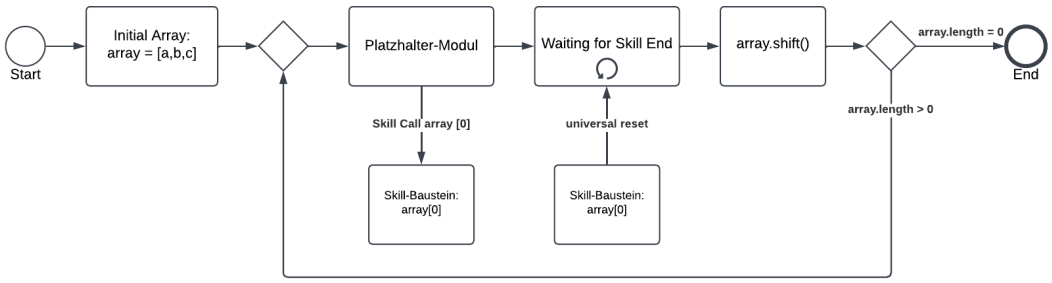


Abbildung 17: Konzept eines OAs mit unbegrenzter Anzahl an aufrufbaren Skills

Die Skill-Bausteine sind bezüglich des Aufbaus fast identisch mit den Bausteinen aus dem vorherigen Konzept. Der wesentliche Unterschied ist, dass die Bausteine nicht in der richtigen Reihenfolge angeordnet sind und somit nicht durch einen universalen Trigger ausgelöst werden können. Stattdessen muss jeder Skill-Baustein durch einen individuellen Skill Call ansteuerbar werden. Im Gegensatz dazu ist die Information über die Beendigung des Skills als universal gleicher Trigger umsetzbar. Die Abb. 18 bietet eine grafische Übersicht über die einzelnen Bestandteile und deren Beziehungen untereinander.

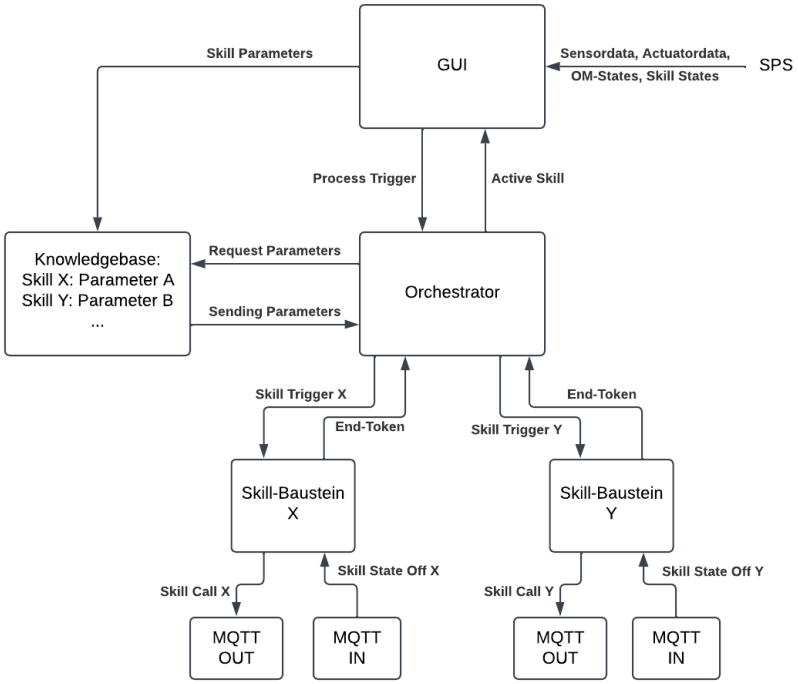


Abbildung 18: Schematische Anordnung und Beziehungen der Bestandteile des OAs

4.3 Konzeption der Skills

Die standardmäßige Umsetzung von Automatisierungsprojekten erfolgt über Schrittketten. Schrittketten bestehen in der einfachsten Form aus einer nicht-verzweigten Kette von Schritten. Diese Schritte repräsentieren Befehle, die vom System ausgeführt werden sollen. Die Schritte sind durch Transitionen voneinander abgetrennt. Bei Transitionen handelt es sich um Bedingungen, deren Erfüllung erforderlich ist, um den nächsten Schritt aktivieren zu können. Durch unterschiedliche Anordnungen dieser Schrittketten mit beispielsweise Verzweigungen oder parallelen Abschnitten können komplexe Prozesse abgebildet werden. Diese Methode der Programmierung hat den Vorteil, dass die Erstellung des Programms einfach und schnell möglich ist. Daraus entstehen allerdings die Probleme, dass das Programm inflexibel, umfangreich und redundant wird. Die Herausforderung sind Änderungen im Programm zu einem späteren Zeitpunkt. Zusätzliche Optionen lassen sich durch eine zusätzliche Verzweigung integrieren, vergrößern allerdings das Programm signifikant. Bei Umstellungen in den bestehenden Abläufen müssen Änderungen am gesamten Programm vorgenommen werden, da jeder Schritt und jede Transition mit dessen Vorgänger und Nachfolger eindeutig verknüpft ist. Häufig sind viele Abschnitte oder Verzweigungen fast identisch, was den Umfang des Programms unnötig vergrößert. Es wird deutlich, dass diese Art der Programmierung für flexible Anlagen nicht optimal geeignet ist. Das Prinzip des SkE bietet einen Ansatz, der zwar den initialen Aufwand der Erstellung des

Programms erhöht, aber die Flexibilität und Modularität signifikant verbessert. Der Vorteil dieses Ansatzes im Vergleich zur klassischen Schrittkette ist, dass die Erstellung nicht bei der Zusammenstellung des gewünschten Prozesses beginnt, sondern bei der Analyse der Ressourcen bezüglich deren Fähigkeiten bzw. Skills. Die Ressource „Conveyor“ besitzt z.B. einen Skill der die Vorwärtsbewegung des Förderbands ermöglicht, dadurch kann ein Transport-Service realisiert werden. Dieser Skill wird anschließend als universal ansteuerbare Schrittkette programmiert. In einem System, dessen Skills alle bekannt sind, können anschließend durch eine übergeordnete Instanz die Skills zu dem gewünschten Prozess angeordnet werden. Dieses Prinzip ermöglicht es, dass Programmbestandteile mehrmals wiederverwendet werden können und das Programm bei Änderungen flexibel angepasst werden kann bzw. modular erweitert werden kann. Die Umsetzung von SkE ist vor allem in Hinsicht auf schrumpfende Losgrößen und einem wachsenden Anteil individueller Produkte relevant. In diesen Bereichen, wo Flexibilität explizit gefordert ist, macht es Sinn diesen Ansatz zu wählen, allerdings ist die Verwendung beispielsweise im Bereich der Serienproduktion von dauerhaft identischen Produkten nicht vorteilhaft, da die Prozesse sich kaum verändern. Für diese Arbeit wird der Ansatz des SkE verfolgt, weil das Zielsystem flexibel bekannte Produkte bzw. individuell erstellte Prozesse umsetzen soll. Die Konzeption der Umsetzung der Skills auf der Ebene der SPS-Steuerung wird nachfolgend beschrieben.

Die physische Ausführung des Programms erfolgt über die PLC, die nach den Ansätzen des SkE programmiert wurde. Die Skills in diesem System sind auf Grund des simplen Aufbaus des Systems im grundlegenden Aufbau ähnlich. Der Großteil der Skills besteht aus der Aktivierung eines Aktors, in der Regel ein Motor, der eine Startbedingung und eine Endbedingung hat. Dabei handelt es sich um Zeitwerte oder um Tastsensoren bzw. Lichtschranken. Um die Komponenten des Demonstrators nach dem SkE anzupassen, müssen zunächst die Fähigkeiten der Komponenten analysiert werden. Es ist festzustellen, dass viele der Fähigkeiten mindestens zu einer Fähigkeit einer anderen Komponente eine große Ähnlichkeit aufweisen. Im Sinne des SkE sollen diese Fähigkeiten in einem Skill gebündelt werden, um die Größe des Programms zu minimieren und die Wiederverwendbarkeit von Bausteinen zu erhöhen. Um das zu erreichen, werden die Fähigkeiten nach deren Aufbau kategorisiert. Die erste Kategorisierung erfolgt in folgende Kriterien:

1. Skills, die aus zwei Tastsensoren und einem Aktor bestehen.
2. Skills, die aus einem Tastsensor, einem Aktor und einer Zeitsteuerung bestehen.
3. Skills, die aus zwei Lichtschranken und einem Aktor bestehen.

4. Skills, die aus zwei Lichtschranken, einem Aktor und einer Zeitsteuerung bestehen.
5. Skills, die aus einer Lichtschranke, einem Aktor und einer Zeitsteuerung bestehen.
6. Skills, die aus einer Zeitsteuerung und einem Aktor bestehen.

Abgesehen vom Sauggreifer kann das ganze System durch Skills beschrieben werden, die sich in die obigen Kategorien einteilen lassen. Innerhalb dieser Kategorien sind bereits identische Skills vorzufinden, da beispielsweise die Stempelmaschine im Demonstrator zweimal mit dem identischen Aufbau existiert. Diese Skills können innerhalb der Kategorien weiter zusammengefasst werden. Beispielsweise können mehrere Förderbänder, die über zwei Lichtschranken verfügen, in einem Skill je Transportrichtung beschrieben werden. Um die Skills so aufzubereiten, dass diese universeller eingesetzt werden können, ist es sinnvoll einzelne Skills zu kombinieren bzw. zu erweitern. Eine sinnvolle Erweiterung um eine Zeitsteuerung wird bei dem Skill für das horizontale Einfahren des Kranauslegers deutlich. Hierbei wird anfangs der Skill nur durch den Anschlagstaster in ganz eingefahrener Position beendet. Hinsichtlich Optimierungsmöglichkeiten bezüglich der Ablaufzeiten ist häufig das Rücksetzen des Skills bei einem Bruchteil des Weges bereits ausreichend. Die Ergänzung der Rücksetzbedingung um eine Zeitsteuerung ermöglicht somit umfangreichere Einsatzmöglichkeiten des Skills. Bei der Berücksichtigung solcher Faktoren können die Skills final in folgende Kategorien eingeteilt werden:

1. One Lightbarrier
2. Two Lightbarriers
3. Time controlled
4. One Pushbutton
5. Two Pushbuttons
6. Other

Die Kategorienamen sind hierbei nicht exklusiv, sondern dienen als identifizierendes Merkmal. Gegenüber der ersten Einteilung bestehen keine Unterschiede in der Anzahl der Kategorien. Theoretisch ist es möglich den Großteil der Skills in einer Kategorie zu deklarieren. Das liegt an dem simplen Aufbau der Ressourcen des Demonstrators. Um einen gewissen Bezug zu realen Anwendungsfällen zu erhalten, werden individuelle Skills für die Ressourcen erstellt, obwohl viele Skills hinsichtlich der Steuerungslogik identisch sind. Die Skills Drilling und Milling sind als Beispiele aufzuführen. Beide Skills bestehen ausschließlich aus einer zeitlich begrenzten Aktivierung eines Motors. Die Skills bilden schematisch, die in der Realität deutlich unterschiedlichen Aktivitäten, Fräsen und Drehen

ab und sind deswegen als individuelle Skills angelegt. Eine Übersicht über die einzelnen Skills und deren Einordnung in Kategorien kann der Tabelle 2 entnommen werden.

Tabelle 2: Übersicht der erstellten Skills

Skill	Category
Conveyor forward FB_Skill_2LBConveyor_forward	Two Lightbarriers
Conveyor backward FB_Skill_2LBConveyor_backward	Two Lightbarriers
Conveyor forward FB_Skill_1LBConveyor_forward	One Lightbarrier
Crane horizontal forward FB_Skill_Crane_hzforward	Time controlled
Crane horizontal backward FB_Skill_Crane_hzbackward	One Pushbutton
Crane vertical upward FB_Skill_Crane_vtup	One Pushbutton
Crane rotation clockwise FB_Skill_Crane_clockwise	One Pushbutton
Crane rotation counterclockwise FB_Skill_Crane_counterclockwise	One Pushbutton
Crane vertical downward FB_Skill_Crane_vtdown	Time controlled
Pusher FB_Skill_Pusher	Pushbuttons / Other
Milling FB_Skill_Milling	Time controlled
Drilling FB_Skill_Drilling	Time controlled
Crane gripper suction FB_Skill_Crane_Suction	Other
Stamping FB_Skill_Stamping	Pushbuttons / Other

Die grün markierten Felder beziehen sich auf Skills, die für die Rotation der Kräne verantwortlich sind. Der Demonstrator verfügt über zwei Kräne, wobei der Anschlagstaster für die Rotationsbewegung nicht an der gleichen Seite platziert ist. Somit ist die Bewegung im Uhrzeigersinn im Sinne von Kran A rein zeitgesteuert, während Kran B in dieser Drehrichtung über einen Anschlag verfügt. Um die Anzahl der Skills zu reduzieren und diese fähiger zu gestalten wird für jede Drehrichtung nur ein Skill erstellt. Bei deren Einsatz ist je nach Anwendungsfall nur die zuerst eintretende Rücksetzbedingung relevant. Die gebildeten Skills sind alle als composite Skills angelegt, das bedeutet, dass diese aus zwei oder mehr basic Skills zusammengesetzt sind. Die gelb markierten Skills sind composite Skills zweiter Ordnung bzw. zusammengesetzt aus zwei individuellen composite Skills. Im Kontext des Pusher Skills besteht der Ablauf aus einer Vorwärts- und Rückwärts-Bewegung. Wie die Bewegung eines Förderbands mit zwei Lichtschranken in eine Richtung, ist die Vorwärtsbewegung des Pushers mit zwei Tastsensoren theoretisch ein eigenständiger Skill. Im Gegensatz zum Förderband, das nicht immer den gleichen Ablauf hat, laufen die beiden basic Skills des Pushers immer nacheinander in der gleichen

Reihenfolge ab. Aus diesem Grund werden die Skills zu einem composite Skill zweiter Ordnung verbunden. Ein vergleichbares Prinzip trifft auf die Stampingmachine zu. Hauptgrund für die Erstellung eines composite Skills zweiter Ordnung ist allerdings, dass dadurch eine variable Wiederholungsschleife im SPS-Programm direkt implementiert werden kann. Grundsätzlich ist das Prinzip einer Wiederholungsschleife nicht komplex. Der Ablauf ist schematisch in einem BPMN-Diagramm in Abb. 19 dargestellt. Die Abänderung des Skills im Vergleich zur Implementierung im Vorgängerprogramm ist, an welcher Stelle die Anweisung für die Wiederholung erfolgt. Im Vorgängerprogramm wurde die Wiederholung in die Service-Schrittkette implementiert. Da diese Implementierung nicht geeignet ist für die Integration des OAs muss der Skill so konzipiert werden, dass dem Skill Call eine Variable übergeben wird, die die Information enthält, wie oft der Skill sich eigenständig wiederholen soll.

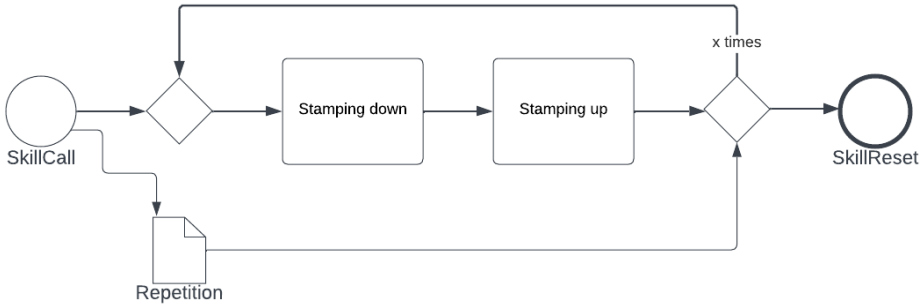


Abbildung 19: Schematischer Ablauf des Stamping Skills

Um den Skill der Kategorie „Two Lightbarriers“ so zu modifizieren, dass er für alle Förderbänder mit zwei Lichtschranken verwendbar ist, muss der Skill folgende Eigenschaften aufweisen: Erstens muss der Skill durch Unterbrechung der Lichtschranke rücksetzbar sein. Zweitens muss der Skill durch Ablauf eines Timers nach Unterbrechung der Lichtschranke rücksetzbar sein. Die zweite Bedingung soll nur gelten, wenn eine Zeitvariable mit dem Wert ungleich null an den Skill übergeben wird. Die gleiche Zeitvariable ist verantwortlich für die Dauer des Timers. Die erste Bedingung soll nur zurücksetzen, wenn der Wert der Zeitvariable gleich null ist. Die Abb. 20. veranschaulicht das Konzept als BPMN-Diagramm.

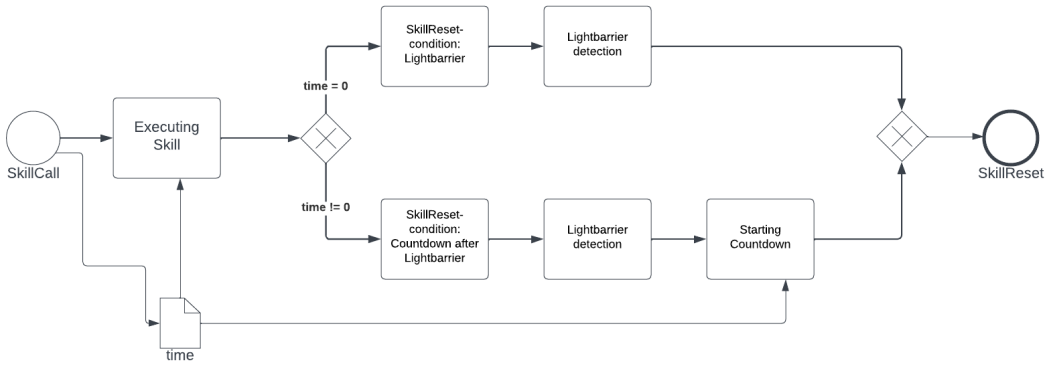


Abbildung 20: Schematischer Ablauf eines Skills der Kategorie Two Lightbarriers

Ein ähnliches Prinzip ist für den Skill der Kategorie „One Lightbarrier“ gültig. Der Skill soll entweder durch eine Lichtschranke oder durch einen Timer zurückgesetzt werden. Wie bei dem vorhergehenden Skill, ist die Entscheidungsvariable der übergebene Zeitwert. Das Konzept über den Ablauf von diesem Skill ist in der Abb. 21 veranschaulicht.

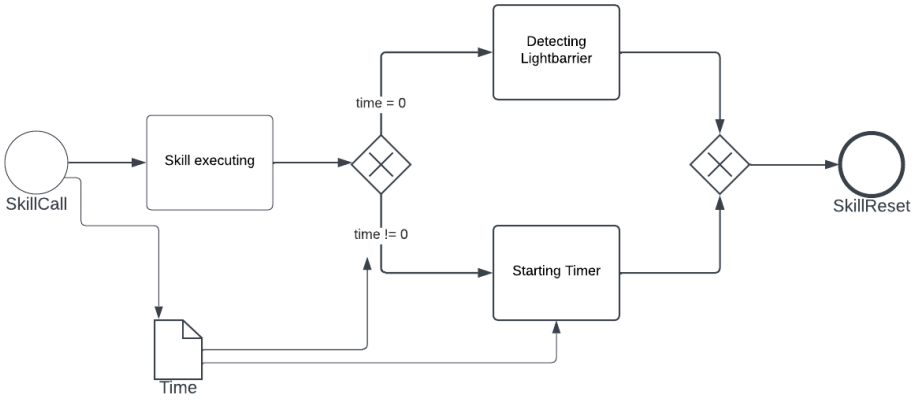


Abbildung 21: Schematischer Ablauf eines Skills der Kategorie One Lightbarrier

Das gleiche Grundprinzip gilt für Skills der Kategorie „One Pushbutton“. Der Skill soll entweder an der Endposition durch den Tastsensor oder an einer variablen Position durch die Zeitsteuerung beendet werden.

Der Skill für die Steuerung des Sauggreifers unterscheidet sich von den restlichen Skills. Die Besonderheit ist, dass der Skill über die Dauer anderer Skills hinweg aktiv sein muss. Hintergrund dafür ist, dass das Vakuum zum Greifen über den ganzen Transportvorgang des Krans erhalten werden muss. Somit hat der Skill nicht die Fähigkeit, nach Durchlauf des internen Zyklus, sich selbst durch Sensoren oder Zeitvariablen zurückzusetzen. Wie beim Skill Call, muss deswegen die Aufforderung zum Zurücksetzen, als externe Anweisung erfolgen. Das bedeutet, dass der OA den Skill zweimal ansteuern muss, anstatt nur einmal wie bei den restlichen Skills. Die Abb. 22 veranschaulicht den Ablauf des Skills.

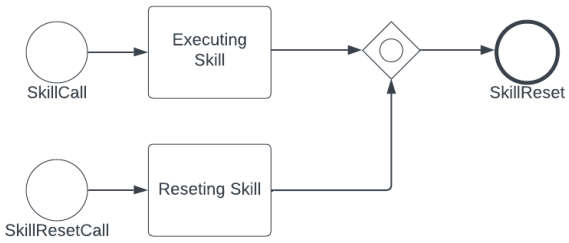


Abbildung 22: Schematischer Ablauf des Skills für den Sauggreifer

Das vorgestellte Konzept zur Verwendung des SkE setzt sich trotz des gleichen Ansatzes deutlich von Vorgängerversionen des Programms ab. Ein großer Unterschied liegt hier in der allgemeinen Ausrichtung und dem Grad der Detaillierung. Die bisherigen Versionen basieren auf der Verwendung von Skills, diese werden allerdings zu Services angeordnet. Die Services sind allerdings als Schrittketten definiert. Das bedeutet, dass die Orchestrierung nicht auf der Ebene der Skills stattfindet, sondern auf der Ebene der Services. Die Abb.23 visualisiert den Unterschied der Konzepte.

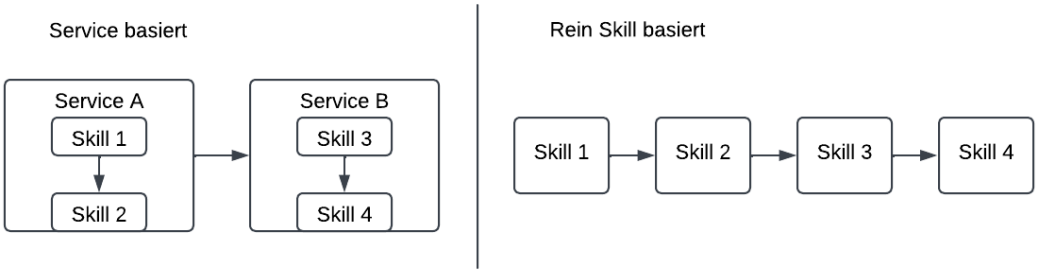


Abbildung 23: Visualisierung des Unterschieds des service-basierten und skill-basierten Ansatzes

Aus der Abb. 23 wird deutlich, was der Nachteil der Service-basierten Variante ist: Durch die Bündelung der Skills zu Services geht ein gewisser Grad an Flexibilität verloren. Das Programm ist in der Lage verschiedene Abläufe variabel anzuordnen, die Abläufe selbst sind jedoch fest definiert. Der Ansatz dieser Arbeit mit Verwendung des rein Skill-basierten Konzepts hat den Vorteil, dass alle definierten Skills flexibel angeordnet und separat aufgerufen werden können. Im Rahmen der Arbeit werden composite Skills verwendet, die sich von basic Skills hinsichtlich erweiterter Funktionalitäten unterscheiden. Die Verwendung von composite Skills ist aus folgenden Gründen besser geeignet: Zum einen ermöglicht die Verwendung von composite Skills die Implementierung von z.B. Skill-internen Wiederholungsschleifen, die durch einen einzigen Skill Call getriggert werden können. Des Weiteren erreicht das System durch die Verwendung von basic Skills in der Theorie einen höheren Grad an Flexibilität, durch die Untergliederung in noch kleinere Einheiten, aber in der Praxis ist ungewiss, ob dieser Grad an Flexibilität überhaupt umsetzbar ist. Der Aufbau des Systems bzw. die vorhandenen Aktoren und Sensoren

beeinflussen maßgeblich die Funktionalität des Systems. Aufgrund des simplen Aufbaus hat die Verwendung von basic skills keinen Vorteil gegenüber composite Skills, dabei haben composite Skills den Vorteil, dass diese fähiger sind und somit häufiger wiederverwendet werden können.

Neben einem einheitlichen Konzept für den Aufruf einzelner Skills muss ein Modell entwickelt werden, das den OA informiert, welche Skills aktiviert bzw. deaktiviert sind. Diese Information ist wichtig für die Reihenfolgesteuerung der Skills. Ohne die Information, ob der aktuelle Skill schon beendet wurde, kann der OA nicht den nächsten Skill aufrufen. Für den Anwendungsfall werden zwei Zustände definiert: Zustand „on“ und Zustand „off“. Die Implementierung der Weitergabe dieser Variablen erfolgt in den Netzwerken der Skills. Der Zustand „on“ wird gleichzeitig mit der Variable zur Aktivierung der Ausgänge des Skills gleich 1 gesetzt. Die Aktivierung des Zustands „off“ erfolgt parallel bei der Erfüllung der Rücksetzvariable. Das Konzept der Weitergabe der Skill-Zustände wird in der Abb.24 grafisch veranschaulicht.

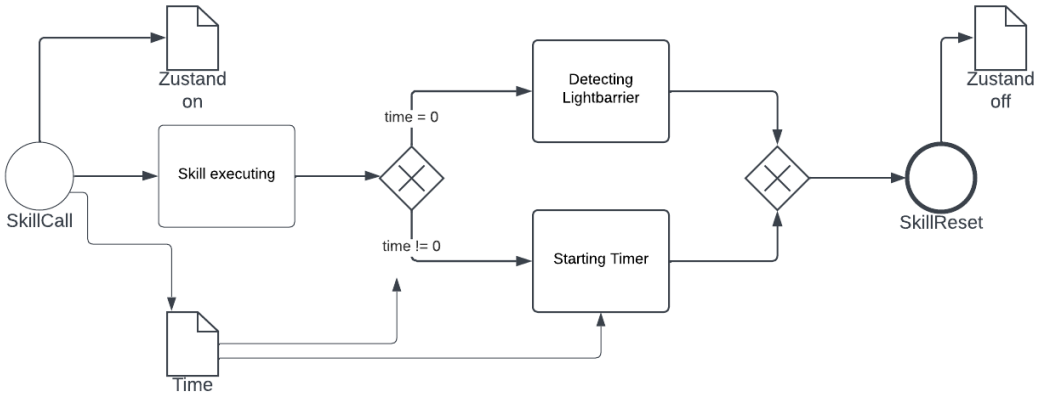


Abbildung 24: Schematischer Ablauf des Zustand-Modells

Diese Erweiterung der Skills hat den Vorteil, dass mehr Informationen über die Prozesse verfügbar gemacht werden. Abhängig von den verfügbaren Informationen über die Prozesse kann zum einen die Orchestrierung der Skills verbessert werden und der Ablauf kann in einem digitalen Abbild detailgetreu abgebildet werden. Hinsichtlich der Integration von Digital Twins in vergleichbare Systeme spielt die vollständige Abbildung aller Prozesse und Komponenten eine entscheidende Rolle.

4.4 Konzeption der Betriebsarten

Für prozesssichere Abläufe und verschiedene Verwendungsarten von SPS gesteuerten Anlagen werden sogenannte Betriebsarten integriert. Betriebsarten sind vergleichbar mit Zuständen oder Modi, in denen sich ein System befindet. Der normale automatische Produktionszustand wird z.B. als Automatikbetrieb bezeichnet. Betriebszustände haben unter anderem die Funktion, das System bei Störzuständen sicher zu regulieren. Dabei kann es sich um den sofortigen Wechsel in den Notaus oder um das Herunterfahren nach dem aktuellen Zyklus, aufgrund nicht-schwerwiegender Störungen, handeln. Somit werden für die Programmierung des Demonstrators Betriebsarten verwendet. Für die Erstellung der Betriebsarten wird die GEMMA Vorlage verwendet bzw. abgeändert. Für das System werden vorrangig folgende Betriebsarten notwendig sein: Im Bereich der Kategorie Stillstände (A) werden die Betriebsarten Initialzustand (A1), Fahrt in den Initialzustand (A4) und Stillstand am Ende des aktuellen Zyklus angefordert (A2) benötigt. Der Initialzustand beschreibt den Zustand des Systems, wobei alle Aktoren sich in ihrer Grundstellung befinden. Folgend ist A4 die Betriebsart, in der die Aktoren in diesen Zustand fahren. Die Betriebsart A2 ist erforderlich, um einen Übergang aus dem Automatikbetrieb in den Initialzustand zu ermöglichen. Dies soll allerdings nicht sofort, sondern erst, wenn die Aufgaben des aktuellen Zyklus abgearbeitet sind, erfolgen. In der Kategorie der Betriebszustände (F) wird für diesen Anwendungsfall nur der Automatikbetrieb (F1) verwendet. Der Notaus (D1) ist die einzige benötigte Art der Störzustände (D). Der Zustand ohne Stromversorgung wird als PZ definiert. Eine Übersicht über die Betriebsarten und deren Anordnungen bietet die Abb. 25.

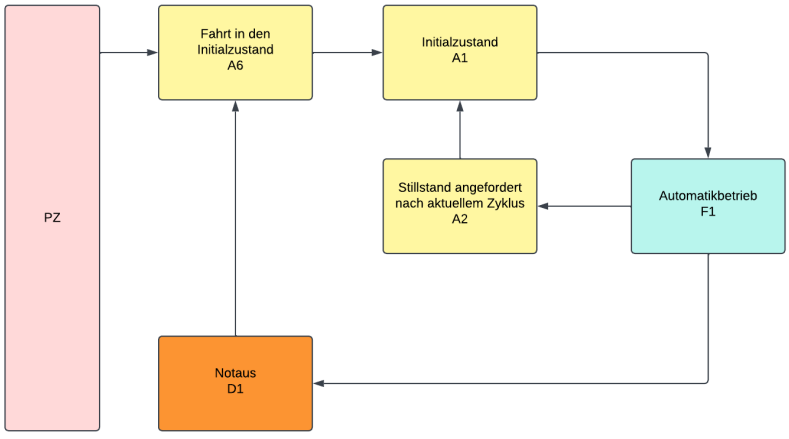


Abbildung 25: Übersicht der erforderlichen Betriebsarten

Die Pfeile zwischen den Betriebsarten bilden die Transitionen/Übergänge ab. Diese können aus physischen Inputs, wie die Betätigung eines Schalters für die Aktivierung des

Automatikbetriebs, bestehen oder automatisch ablaufen. Beispielhaft für einen automatischen Wechsel ist der Übergang von A6 zu A1.

Für die Implementierung der Betriebsarten gibt es drei Möglichleiten: Der klassische Ansatz ist die Integration der Betriebsarten direkt in das SPS-Programm. Dieses Vorgehen ist insofern geeignet, da die Betriebsarten zu den Grundlagen des Programms gehören und diese nur selten Gegenstand von Änderungen sind. Zudem kann dadurch eine angemessene Reaktionsgeschwindigkeit des Systems erreicht werden, die z.B. für den Notaus erforderlich ist. Aufgrund des SkE Ansatzes muss es zur Initialisierung der Aktoren eine Schnittstelle zwischen den Betriebsarten und den Skills geben. Nach der aktuellen Konzeption des Systems sollen die Skills durch den OA gesteuert werden. Werden die Betriebsarten im SPS-Programm umgesetzt, müssten die Anweisungen der Betriebsarten zuerst an den OA weitergegeben werden, der diese dann an den Skills ausführt.

Der zweite Ansatz ist, die Betriebsarten in Node-RED zu erstellen. Das hat den Vorteil, dass die Skills weiterhin, über nur einen Skill Call angesteuert werden können bzw. ein Kommunikationsschritt ausgelassen werden kann. Der Nachteil an diesem Ansatz ist, dass die Zykluszeiten für den Notaus zu lang sein könnten, um sichere Prozesse zu garantieren. Die dritte grundlegende Variante ist eine Schnittmenge aus beiden Bereichen. Eine Möglichkeit hierfür ist, dass die Betriebsarten grundsätzlich im SPS-Programm definiert werden. Abweichend zur klassischen Variante kann aber der Vorgang der Initialisierung bzw. der Fahrt in die Grundstellung über einen vordefinierten Ablauf in Node-RED gesteuert werden. Für diese Abläufe ist zwar keine richtige Orchestrierung notwendig, da die Abläufe immer gleich sind, aber durch diese Variante kann erreicht werden, dass die Skills weiterhin über einen Skill Call angesteuert werden können. Dieses Konzept hat allerdings den Nachteil, dass nur ein Teil der Betriebsarten ausgelagert wird und somit mehr Informationsaustausch zwischen Node-RED und der SPS notwendig ist, und der Demonstrator kann ohne Verbindung zu Node-RED nicht prozesssicher in die Grundstellung gefahren werden.

Um diesen Aspekt zu berücksichtigen ist folgendes Konzept eine Lösung: Neben der Kategorie der Skills für die Prozessabläufe werden im SPS-Programm Funktionsbausteine implementiert, die nur für die Initialisierung des Systems verantwortlich sind. Das SPS-Programm wird untergliedert in den Hauptteil, der aus den Skills besteht, die zu den tatsächlichen Prozessen orchestriert werden, und einem Teil, der prozesssicher die grundlegende Funktionalität des Systems garantiert. Durch die komplette Implementierung der Betriebsarten im SPS-Programm in dieser Ausführung kann die Reaktionszeit für den Notaus gewährleistet werden, die Skills weiterhin nur über einen Eingang getriggert werden und die Betriebsarten bei Bedarf nicht nur über Node-RED prozesssicher gesteuert werden.

Auf dieser Basis lässt sich nicht final entscheiden welches Konzept am besten für die Implementierung geeignet ist. Sowohl die SPS-seitigen als auch Node-RED-seitigen Ansätze haben Vorteile. Absehbar ist allerdings, dass eine Kombination aus beiden Systemen den größten Nutzen liefern kann. Wichtige Entscheidungskriterien sind dennoch die Reaktionsgeschwindigkeit bei kritischen Situationen, die simple Skalierbarkeit der Betriebsarten parallel zum System, im Sinne des SkE und die Möglichkeit für menschliches Eingreifen in kritischen Situationen.

4.5 Kommunikation

4.5.1 Anforderungen

Aufgrund der Implementierung des OAs wird neben der Zuweisung der Variablen innerhalb des SPS-Programms ein definierter Kommunikationsstandard mit definierten Variablen zwischen Node-RED und der SPS benötigt. Der Fokus hinsichtlich der Kommunikation liegt auf dem kontrollierten Aufrufen von Skills durch den OA. Wie bereits im Kapitel 4.3 beschrieben, ist jeder Skill so definiert, dass dieser durch die Aktivierung eines definierten Inputs ausgeführt werden kann. Die weitere Ansteuerung der Ausgänge durch den Skill erfolgt anschließend auf der Ebene der SPS und erfordert somit keine weiteren externen Schnittstellen. Diesbezüglich liegen im Konzept dieser Arbeit deutliche Unterschiede vor, im Vergleich zu vorgehenden Arbeiten im Rahmen des SkE. Diese wurden durch eine SPS interne Schrittkette aufgerufen und unterscheiden sich somit in zwei wesentlichen Aspekten: Der Aufruf der Skills erfolgt nur durch einen Input, während andere Varianten durch die jeweiligen Ablaufschritte aufgerufen wurden. Somit musste jeder Schritt als Input bei den gewünschten Skills angelegt werden. Die Reduzierung der Inputs für den Skill Call auf eine Variable ermöglicht den universalen Einsatz. Des Weiteren muss der Skill Call für das aktuelle Konzept extern ansteuerbar sein und hierfür als globale Variable deklariert werden. Durch die Änderung des Skill-Modells werden, neben den Skill Calls für den Aufruf der Skills, teilweise weitere globale Inputs benötigt. Diese Inputs beziehen sich auf alle Skills, die abhängig von einer Zeitvariable oder einer Variable für Wiederholungen sind. Die Eliminierung der Schrittkette erfordert eine Schnittstelle zwischen OA und den Skills, die die Übertragung von Prozessparameter ermöglicht, sodass diese Regelung Skill-intern erfolgen kann. Parallel dazu sind diese Inputs teilweise Entscheidungsvariablen dafür, welche Variante eines Skills ausgeführt wird (z.B. Rücksetzung durch Taster oder durch Zeitvariable). Bei dieser Funktion wird die Inputvariable im Skill-Funktionsbaustein auf ihren Wert geprüft. Liegt ein Wert ungleich 0 vor, soll die Rücksetzbedingung zeitbasiert sein. In Fällen, wo der Input gleich null ist, erfolgt die sensorbasierte Rücksetzung. Für diesen Input gelten die gleichen Bedingungen wie für den Skill Call. Durch die Integration des OAs

müssen zudem neue globale Outputs im SPS-Programm hinzugefügt werden. Für die Orchestration ist es notwendig, dass die Skills den aktuellen Zustand (on/off) an den OA weitergeben. Ohne diese Kommunikation kann der OA nicht prozesssicher erkennen, wann der aufgerufene Skill beendet ist. Allgemein werden alle Aktoren- und Sensordaten des Demonstrators über die Systemgrenze der SPS hinaus veröffentlicht, um diesen repräsentativ in einem Dashboard zu visualisieren. Somit können alle relevanten Daten, wie die aktiven Skills, Sensoren, Aktoren, das zu produzierende Produkt und die aktuelle Betriebsart, angezeigt werden. Die Abb. 26 visualisiert schematisch die einzelnen Daten, die ausgetauscht werden.

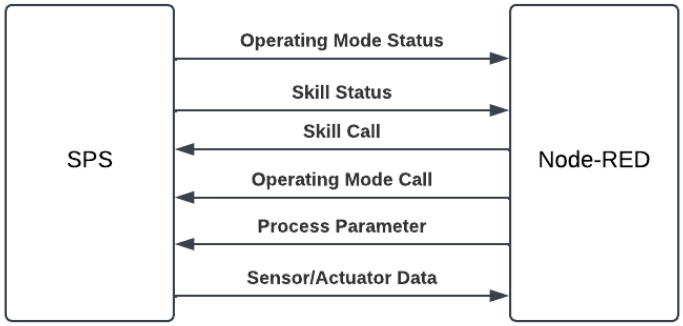


Abbildung 26: Übersicht über die Informationen, die zwischen den Teilnehmern ausgetauscht werden sollen

4.5.2 MQTT

Der Datenaustausch zwischen der SPS und dem OA erfordert ein einheitliches Kommunikationsprotokoll. Hierfür stehen mehrere Varianten zur Verfügung: Die erste Option ist die Verwendung des MQTT-Protokolls. MQTT basiert auf dem Publish-Subscribe Modell. Das Modell hat den Vorteil, dass das System modular erweitert werden kann, durch die Ergänzung von weiteren Clients. Hinsichtlich der Gestaltung eines flexiblen und modularen Systems ist dieser Aspekt vorteilhaft. MQTT ist zudem simpel in der Implementierung und geeignet, um unkompliziert den Datenaustausch zu ermöglichen. Ein weiterer Vorteil ist die standardmäßige Integration von *MQTT Input* und *Output nodes* in Node-RED. Für die Verwendung von MQTT ist die Erstellung eines MQTT-Brokers erforderlich. In dieser Instanz werden die entsprechenden Informationen veröffentlicht (publish) und abgerufen (subscribe). Als Identifikation bzw. Kategorisierung muss eine Topic definiert werden, die als eine Art Überschrift für die restlichen Daten fungiert. Die SPS und Node-RED werden in dem System als Clients angelegt und können publishen und subscriben. Das Publish/Subscribe Modell von MQTT ermöglicht, dass dauerhaft beispielsweise Sensordaten veröffentlicht werden aber nur bei Bedarf abgerufen werden. Für den Datenaustausch wird das Datenformat JSON verwendet. JSON ermöglicht die

kompakte Bündelung von Daten in einer Einheit. Somit können beispielsweise alle relevanten Daten für die Aktivierung eines Skill Calls in einem Objekt gebündelt werden. In einem JSON Objekt können unterschiedliche Datentypen verwendet werden z.B. der Skill Call als boolean und die Zeitvariable als integer. Die Implementierung des MQTT-Protokolls erfolgt nach dem in Kapitel 2.5 beschriebenen ISO/OSI Modell. Hierbei ist die SPS auf der Bitübertragungsschicht über ein Ethernet Kabel mit dem Internet verbunden. Die Verbindung mit dem Internet ist essenziell für die Weitergabe der Daten an den MQTT-Broker. Des Weiteren werden die Protokolle IP (Sicherungs- und Vermittlungsschicht) und TCP (Transportschicht) verwendet. Auf der Anwendungsschicht wird das MQTT-Protokoll mit JSON als Datenformat verwendet.

Um MQTT für den Anwendungsfall konkret umsetzen zu können wird eine einheitliche Topic definiert, die von beiden Clients verwendet wird, um den korrekten Datenaustausch zu garantieren. Anschließend können für beide Clients die Input- und Output-Schnittstellen erstellt werden. Durch die in Node-RED standardmäßig integrierten *„MQTT in“* und *„MQTT out“ nodes* ist der Verbindungsaufbau zwischen dem Node-RED Client und dem Broker unkompliziert. Die zu verschickenden Daten aus den *flows* müssen allerdings spätestens in der letzten *node* in das JSON-Format gewandelt werden. Die *„MQTT out“ node* ermöglicht anschließend die Zuweisung der Daten zu der festgelegten Topic.

Parallel zur Zuweisung der Topic wird die QoS festgelegt. Dafür sind drei Optionen verfügbar: QoS0 ist für den Anwendungsfall nicht relevant, da diese Variante nicht die erfolgreiche Zustellung garantiert. Die nicht-Zustellung eines Skill Calls könnte somit das System in einen undefinierten Zustand bringen, da der Orchestrator den Skill aufgerufen hat und auf dessen Ende wartet, während der Skill von der SPS aufgrund einer verlorenen Nachricht noch nicht aktiviert wurde. Für den Anwendungsfall sind folgend nur Optionen mit garantierter Zustellung relevant. Dieses Kriterium erfüllen sowohl QoS1 und QoS2. Der Unterschied liegt hier in der Art der Sicherstellung, dass die Nachricht empfangen wurde. Bei QoS1 kann es aufgrund des Ablaufs dazu kommen, dass dieselbe Nachricht zweimal empfangen wird. Aufgrund eines anderen Ablaufs garantiert QoS2, dass eine Nachricht genau einmal empfangen wird. Im Vergleich zu den anderen Möglichkeiten ist QoS2 allerdings langsamer und datenintensiver. Entscheidend für die Auswahl ist der Aspekt, ob das System anfällig dafür ist in einen ungewollten Zustand zu kommen, falls eine Nachricht zweimal empfangen wird. Bei den Prozessabläufen sind vor allem zwei Variablen, die über MQTT verschickt werden, wichtig: Der Skill Call zur Aktivierung und der Skill Status als Weiterschaltbedingung für den OA. Eine doppelte Aktivierung des Skill Calls ist in einem gewissen Rahmen nicht als kritisch anzusehen, da die Skills über einen Rücksetz-dominanten Flipflop gesteuert werden und dieser über die Dauer des Skills aktiviert ist.

Diese Aussage ist allerdings nur wahr, wenn die zweite Nachricht empfangen wird, während der Skill noch aktiv ist. Wird die Nachricht empfangen, nachdem der Skill schon beendet wurde, erfolgt eine ungeplante Aktivierung des Skills und das System befindet sich in einem kritischen Zustand. Aufgrund des physischen Aufbaus des Demonstrators dauern alle Skills zum aktuellen Stand mindestens circa ein bis zwei Sekunden an. Es wird angenommen, dass der Zeitraum der verzögerten zweiten Nachricht kürzer ist als die Dauer des Skills mit der geringsten Dauer. Somit ist die Verwendung von QoS1 hinsichtlich der Aspekte der Übertragungsgeschwindigkeit und der Menge der versendeten Daten besser für den Anwendungsfall geeignet als QoS2. Für Node-RED Inputs über MQTT ermöglicht die *„MQTT in“ node* zusätzlich zur Wahl des QoS, die Auswahl zwischen zwei Empfangsmethoden: Empfangen von Nachrichten einer definierten Topic oder das Empfangen von Nachrichten aller Topics eines Brokers. Für den Anwendungsfall wird die Variante mit einer definierten Topic verwendet. Zusätzlich kann über die *node*, das Datenformat des *node-Outputs* definiert werden.

Die Erweiterung der SPS-Clients mit einer MQTT-Funktionalität erweist sich als umfangreicher. Das e!Cockpit von Wago, auf Basis von Codesys, verfügt nicht über standardmäßig integrierte Bausteine für das Publishen und Subscriben. Um diese Funktionalität zu erwirken, muss eine MQTT-Bibliothek dem Programm hinzugefügt werden. Anschließend wird für jede Ressource des Demonstrators ein Programm für Subscribe und ein Programm für Publish instanziiert. Diese sind untergliedert in die jeweiligen Aktoren und Sensoren der Ressourcen. Dieses Schema wird analog für alle anderen Bereich des Programms, wie z.B. die Betriebsarten, umgesetzt. Zusätzlich muss in das Hauptprogramm ein initiales Setup der Peripherie bezüglich MQTT erfolgen. Die Subscribe/Publish Programme der Ressourcen haben die Aufgabe die Inputs und Outputs des SPS-Programms den Variablen zur MQTT-Übertragung bzw. einer Topic zuzuordnen. Des Weiteren müssen diese Informationen vor dem Publishen in das JSON-Format umgewandelt werden bzw. beim Empfangen in den entsprechenden Datentyp umgewandelt werden.

4.5.3 OPC UA

Entgegen dem Publish/Subscribe Ansatz über MQTT besteht die Möglichkeit die Kommunikation zwischen SPS und Node-RED über OPC UA einzurichten. Bei diesem Ansatz ist kein Broker notwendig, da die Kommunikation nach dem Server/Client Modell erfolgt. Hierbei ist der OPC UA Server in die Wago PLC integriert und Node-RED agiert als Client. Für die Informationsübertragung ist es erforderlich, dass zuvor die Variablen auf der Server-Seite deklariert werden. Die Variablen, die kommuniziert werden sollen, werden

ausgewählt und hinsichtlich deren Zugänglichkeit beschrieben. Unterschieden wird in die Möglichkeit, Variablen auszulesen, zu schreiben oder beides. Der gesamte Prozess wird als Mapping bezeichnet und läuft folgendermaßen ab: Zunächst wird eine Symbolkonfiguration dem SPS-Programm hinzugefügt. Die Symbolkonfiguration listet alle Variablen auf, die über OPC UA manipuliert werden können. Dazu gehören vor allem die globalen Variablen und Variablen, die in einem Programmbaustein definiert werden. Standardmäßig können alle Variablen, wie die Skill Calls oder Prozessparameter als „ReadWrite“ deklariert werden. Die Statusvariablen der Skills, Sensoren und Aktoren können als „Read“ deklariert werden, da der Node-RED Client keine Änderung an diesen Variablen vornehmen soll. Der Mapping-Prozess erfolgt im Wago OPC UA Mapping Editor. Die Symbolkonfiguration wird als XML-Dokument gespeichert und in den Mapping Editor geladen. Im Mapping Editor werden die Variablen aus der Symbolkonfiguration über das Mapping dem Informationsmodell hinzugefügt. Bei dem Mapping Prozess werden den Variablen alle erforderlichen Informationen für die Kommunikation zugewiesen. Darunter fällt die NodeId, die die Variable eindeutig identifiziert und ansteuerbar macht. Das Informationsmodell wird als XML-Dokument über das Web-based Management auf die PLC geladen. Diese Schritte ermöglichen, dass der Server die eingerichteten Variablen mit den entsprechenden Eigenschaften zur Kommunikation mit den Clients anbietet.

Auf der Client-Seite müssen externe *OPC UA nodes* zu Node-RED hinzugefügt werden. Im Gegensatz zu MQTT werden die *nodes* nicht unterschieden in *Publish und Subscribe nodes*, sondern *OPC UA Client nodes* verwendet. Die *Client nodes* werden mit dem Server verbunden und können für verschiedene Aufgaben wie Lesen oder Schreiben von Variablen konfiguriert werden. Im Gegensatz zu MQTT werden dafür keine Topics verwendet, sondern NodeIds. Diese setzen sich aus den zwei Bestandteilen Namespace und Identifyer zusammen. Der Identifyer kann in Form eines Strings (s) oder numerisch (i) definiert werden. Ein Beispiel für eine NodeId ist: *„ns=1;s=myswitch“* oder *„ns=1;i=1221“.* Um die Anzahl der Clients zu minimieren, können mehrere *flows* zum Schreiben oder Lesen mehrerer Variablen an einer *OPC UA Client node* zusammengeführt werden. Für die Auswahl des Kommunikationsprotokolls müssen beide Protokolle zusätzlich bezüglich der Durchlaufzeit verglichen werden. Der Vergleich der Durchlaufzeit kann durch die Durchführung des selben Prozesses über beide Protokolle gemessen werden.

4.6 GUI-Konzept

Ein Graphical User Interface ist eine Schnittstelle zwischen dem menschlichen Bediener und dem System. In diesem Anwendungsfall verfügt Node-RED bereits über die Funktionalität, ein Dashboard zu integrieren. Über spezielle *nodes* ist es möglich

Informationen aus den einzelnen *flows* in dem Dashboard anzeigen zu lassen. Somit können alle Sensor Daten und Aktor Daten aus dem Demonstrator über MQTT oder OPC UA an Node-RED übermittelt werden und dort auf das Dashboard übertragen werden. Durch die Zuordnung der Daten zu den korrekten Komponenten kann eine digitale graphische Visualisierung des Demonstrators mit den aktuellen Prozessdaten erstellt werden. Zusätzlich dazu können die aktiven Skills oder die Betriebsarten abgebildet werden. Das Dashboard ermöglicht allerdings nicht nur eine Abbildung der Output Daten des Systems, sondern bietet die Möglichkeit das System über Inputs zu manipulieren. So können verschiedene Schaltflächen integriert werden, die Aktionen im System auslösen. Bei den Schaltflächen gibt es verschiedene Kategorien, die implementiert werden sollen. Die erste Kategorie besteht aus Schaltflächen, die vorbestimmte Prozesse starten. Durch deren Aktivierung werden entsprechend vorprogrammierte Schrittketten abgearbeitet. Die zweite Kategorie beinhaltet Schaltflächen, die die Auswahl eines zu produzierenden Produktes ermöglichen. Dafür wird eine Schaltfläche, in Form einer Drop-down Liste oder Ähnliches, zur Auswahl des Produktes benötigt und einen Start Button. Für die individuelle Zusammenstellung eines Prozesses wird eine weitere Kategorie an Schaltflächen benötigt. Zunächst wird eine Drop-down Liste oder Ähnliches benötigt, um den gewünschten Skill auszuwählen. Analog dazu ist eine Schaltfläche erforderlich, um dem Skill die gewünschten Prozessparameter zuzuordnen. Eine dritte Schaltfläche „Add“ verknüpft die beiden Eingaben und fügt es dem Prozessablauf an. Dieser Prozess ist variabel wiederholbar. Abschließend wird der finale Prozessablauf durch die Schaltfläche „Start“ an den Orchestrator übergeben und der Prozess gestartet. Um mögliche Fehlerzustände im System lösen zu können müssen alle Bestandteile zusätzlich über einen „Reset“ Button zurückgesetzt werden können.

4.7 Zusammenfassung Konzept

Aus den angesprochenen Möglichkeiten soll sich die praktische Umsetzung aus den folgenden Konzepten zusammensetzen. Das SPS-Programm wird abgewandelt von der Ablaufsteuerung der Skills über Schrittketten zu unabhängigen Skills. Diese sind abgesehen von der Übergabe der Prozessparameter nur über den individuellen Skill Call aufrufbar. Die einzelnen Skills werden hierbei als composite Skills angelegt, die durch unterschiedliche Ausführungsvarianten die Redundanz im Programm verringern. Aufgrund der fehlenden physischen Schaltflächen und dem Ansatz, die Steuerung aus der Ausführungsebene auszulagern, werden die Betriebsarten über Node-RED implementiert, trotz der nicht optimalen Umsetzung des Notaus. Die Orchestrierung erfolgt in Node-RED über die Verwendung eines Arrays für die Reihenfolge der Skills und repräsentativen Skill-Bausteinen für den Aufruf der Skills im SPS-Programm. Der Array wird über zwei Loops

vom OA abgearbeitet, bis dieser leer ist. Die Kommunikation der binären Zustände der Skills erfolgt über MQTT mit QoS1, um die Zustellung zu garantieren aber die Übertragungsdauer zu minimieren. Die Auswahl der Prozesse ist über auswählbare, vordefinierte Produkte oder über die Zusammenstellung eines individuellen Prozessablaufs möglich. Hierbei können die Skills in variabler Reihenfolge mit individuellen Prozessparameter in unbeschränkter Anzahl zu Prozessen angeordnet werden. Die Tab. 3 bietet eine Übersicht über die Ausführungsvarianten, die für unterschiedliche Bereiche möglich sind. Die grün hinterlegten Optionen werden in der praktischen Umsetzung verwendet. Die gelb hinterlegten Varianten sind Konzepte, die einen fortschrittlicheren Ansatz verfolgen, aber nicht in den Rahmen der praktischen Ausarbeitung dieser Arbeit passen. Dazu gehört die Umsetzung des Orchestration Konzepts in einer Hochsprache wie z.B. Python, die Umstellung auf einen Knowledge-Graph und die Erweiterung des Statusmodells um zusätzliche Zustände.

Tabelle 3: Übersicht über die möglichen Unterkonzepte für die Umsetzung

Kategorie	Konzept 1	Konzept 2	Konzept 3	Entscheidungs-argument
SPS-Konzept	Schrittkette	Skills + Service-Schrittkette	Skills	Flexibilität
Skills	basic	composite		geringere Redundanz
Betriebsarten	Umsetzung in Node-RED	Umsetzung im SPS-Programm	Kombination	Ansatz: Auslagerung der Steuerung
Orchestrierung	Bündelung über JS-Code in function node	Spezifische nodes mit definierten Aufgaben	Hochsprache	Einschränkung durch Node-RED
Prozesslänge	Definierte Anzahl an überschreibbaren Platzhalter- Bausteinen	Aufruf von Skill-Bausteinen		keine Einschränkung der Prozesslänge
Skill-Bausteine	Ein Skill-Baustein je Skill Funktionsbaustein	Ein Skill-Baustein je instanziierten Skill		Eindeutige Zuweisung
Prozess Datentyp	Array	Object	Knowledge-Graph	Aufruf einer unbekannten Variable möglich
Statusmodell	Keine	On/Off	+ Fehlermeldungen	Für OA notwendig
Prozessauswahl	Starre Services	Produktauswahl	Zusammenstellung	
Kommunikation	OPC UA	MQTT		Nicht im Zeitrahmen
Quality of Service	QoS0	QoS1	QoS2	Kombination aus Zuverlässigkeit und Effizienz

5. Implementierung

5.1 SPS-Programm nach Ansatz des Skill-based Engineering

Die Erstellung des SPS-Programms für die Wago SPS erfolgt über das e!Cockpit. Bei der Umsetzung des SkE Ansatzes liegt der Fokus darauf, die Programmierung auf der Ebene der SPS auf die ausführenden Elemente zu begrenzen. Das bedeutet, dass ausschließlich die Skills im SPS-Programm definiert werden sollen. Die restlichen Komponenten zur Steuerung sollen ausgelagert werden. Die Erstellung der Skills erfolgt im Unterordner „FB_Skills“. Jeder Skill wird in einem separaten Funktionsbaustein erstellt. Für die Generierung der Logik wird die Funktionsbausteinsprache (FUP) verwendet. Um, die im Konzept beschriebene Funktionalität zu erreichen, werden 15 Skills benötigt (siehe Tab. 4).

Tabelle 4: Übersicht der umgesetzten Skills

Conveyor	Crane	Production
2LBConveyor_forward / 2LBConveyor_backward	Crane_clockwise / Crane_counterclockwise	Drilling
1LBConveyor_forward	Crane_hzbackward / Crane_hzforward	Milling
1LBConveyor_end	Crane_vtdown / Crane_vtup	Stamping
	Suction	Pusher

Die Aktivierung der einzelnen Skills soll durch einen je Skill individuellen Input erfolgen, anstatt durch unterschiedliche Schritte einer Schrittkette. Dieser Input hat als Vorzeichen „SkillCall_...“. Die Umsetzung eines Skills kann der Abb. 27 entnommen werden.

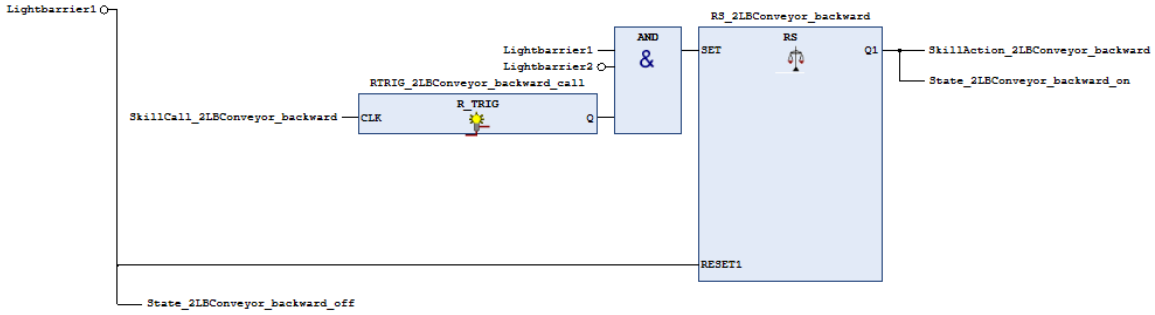


Abbildung 27: Implementierung des Skills „2LBConveyor_backward“ in FUP

Das abgebildete Netzwerk beschreibt den Skill „2LBConveyor_backward“, der die Fähigkeit hat, ein Förderband in eine bestimmte Richtung zu bewegen. Der Skill ist anwendbar für alle Förderbänder mit zwei Lichtschranken („2LB“ = 2 Light Barriers). Hauptkomponente aller Skills ist der „RS-Flipflop“. Dieser ist ein rücksetzdominanter Speicher-

Funktionsbaustein, der bei Aktivierung des „Set“ Eingangs so lange einen aktiven Output hat, bis der „Reset“ Eingang aktiv wird. Bei gleichzeitiger Aktivierung beider Inputs ist die Rücksetzbedingung dominant. Über den „Set“ Input wird festgelegt, welche Bedingungen erfüllt sein müssen, um den Skill zu aktivieren. Der Skill Call durchläuft vor dem „RS-Flipflop“ einen „R-TRIG“ Baustein. Dieser dient der positiven Flankenerkennung und identifiziert den Wechsel des Inputsignals von 0 auf 1. Dadurch wird erreicht, dass der Input Skill Call nur einmal den Skill auslöst. Der Output des „R-TRIG“ Bausteins kann nur dann erneut aktiv werden, wenn der Skill Call zwischenzeitlich deaktiviert wurde. Der Ausgang der Skills ist durch das Vorzeichen „SkillAction_...“gekennzeichnet. Entsprechend wird bei Aktivierung des Outputs der Skill ausgeführt. Für die Steuerung der Prozesse benötigt der OA Informationen über die aktuellen Zustände der einzelnen Skills. Dafür wird ein Statusmodell mit zwei Zuständen integriert, wobei die zugehörigen Variablen mit dem Vorzeichen „State“ gekennzeichnet sind. Jeder Skill setzt parallel zur Aktivierung der „SkillAction“ den Status auf „on“. Analog dazu wird der Status auf „off“ gesetzt, wenn die Rücksetzbedingung des Skills aktiv wird. Um die Skill Variablen zu reduzieren, werden in einem separaten Funktionsbaustein die Variablen „on“ und „off“, über einen RS-Flipflop, zu einer Variable mit der Endung „…_active“ zusammengeführt (siehe Abb. 28).

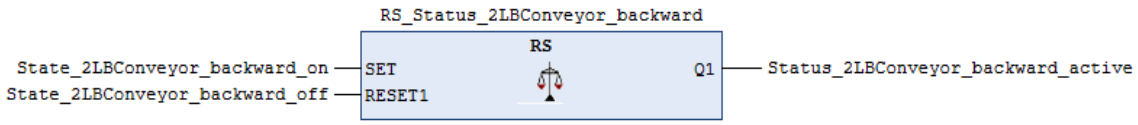


Abbildung 28: Vereinfachung des Statusmodells eines Skills auf eine Variable

Der Skill „2LBConveyor_forward“ ist vom Grundaufbau ähnlich zum Skill „2LBConveyor_backward“. Das Status Konzept, und die Aktivierung sind vom Aufbau identisch, unterscheiden sich allerdings in der Benennung der Variablen (backward → forward). Der wesentliche Unterschied zwischen den Skills ist, dass „2LBConveyor_forward“ zwei Varianten als Rücksetzbedingungen besitzt. Wie im Kapitel 4.3 beschrieben, kann der Skill theoretisch an zwei Stellen eingesetzt werden. Allerdings soll das Förderband bei der ersten Variante auf der Höhe der Lichtschranke anhalten und bei der zweiten Variante zusätzlich variabel länger fahren, um das Objekt auf eine bestimmte Position zu transportieren. Um den Skill so anzulegen, dass er in zwei Varianten ausführbar ist, müssen zwei Rücksetzbedingungen definiert werden, die über eine Entscheidungsvariable exklusiv ausgewählt werden können. Die Abb. 29 visualisiert die Programmierung der Rücksetzbedingungen des Skills. Die Entscheidungsvariable für diesen Skill ist „Time_2LBConveyor_forward_UDINT“. Die Prüfung der Variable findet im Baustein „EQ“ statt. Hier wird geprüft, ob die Entscheidungsvariable gleich null oder ungleich null ist. Wenn diese gleich null ist, erfolgt die Rücksetzung des Skills ausschließlich

über die Lichtschranke. Hierfür wird die Variable „Reset_2LBConveyor_bytime“ auf false gesetzt. Diese Variable geht im unteren Eingang des XOR-Gates negiert in die AND-Bedingung ein. Somit wird durch zusätzliches Unterbrechen der Lichtschranke der Skill zurückgesetzt. Bei der zweiten Variante wird „Reset_2LBConveyor_bytime“ am AND-Gate mit der Lichtschranke nicht negiert. Um die längere Fahrzeit mit dem Wert „Time_2LBConveyor_forward_TIME“ zu realisieren wird ein TON-Baustein benötigt. Dieser Baustein ermöglicht die Verzögerung der Rücksetzung des Skills nach Unterbrechung der Lichtschranke. Allerdings ist für die Einschaltverzögerung ein konstantes Eingangssignal notwendig, die Lichtschranke wird jedoch nur kurz unterbrochen. Aus diesem Grund ist ein zweiter RS-Flipflop erforderlich, um das konstante Eingangssignal für die Einschaltverzögerung zu erzeugen. Der RS-Flipflop wird durch Aktivierung der „State_2LBConveyor_forward_off“ Variable zurückgesetzt. Der Output des XOR-Gates führt identisch zum „2LBConveyor_backward“ Skill zum Reset Eingang des RS-Flipflops und zur „State_2LBConveyor_forward_off“ Variable. Es wäre sinnvoll, die Variablen „Time_2LBConveyor_forward_TIME“ und „Time_2LBConveyor_forward_UDINT“ als eine Variable zusammenzufassen, um so eine Input Variable zu eliminieren. Das Hindernis sind die unterschiedlichen Datentypen TIME und UDINT, die erforderlich sind für die Funktionsbausteine.

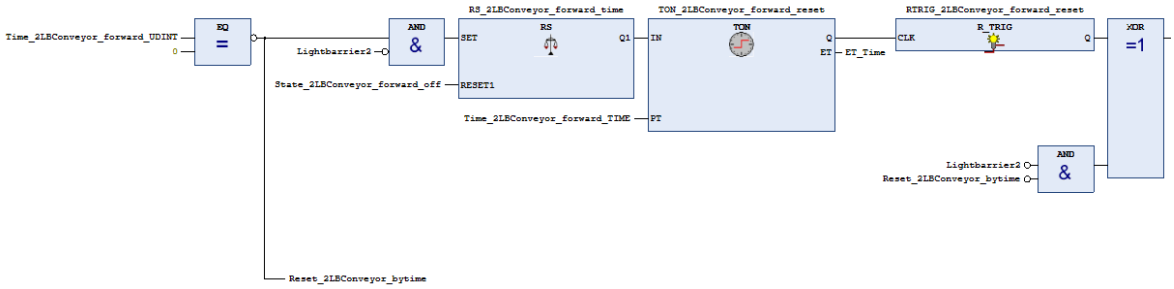


Abbildung 29: Umsetzung des Skills „2LBConveyor_forward" in FUP

Das Konzept, Skills in Varianten mit unterschiedlichen Rücksetzbedingungen aufzuteilen, wird in weiteren Skills wiederverwendet. Im Skill „1LBConveyor_forward“ wird der Ansatz der Entscheidungsvariable aufgegriffen, allerdings wird hier unterschieden in reine zeitbasierte Steuerung oder sensorbasierte Steuerung. Für die zeitbasierte Rücksetzung wird ein TON-Funktionsbaustein benötigt. Der Aufbau des Rücksetzteils des Skills kann der Abb. 30 entnommen werden.

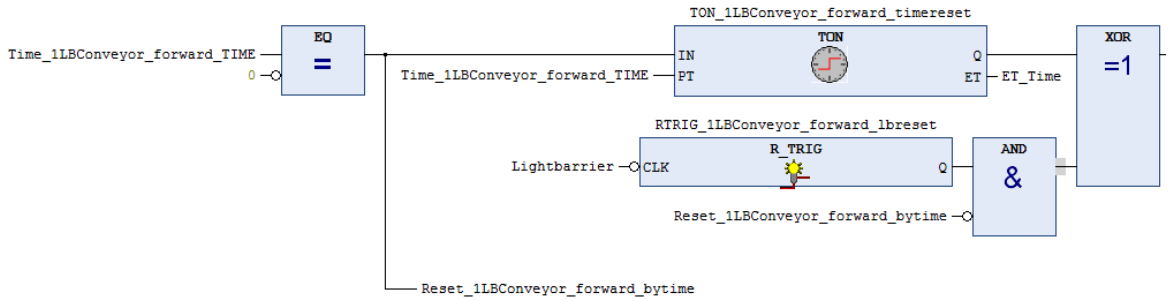


Abbildung 30: Umsetzung des Skills „1LBConveyor_forward" in FUP

Der Skill „Suction“ bildet eine Ausnahme zu den anderen Skills, denn dort werden zwei Ressourcen gleichzeitig angesteuert. Die Steuerungslogik des Funktionsbausteins ist mit dem Ausgang für das Ventil verbunden. An diesen Output ist die Steuerung für den Kompressor gekoppelt. Das bedeutet, dass bei Aktivierung des Ventils der Kompressor aktiviert wird. Eine weitere Besonderheit bei diesem Skill ist, dass zwei Inputs notwendig sind: „SkillCall_Suction_on“ und „SkillCall_Suction_off“. Das ist erforderlich aus dem Grund, dass das Vakuum über die Dauer des Transports des Krans erhalten bleiben muss, um das Objekt am Greifer zu fixieren. Daraus folgt, dass der Skill über die Dauer anderer Skills aktiv bleiben muss. Um das zu erreichen, muss der Skill zweimal angesteuert werden, einmal zum Aktivieren des Kompressors und des Ventils und einmal, um diese wieder zu deaktivieren. Nur so kann prozesssicher festgelegt werden, wann das Objekt losgelassen werden soll. Der Kompressor beginnt erst bei Aktivierung des Skills, somit ist die Bereitstellung des erforderlichen Vakuums zeitlich verzögert. Um dennoch sicherzustellen, dass das Vakuum ausreichend ist, um das Objekt sicher anzusaugen, wird die Statusmeldung an den OA verzögert. Umgesetzt wird die Verzögerung über eine Einschaltverzögerung, die vor dem Output „State_VacuumValve_on“ gesetzt wird. Somit wird sichergestellt, dass der Greifer nicht bewegt wird, bevor das Teil angesaugt ist bzw. dass ausreichend Zeit für den Aufbau des Vakuums verfügbar ist. Das Konzept des Statusmodells ist bei diesem Skill abgeändert. Die Variable „State_VacuumValve_on“ ist für den OA die Weiterschaltbedingung, nachdem das Objekt angesaugt wurde und transportiert werden kann. Die Variable „State_VacuumValve_off“ ist die Transition für den OA, die aktiv wird, sobald der Greifer deaktiviert und das Objekt abgelegt wurde. Deswegen können die Status Variablen nicht zusammengefasst werden, wie bei den restlichen Skills. Die Abb. 31 bildet die Netzwerke für den Skill „Suction“ ab.

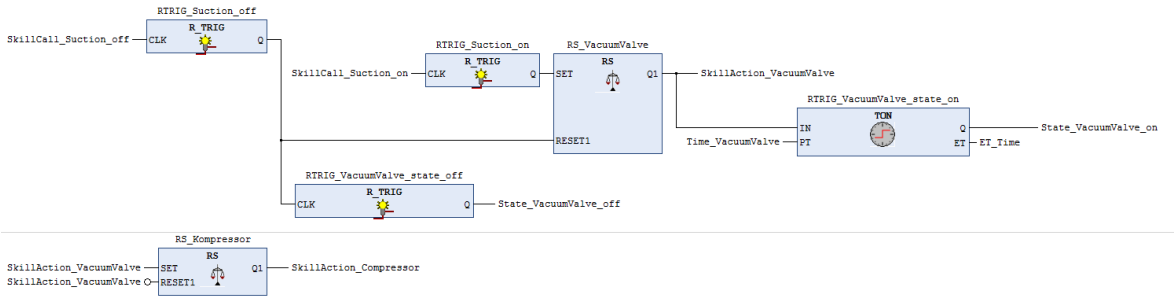


Abbildung 31: Umsetzung des Skills „Suction" in FUP

Bei der Implementierung des „Stamping“ Skills, musste dieser im Vergleich zum ursprünglichen Konzept abgeändert werden. Das ursprüngliche Konzept sieht vor, dass der Skill die Auf- und Abbewegung der Stempelmaschine bündelt. Für den Normalbetrieb ist das unproblematisch, allerdings müssen Fehlersituationen oder die Betätigung eines Notaus berücksichtigt werden. Der Skill muss bei Abbruch an einer undefinierten Position in die Ausgangsposition gebracht werden können. Um diese Eigenschaft zu erreichen, wird der Skill um zwei Varianten erweitert. Eine Variante für die Bewegung vertikal hoch und eine Variante für die Bewegung vertikal abwärts. Vergleichbar zum Skill „2LBConveyor_forward“ erfolgt die Auswahl der Variante über eine Entscheidungsvariable. Die Umsetzung erfolgt nicht über die Prüfung auf gleich null oder ungleich null, da dieser Skill drei Varianten hat, sondern jede Variante beginnt mit einem AND-Gatter. An diesem Gatter wird bei Skill Call geprüft, ob die Entscheidungsvariable den korrekten Wert hat, um die Variante zu starten. Somit wird jeder Variante ein fester Wert der Entscheidungsvariable zugeordnet. Die Zuweisung dieses Werts erfolgt in der GUI. Alle Varianten verwenden den identischen Skill Call, Statusvariablen und Skill Action Variablen. Der Aufbau der Variante, die beide Bewegungen kombiniert, kann der Abb.32 entnommen werden.

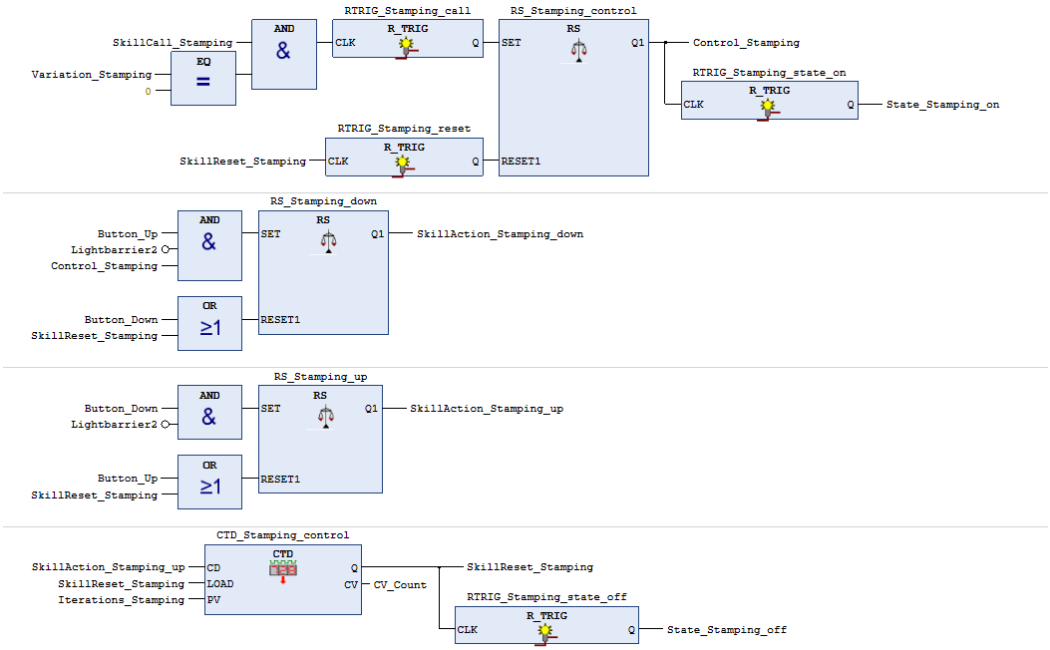


Abbildung 32: Umsetzung des Skills „Stamping" in FUP – Standard Stampingbetrieb

In diese Variante ist die Funktionalität integriert, dass der Skill in einer definierbaren Anzahl wiederholt werden kann. Umgesetzt wird dies über das oberste und unterste Netzwerk. Das oberste Netzwerk prüft anhand einer Variable „SkillReset_Stamping“, ob die Anzahl der vorgegebenen Wiederholungen schon erreicht wurde. Das unterste Netzwerk regelt den Wert dieser Variable über einen Zähl-Baustein. Dieser Baustein vergleicht den Wert der SOLL-Wiederholungen mit den IST-Wiederholungen. Sobald diese Werte gleich sind, wird die Variable „SkillReset_Stamping“ wahr und der Stamping Vorgang wird nicht wiederholt. Die Netzwerke zwei und drei führen nacheinander die vertikalen Bewegungen aus.

Die Varianten, die ausschließlich die Bewegung in eine Richtung bewirken, sind im Aufbau untereinander identisch und unterschieden sich ausschließlich durch die Benennung der Variablen und Bausteine. Die Varianten werden durch das Auslösen der Taster am oberen oder unteren Ende beendet und erfordern keine weiteren Netzwerke zur Steuerung. Der Aufbau der zwei Varianten wird repräsentativ am Beispiel der Bewegung vertikal aufwärts, in der Abb.33 dargestellt.

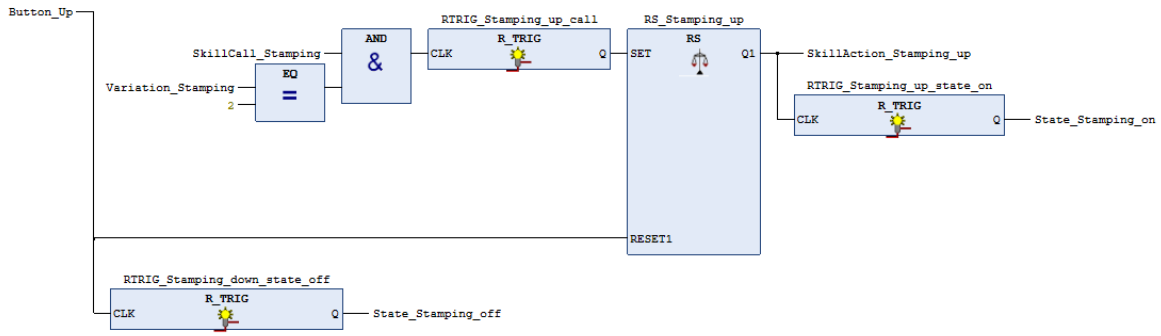


Abbildung 33: Umsetzung des Skills „Stamping" in FUP - Fahrt nach oben

Nachdem für jeden Skill-Funktionsbaustein die Netzwerke erstellt wurden, werden in separaten Funktionsbausteinen die Ausgänge zugewiesen. In diesen Funktionsbausteinen erfolgt die Zuordnung der Ausgänge der Skills zu den Outputvariablen. Darunter wird allerdings nicht die Verknüpfung der physischen Aktoren verstanden, sondern die Verknüpfung zu einer repräsentativen SPS-internen Variable. Die Steuerungslogik besteht hier in der Regel nur aus einfachen Zuweisungen von Outputs der Skills zu den repräsentativen Ausgängen. Der Zweck hierbei ist, dass jeder Ausgang nur einmal definiert wird. Die Verknüpfung der internen Variablen mit den physischen Eingängen und Ausgängen erfolgt in den Programmbausteinen. Jede Komponente wird in einem separaten Programmbaustein instanziiert.

5.2 Orchestration Agent

5.2.1 Skill-Bausteine

Wie im Kapitel zur Konzeption bereits dargestellt wurde, erfolgt die Umsetzung des OAs in Node-RED. Die Erstellung des OAs kann größtenteils abgelöst vom SPS-Programm geschehen, da die Schnittstellen mit manuell auslösbaren *Input nodes* überbrückt werden können. Der erste Versuch der Erstellung eines OAs hat den Ansatz den Orchestrierungsprozess innerhalb einer *function node* abzubilden. Der Vorteil wäre eine übersichtliche Lösung der Aufgabe. Allerdings kann der Prozess nicht durch eine *function node* abgebildet werden, aufgrund der Struktur von Node-RED. Das Programm ist standardmäßig so konzipiert, dass Nachrichten die *flows* vom Anfang zum Ende einmal durchlaufen. Das bedeutet, dass die Inputs an den einzelnen *nodes* nur beim Eintreffen verwertbar sind und nicht gespeichert werden. Eine Folge daraus ist, dass beispielsweise Logik Gatter wie „AND“ oder „OR“ nicht umsetzbar sind, außer die Nachrichten kommen zum exakt gleichen Zeitpunkt an derselben *node* an. Ähnliche Probleme verhindern dementsprechend die Erstellung des Programms innerhalb einer *function node*. Hierbei ist die Herausforderung, dass Loops integriert werden müssen, um auf die Beendigung der

Skills zu warten. Innerhalb einer *function node* kann allerdings kein neuer *node-Input* registriert werden, sobald der *node-interne Loop* aktiviert ist. Das bedeutet, dass der Loop nicht gebrochen werden kann, da die erforderliche Variable nicht in der *node* gelesen wird. Dieses Problem gilt für eingehende Nachrichten im *„msg.payload“* Format und für globale Variablen, die über Methoden abgerufen werden. Folglich könnten nacheinander die einzelnen Skills angesteuert werden, da diese zu Beginn als Array in die *node* eingehen und anschließend im Loop sind, bis alle Skills aus dem Array entfernt wurden. Die Skills könnten aber nicht beendet werden, da das Signal zum Rücksetzen von der SPS entspringt und somit außerhalb der *function node*.

Aus diesen Gründen muss ein anderer Ansatz gewählt werden. Node-RED kann um erstellte *nodes* von anderen Usern ergänzt werden, was das Programm fähiger zur Lösung spezieller Herausforderungen macht. In diesem Fall wird eine *Loop node* zusätzlich installiert. Durch eine *Loop node* kann die ursprünglich geplante *function node* in mehrere *nodes* aufgeteilt werden. Der Ansatz mit vielen *nodes*, die eine spezielle Funktion erfüllen, hat sich bewährt und wird für die gesamte Umsetzung in Node-RED verwendet. Die Erstellung des OAs startet mit der Erstellung der Skill-Bausteine in Node-RED. Für jeden Skill, der im SPS-Programm definiert ist, wird eine repräsentative Kopie in Node-RED angelegt. Das bedeutet, dass der „Stamping“ Skill, der in beiden Conveyern in Sub2 und Sub3 vorkommt, einen Skill-Baustein für Sub2 und für Sub3 zugewiesen bekommt. Nur so kann garantiert werden, dass jeder Skill an der korrekten Ressource ausgeführt werden kann. Hauptbestandteil der Skill Bausteine ist ein Loop, der dafür zuständig ist, dass der repräsentative Skill erst dann als beendet gemeldet wird, wenn der Skill in der SPS beendet ist. Ohne diese Loops würden die Skills in der SPS zwar gestartet werden aber sofort wieder als beendet deklariert werden. Die Abb. 34 veranschaulicht die Kombination von *nodes*, die vor allem für die Warteschleife bezüglich der Statusänderung der Skills eingesetzt werden, wobei die *change node* „setze msg.token“ aktuell nicht zu beachten ist.

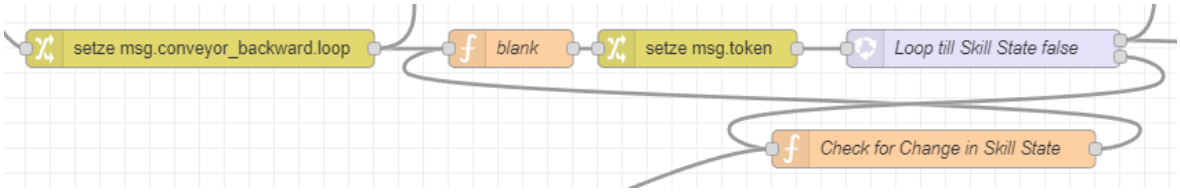


Abbildung 34: flow zur Skillbeendigung nach Zustandsänderung

Die lila gefärbte *node* beinhaltet die Prüfung der Loop Bedingung. Solange diese wahr ist, ist der Loop und somit der untere Output aktiv. Für die *Loop node* werden folgende Einstellungen vorgenommen: Die *Loop-Condition* erfolgt in JavaScript und enthält folgende Bedingung: *„msg.conveyor_backward.loop=== true“.* Die *msg. „Conveyor_backward.loop“* wird in der *change node* auf der linken Seite der Abb. 34 auf wahr gesetzt, somit ist die

Loop Bedingung direkt aktiv. Für den Ablauf des Loops ist neben *„msg.conveyor_backward.loop“* die Variable „msg.conveyor_forward.state“ erforderlich. Die Bezeichnung der Variable ist abhängig vom betrachteten Skill-Baustein. Die Variable enthält die Information über den Status des Skills in der SPS und ist somit die rücksetzende Variable für den Loop. Dementsprechend ist die Variable ein Input für die *function node*, die durchlaufen wird, solange der Loop aktiv ist. In der *function node „Check for Change in Skill State“* wird abhängig von der Status Variable die Loop Variable geändert (siehe Abb. 35).

```
1   var a = msg.conveyor_forward.loop;
2   var state = msg.conveyor_forward.state;
3
4   if (state === false){
5       a = false;
6   }else{
7       a = true;
8   };
9
10  msg.conveyor_forward.loop = a;
11  return msg;
12
```

Abbildung 35: JavaScript Code zur Prüfung auf Zustandsänderung

Über einen simplen if/else-Test kann bei Änderung des Status die Loop Bedingung auf false gesetzt werden. Dafür werden die eingehenden *Messages* zunächst *node-internen* Variablen zugewiesen. Nach Durchlauf des if/else statements müssen die relevanten internen Variablen wieder einer *„msg.“* Variable zugeordnet werden. Abschließend werden über *„return msg“*, alle *„msg.“* Variablen als Output ausgegeben. Durch diese *function node* wird es möglich aus dem Loop auszubrechen und den *flow* am oberen Output der *Loop-node* fortzusetzen. Die *function node „blank“* hat keine spezielle Funktion, sondern dient als reiner Platzhalter, der die eingehenden Nachrichten unverändert als Output weiterleitet. Im Rahmen der Erstellung des Programms hat es sich als vorteilhaft erwiesen, wenn mehrere Nachrichten an einer *node* eingehen sollen, diese an einer vorgelagerten Platzhalter *node* zusammenzuführen. Das hat beispielsweise den Hintergrund, dass die Outputs mancher *nodes* standardmäßig als *„msg.payload“* deklariert werden. Bei einer Zusammenführung zweier solcher *nodes* kann es so teilweise zu falschen Zuweisungen kommen. Der komplette Aufbau eines Skill-Bausteins kann der Abb. 36 entnommen werden.

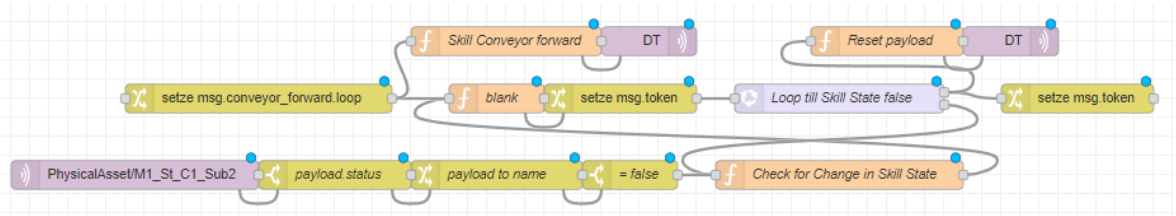


Abbildung 36: Repräsentativer flow für einen Skill-Baustein

Die vier *nodes*, die der *„Check for Change in Skill State“ node* vorgelagert sind, sind für das Empfangen des Skill Status und die Umwandlung dessen in das richtige Format zur Weiterverarbeitung verantwortlich. Zwei weitere essenzielle Bestandteile sind das Verschicken des Skill Calls und die Deaktivierung des Skill Calls. Die Aktivierung des Skill Calls findet über die *function node „Skill Conveyor forward“* und den zugehörigen MQTT-Ausgang statt. Während in der *MQTT node* die Topic festgelegt wird, definiert *die function node* die zu verschickende Nachricht (siehe Abb. 37).

```
1  var var1 = msg.sequenz;
2  var var2 = var1[0].parameter;
3  msg.payload = {
4      "SkillCall_2LBConveyor_forward_Sub2_C1": "TRUE",
5      "Time_2LBConveyor_forward_UDINT_Sub2_C1": var2
6  }
7  return msg;
```

Abbildung 37: JavaScript Code zur Definition und Formatierung der zu verschickenden MQTT-Nachricht

Die Variablen werden in *„msg.payload“* als Objekt gespeichert. Hierbei ist eine Variable für den Skill Call und eine Variable für die Prozessparameter bzw. in diesem Fall die Zeit definiert. Die Variablen müssen hier den gleichen Namen haben wie die Variablen im SPS-Programm, die überschrieben werden sollen. Die *MQTT node* verschickt nur Nachrichten die in *„msg.payload“* definiert sind. Der Skill Call ist an dieser Stelle immer wahr, deshalb kann an dieser Stelle „TRUE“ als Konstante gesetzt werden. Die Variable zur Steuerung der Zeit soll hingegen bei jeder Aktivierung variabel einstellbar sein. Aus diesem Grund wird der Wert mit einer *node-internen* Variable (var2) ersetzt. Die Information über die individuelle Zeit wird über den OA zur Verfügung gestellt. Dieser gibt den Array über die aktuelle Sequenz der Skills weiter, der als interne Variable gespeichert wird (msg.sequenz → var1). Der aktuelle Skill ist hierbei immer auf dem Index 0 des Arrays gespeichert. Die Information über den Prozessparameter ist bei jedem Skill dem Namen „parameter“ zugeordnet. Somit kann diese Infomation über „var1[0].parameter“ ausgelesen werden. Die Rücksetzung des Skill Calls über die *node „Reset payload“* ist im Aufbau und der Benennung der Variablen gleich zur aktivierenden *node*, allerdings sind beide Werte als Konstanten hinterlegt. Der Skill Call wird immer auf false gesetzt und der Parameter immer auf 0. Die *node „setze msg.token“* ist die letzt *node* im *flow* von allen Skill-Bausteinen. Diese wird nur aktiviert, wenn der Skill beendet ist. Die *node* setzt *„msg.token“* auf den Wert „2“.

Somit sendet jeder Skill-Baukasten bei Beendigung *„msg.token“* und signalisiert somit dem OA, dass der Skill beendet wurde.

5.2.2 Orchestrator

Der Aufbau des Hauptteils des Orchestrators wird in der Abb. 38 visualisiert. In der Abbildung ist der Orchestrator so konzipiert, dass drei Skills in beliebiger Reihenfolge in beliebiger Anzahl angeordnet werden können. Die Ausweitung auf mehr Skills ist ausschließlich abhängig von der zusätzlichen Implementierung von *switch nodes* für die Skill-Bausteine.

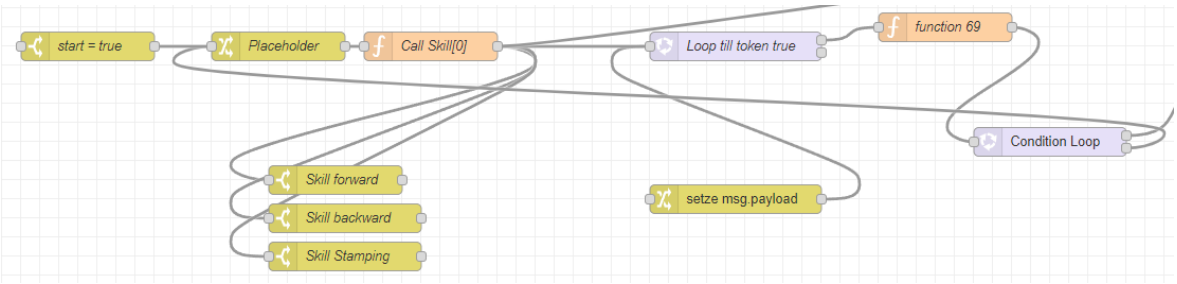


Abbildung 38: Übersicht über den flow des Orchestrators

Der Orchestrator wird über die Aktivierung der Variable *„msg.start“* aktiviert. Parallel zur Aktivierung wird dem OA in Form eines Arrays die Information übergeben, welche Skills angeordnet werden sollen. Innerhalb dieses Arrays werden die Skills als Objekte angelegt. Das Format der Bereitstellung der Informationen für die Orchestrierung wird an einem Beispiel in Abb. 39 dargestellt. Die Erstellung der Inhalte in diesem Format entsteht über das Dashboard. Hierbei wird der Index 0 – 2 des Arrays belegt. Auf jedem Index wird der gewählte Skill als Objekt mit zwei Werten angelegt. Diese Art der Speicherung der Skills ermöglicht den Orchestrationsprozess. Jeder Skill ist individuell identifiziert, denn nur so kann der korrekte Skill-Bautein aufgerufen werden, allerdings erschwert das die Zuordnung von geforderten Skills zu den Bausteinen. Durch die Speicherung in Form von Arrays kann der Aufruf vereinfacht werden, da nicht der Name des nächsten Skills bekannt sein muss, sondern nur der Index im Array. Die Methode *„shift()“* ermöglicht zudem die Löschung des Index 0. Somit ist es möglich, dass der aufzurufende Skill auf Index 0 definiert wird. Nach Beendigung des Skills wird das aktuelle Objekt auf Index 0 gelöscht und das nachfolgende Objekt rückt auf die Position auf Index 0 vor. Aufgrund dessen, dass alle Skill-Objekte im Aufbau identisch sind, können die einzelnen Werte wiederum über die Namen, wie „skill“ oder „parameter“ ausgelesen werden. In der Abb. 39 kann der Wert „conveyor_forward“ über die Adresse „sequenz[0].skill“ gelesen werden.

```
var sequenz = [
    {
    "skill": "conveyor_forward",
    "parameter": 0
    },
    {
    "skill": "conveyor_backward",
    "parameter": 0
    },
    {
    "parameter": 6,
    "skill": "stamping"
    },
]

msg.sequenz = sequenz;
return msg;
```

Abbildung 39: Beispielhafter JavaScript Code zur Speicherung der Sequenz und Parameter der Skills des aktuellen Prozesses

Anschließend an die Prüfung, ob *„msg.start“* wahr ist, erfolgt wiederum eine Platzhalter *node*. Die *function node „Call Skill“* ist dafür zuständig, dass der richtige Skill aktiviert wird. Es wird der Array, der die Reihenfolge und Informationen über die Skills enthält, zu einer internen Variable umgewandelt. Anschließend wird der Index 0 des Arrays gelesen und der Variable *„msg.call“* zugewiesen. Zudem wird der aktuelle Stand des Arrays in der globalen Variable „sequenz“ gespeichert. Das ist erforderlich, da diese Information ansonsten beim Durchgang durch die nachfolgende *Loop node* verloren geht. Der Inhalt der *function node* ist in der Abb. 40 visualisiert.

```
var var1 = msg.sequenz;

msg.call = var1[0].skill;
global.set("sequenz", var1);
return msg;
```

Abbildung 40: JavaScript Code zur Extraktion des aktuellen Skills inklusive Parametern

Die Variable *„msg.call“* dient anschließend als Entscheidungsvariable. Die Entscheidungsbedingung wird statt in einer *function node* mit if-statement, über eine *switch node* je Skill-Baukasten integriert. Hierbei wird *„msg.call“* auf den Wert geprüft. Die *switch nodes* leiten dementsprechend nur den *flow* weiter, wenn die eingestellte Bedingung übereinstimmt. Die *switch node „Skill forward“* leitet beispielsweise nur den *flow* weiter, wenn *„msg.call“* gleich *„Sub2_Conveyor_forward“* ist. Für die Ansteuerung eines weiteren Skill-Bausteins muss entsprechend eine weitere *switch node* integriert werden. Die *function node* regelt in Verbindung mit den *switch nodes* maßgeblich die korrekte Ansteuerung der Skill-Bausteine. Parallel zur Ansteuerung der Skill-Bausteine wird der erste Loop aktiviert.

Dieser Loop ist so lange aktiv, bis der aktivierte Skill beendet wurde. Die Loop Bedingung muss von allen Skill-Bausteinen zurückgesetzt werden können. Aus diesem Grund sendet jeder Skill-Baustein nach Beendigung das gleiche Signal. Sobald der aktuelle Skill beendet und somit der Loop unterbrochen ist, wird die *function node „Deleting Skill[0]“* aktiv. Innerhalb dieser *node* wird über die Methode *„global.get()“* die Sequenz der Skills abgerufen. Anschließend wird der Index 0 des Arrays über die Methode „shift()“ gelöscht, da dieser bereits abgearbeitet wurde. Nachdem der aktuelle Skill entfernt wurde, wird geprüft, ob der Array leer ist oder ob noch Skills abgearbeitet werden müssen. Diese Prüfung erfolgt über die Ermittlung der Länge des Arrays. Die Abb. 41 veranschaulicht den JavaScript Code der *function node*.

```
1   var var1 = global.get("sequenz");
2   var var2 = true;
3   var1.shift();
4   global.set("sequenz", var1);
5
6   if (var1.length === 0){
7       var2 = false
8   };
9
10  msg.repeat = var2;
11  msg.sequenz = var1;
12  return msg;
```

Abbildung 41: JavaScript Code zur Überprüfung auf nachfolgende Skills

Das Ergebnis dieser Prüfung ist die Loop Bedingung für die nachfolgende *Loop node*. Wenn der Array weitere Skills beinhaltet, ist der untere Output der *Loop node* aktiv. Somit wird der *flow* wieder an der *„Placeholder“ node* fortgesetzt. Der Prozess ist im Ablauf identisch, mit dem Unterschied, dass der Index 0 des Sequenz-Arrays sich verändert hat und somit ein anderer Skill-Baustein aufgerufen wird. Dieser Prozess wird so lange wiederholt, bis alle Skills abgearbeitet wurden und die Länge des Arrays gleich null ist. In diesem Fall wird der obere Output der *Loop node* aktiv und der Orchestrationsprozess ist beendet. Die nachfolgenden *nodes* haben zwei Funktionen: Die *node „Reset Skill order“* ist dafür verantwortlich, dass nach Durchlauf des Orchestration Prozesses die ursprünglich eingegebene Anordnung der Skills zurückgesetzt wird, um die Erstellung eines neuen Prozesses zu ermöglichen. Die restlichen *nodes* sorgen für optisches Feedback auf dem Dashboard. Es wird unterschieden in die States *„Running“, „Process ended“* und *„Idle“.* Der State „Running“ wird ausgelöst sobald die *function node „Call Skill[0]“* aktiv ist. Nach Beendigung des Orchestration Prozesses wird der Status „Process ended“ angezeigt. Daran gekoppelt ist eine Zeitschaltung, die nach fünf Sekunden den Status von *„Process ended“* auf *„Idle“* ändert. Der gesamte Aufbau des Orchestrators kann der Abb. 42 entnommen werden.

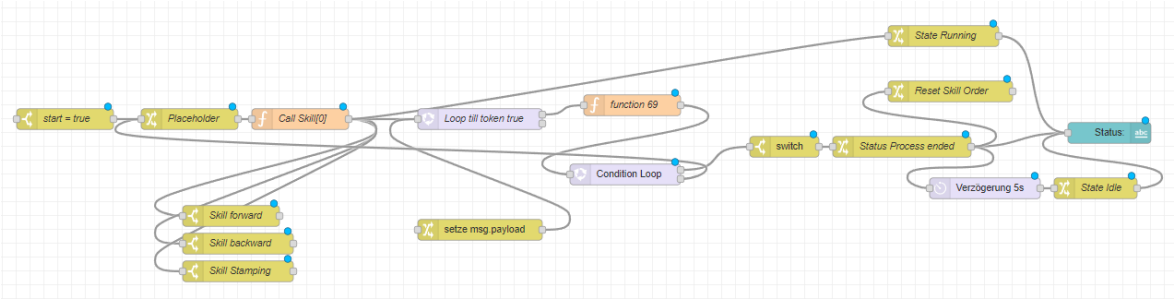


Abbildung 42: Übersicht über den flow des OAs

5.2.3 Graphical User Interface

Die Steuerung und Visualisierung des Systems erfolgt über das Node-RED Dashboard. Für die Steuerung werden drei Unterkategorien erstellt. Die erste Kategorie behandelt die Möglichkeit zur Erstellung von individuellen Prozessen. Die zweite Kategorie beinhaltet die Auswahl von vordefinierten Prozessen und die letzte Kategorie sind einzelne vordefinierte Prozesse, die immer gleich sind und die durch das Betätigen eines Buttons aktiviert werden. Prozesse, die immer gleich sind, sind beispielsweise die Fahrt in die Grundstellung. Die Zusammenstellung der individuellen Prozesse erfordert vier Eingabemöglichkeiten. Zur Auswahl des Skills wird eine Drop-down Liste mit allen verfügbaren Skills integriert. Für die Eingabe der Prozessparameter wird eine Freitext-Eingabefläche integriert. Der Skill „*Stamping*“ erfordert als einziger Skill einen zweiten Prozessparameter, der die Variante festlegt. Darum wird, wenn dieser Skill in der Drop-down Liste ausgewählt wird, eine zweite Eingabefläche eingeblendet. Um den ausgewählten Skill mit dem dazugehörigen Prozessparameter in den Prozess aufzunehmen, wird ein „Add“ Button benötigt. Sobald alle gewünschten Skills dem Prozess hinzugefügt wurden, wird der zusammengestellte Prozess über den „Start“ Button an den Orchestrator weitergegeben und der Prozess wird gestartet.

Für ein Feedback über die aktuelle Länge des zusammengestellten Prozesses wird über das Dashboard die Anzahl der bereits hinzugefügten Skills angezeigt. Neben der Länge des zusammengestellten Prozesses wird der aktuelle Status angezeigt. Unterschieden wird in die Zustände *„Idle“,* „*Running*“ und *„Process ended“*. Um mögliche Fehleingaben zu korrigieren, kann über den Button „Reset Order“ der zusammengestellte Prozess zurückgesetzt werden. Für mögliche Fehler im Ablauf wird ein zusätzlicher Button integriert, der es ermöglicht die Skill-Bausteine zurückzusetzen. Die Herausforderung bei der funktionalen Umsetzung ist, dass die Auswahl aus der Drop-down Liste und der Start Button, bei direkter Verknüpfung, den Start des *flows* auslösen können. Somit würde immer eine unvollständige Version des zusammengestellten Prozesses an den Orchestrator weitergegeben werden. Deswegen muss die Lösung so konzipiert werden, dass nur der

flow des Start Buttons mit dem Orchestrator verbunden ist. Der Informationsaustausch bezüglich des ausgewählten Skills und der Parameter muss über globale Variablen erfolgen. Demnach wird an die *Input nodes* jeweils eine *function node* angehängt. Darin wird eine individuelle globale Variable definiert und dieser den aktuellen Input Wert zugeordnet. Die globalen Variablen werden in der *node „global.get all parameters“* gelesen. Die *function node* beinhaltet eine Vorlage, in welchem Format die Informationen an den Orchestrator weitergegeben werden müssen. Die Vorlage enthält interne Variablen, die mit den Werten der globalen Variablen überschrieben werden. Die globalen Variablen können für einen Prozessschritt so lange geändert werden, bis der *„Add“* Button aktiviert wird. Eine *switch node* ermöglicht, dass nur durch Aktivierung dieses Buttons die *node „global.get all parameters“* aktiviert wird. Dadurch wird die Vorlage mit den aktuell gesetzten globalen Variablen im Objekt Format in den Array *„sequenz“* gepusht. Der Array wird nachfolgend ebenfalls auf eine Variable geschrieben. Dadurch, dass diese Variable zu Beginn der *function node* ausgelesen wird und das neue Objekt nur angehängt, statt überschrieben wird, kann dieser Prozess beliebig oft wiederholt werden. Somit kann ein Prozess eine beliebige Anzahl an Skills mit individuellen Parametern in beliebiger Reihenfolge beinhalten. In der *function node* wird zudem die aktuelle Länge des Arrays über *„sequenz.length“* ermittelt und über eine *Dashboard Output node* visualisiert. Der Aufbau des Codes dieser *function node* wird in Abb. 43 abgebildet.

```
1   var var1 = global.get("selection");
2   var sequenz = global.get("sequenz");
3   var var2 = global.get("parameter");
4   var var3 = global.get("variation");
5
6   var newelement = {
7       "skill": var1,
8       "parameter": var2,
9       "variation": var3,
10  };
11
12  sequenz.push(newelement);
13  msg.selection = sequenz;
14  msg.length = sequenz.length;
15  global.set("sequenz",sequenz);
16  return msg;
```

Abbildung 43: JavaScript Code zur Bündelung der Skill-Parameter als Objekt und Eingliederung in die Prozessfolge

Sobald die gewünschte Anordnung des Prozesses erreicht ist, kann durch den *„Start“* Button der *„sequenz“* array an den Orchestrator weitergegeben werden und der Prozess gestartet werden. In der *node „global.get final sequenz“* wird der *„sequenz“* Array gelesen und auf *„msg.sequenz“* überschrieben. Nach Ende des Prozesses wird der Inhalt des „sequenz“ Arrays gelöscht, um einen neuen Prozess zu ermöglichen. Ansonsten würde der

alte Prozess nur um die hinzugefügten Skills erweitert werden. Die selbe Funktion erfüllt die Aktivierung des Buttons *„Reset Order“*. Hierbei wird die globale Variable *„sequenz“* mit einem leeren Array überschrieben und somit alle hinzugefügten Skills entfernt. Der Button *„Reset all Skills“* simuliert das rücksetzende Signal für die Skill-Bausteine, das im Normalbetrieb über die SPS verschickt werden soll. Der Button ist mit den einzelnen Skill-Bausteinen verbunden und ermöglicht bei undefinierten Zuständen, dass die Loops der Bausteine beendet werden können. Die Abb. 44 veranschaulicht die Verbindungen zwischen den einzelnen *nodes* für die behandelten Abläufe.

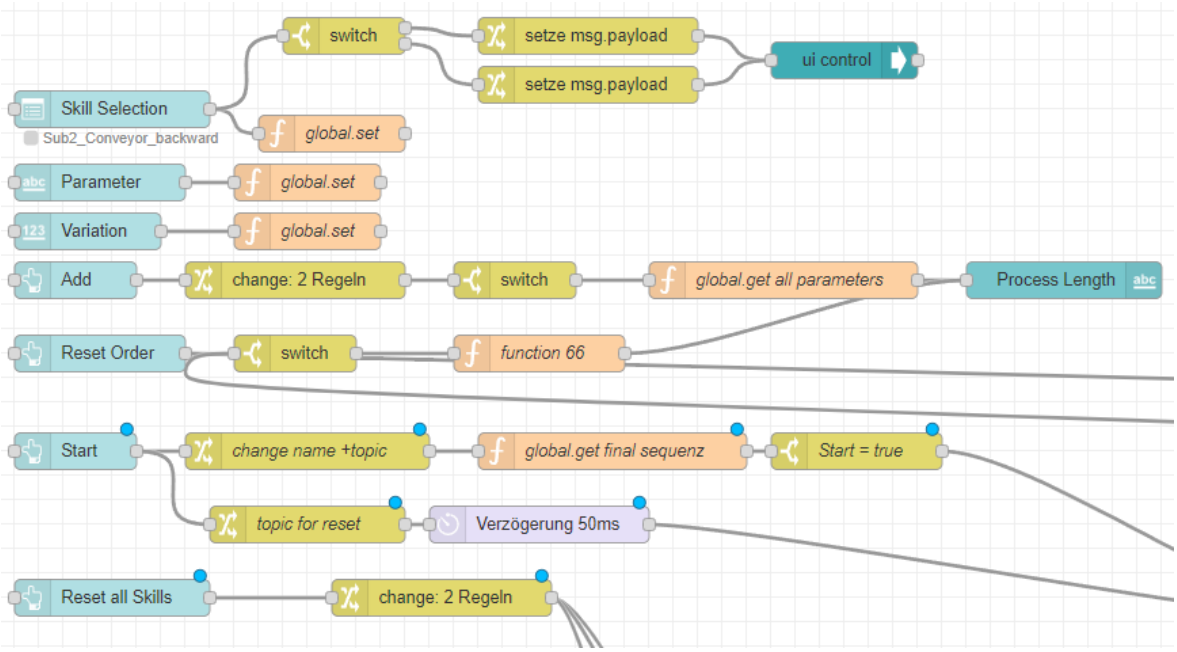


Abbildung 44: Übersicht des flows zur Erstellung von individuellen Prozessen

Die Kombination aus *switch, change und ui control nodes* im Anschluss an *„Skill Selection“* ermöglicht das Einblenden und Ausblenden der zusätzlichen Eingabefläche *„Variation“* für den Skill Stamping. Über die *switch node* wird festgestellt, ob der Skill Stamping ausgewählt wurde. Falls dies zutrifft, wird *„msg.payload“* zu folgendem Wert gesetzt: *„{"group":{"show":["CustomProcess_Variation"]}}“.* Dieser Wert erwirkt über die *ui control node*, dass die Eingabefläche *„Variation“* im Dashboard angezeigt und manipulierbar wird. In allen anderen Fällen, in denen nicht der Stamping Skill ausgewählt wird, wird *„msg.payload“* auf den Wert *„{"group":{"hide":["CustomProcess_Variation"]}}“* gesetzt, das über die *ui control node* das Ausblenden der Schaltfläche bewirkt.

Die zweite Kategorie der Eingabemöglichkeiten ist die Auswahl von vordefinierten Produkten. Für den Anwendungsfall wurden den Produkten rot, weiß und blau jeweils zufällige und unterschiedliche Prozesse zugewiesen. Die Auswahl aus diesen Produkten erfolgt über eine Drop-down Liste im Dashboard. Nachdem die Auswahl getroffen wurde, kann durch Betätigung des „Start“ Buttons, der vordefinierte Prozess des ausgewählten

Produkts an den Orchestrator weitergegeben und der Prozess gestartet werden. Umgesetzt wird dies durch die Unterscheidung der ausgewählten Produkte über eine *switch node*. Für jedes Produkt folgt eine *function node*, in der der vordefinierte Prozess inklusive aller Parameter in einem Array definiert ist. Abhängig davon, welches Produkt ausgewählt wird, wird der entsprechende Array auf eine globale Variable *„list“* geschrieben. Analog zum individuellen Prozess erfolgt über den *„Start“* Button das Auslesen dieser globalen Variable mit dem aktuellen Array und der Beginn des Prozesses. Die Anordnung der *nodes* für diesen Ablauf ist in der Abb. 45 dargestellt.

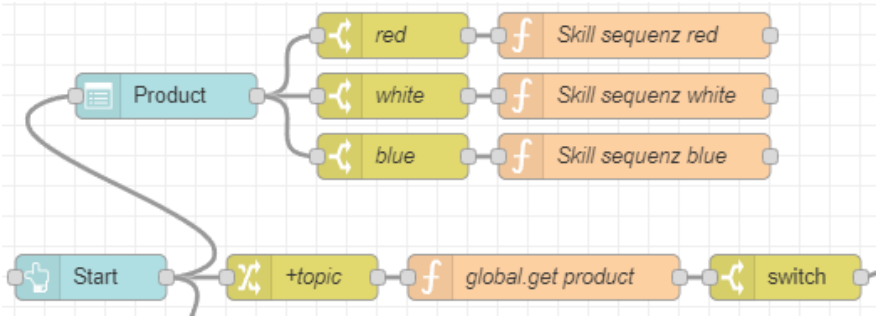


Abbildung 45: Visualisierung des flows zur Produktauswahl

Die dritte und simpelste Kategorie, wie Prozesse über das Dashboard gestartet werden können ist über vordefinierte Abläufe, die direkt auf die Aktivierung eines Buttons reagieren. Diese Kategorie wurde für die Initialisierung aller Aktoren verwendet. Bei der Aktivierung des Buttons *„Initial State“* wird sofort der vordefinierte Prozess gestartet. Somit ist bei dieser Kategorie nur eine *Input node* erforderlich. Umgesetzt wird der Ablauf über die *node „Initial State Skills“,* worin alle Skills, die initialisiert werden müssen, in einem Array definiert sind. Die Skills für die Aktivierung des Förderbands müssen nicht initialisiert werden, da die Position des Förderbands belanglos ist. Aus diesem Grund wird nur die Stamping machine initialisiert. Bei Aktivierung des *„Initial State“* Buttons wird der Array auf *„msg.sequenz“* überschrieben und der Prozess wird gestartet. Die detaillierte Beschreibung der Umsetzung der Betriebsarten erfolgt im Kapitel 5.3. Die Abb. 46 bildet die Benutzeroberfläche des Dashboards ab.

Die Gemeinsamkeit aller Kategorien der Prozessauslösung ist, welche Variable am Ende beschrieben wird. Alle Prozessinformationen müssen in Form eines Arrays auf *„msg.sequenz“* gespeichert werden, um den Prozess durchzuführen. Zudem müssen die Indexe des Arrays als Objekte in der oben bereits beschriebenen Form definiert werden.

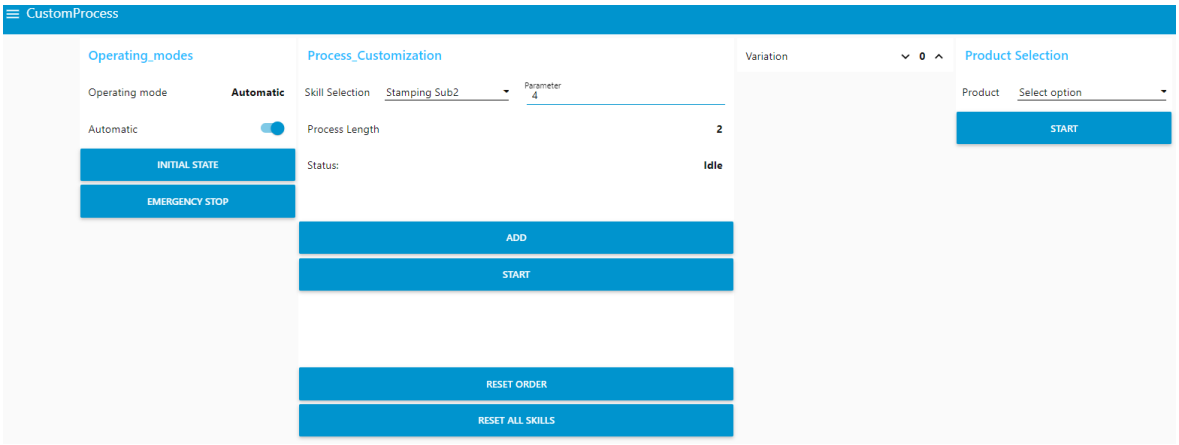


Abbildung 46: Abbildung des Dashboards zur Steuerung der Prozesse

5.2.4 Abläufe der auswählbaren Produkte

Die in der drop-down Liste auswählbaren Produkte bilden vordefinierte Services ab. Die Produkte besitzen individuelle, repräsentative Abläufe. Bei der Auswahl des blauen Teils fährt bei Prozessstart das Objekt auf dem Förderband vorwärts und anschließend zurück. Bei dem Ablauf wird nur die Ressource Förderband und nicht die Stempelmaschine verwendet. Der Ablauf des roten Teils ist folgendermaßen gestaltet: Das Objekt fährt auf dem Förderband vorwärts und anschließend zurück. Nach der nächsten Fahrt vorwärts erfolgt ein Stempelvorgang mit vier Wiederholungen. Abschließend wird das Objekt in die Ausgangslage zurückbefördert. Der Ablauf des weißen Teils ergänzt den des roten. Hierbei erfolgt der gleiche Ablauf wie beim roten Teil, allerdings erfolgen nur zwei Wiederholungen des Stempel-Prozesses. Zusätzlich wird bei dem weißen Teil das Objekt erneut unter die Stempelmaschine befördert und einmal gestempelt. Abschließend wird das Objekt in die Ausgangsposition transportiert. Die Prozesse sind frei wählbar und nicht abhängig von der Farbe des physischen Objekts im Demonstrator.

5.3 Betriebsarten

Die Implementierung der Betriebsarten erfolgt für den Anwendungsfall ausschließlich über Node-RED. Die Entscheidung dazu basiert auf folgenden Argumenten: Die Hardwareseite des Demonstrators verfügt über keine physischen Schaltflächen, um den Zustand des Systems zu manipulieren. Das bedeutet, dass kein physischer Notaus vorhanden ist. Die gesamte Steuerung des Demonstrators erfolgt somit über das Dashboard bzw. die GUI. Eine Implementierung der Betriebsarten würde einen zusätzlichen Informationsaustausch zwischen Node-RED und der SPS erfordern. Die daraus folgenden längeren Reaktionszeiten verhindern wiederum eine verlässliche Integrierung eines Notaus. Zudem die Arbeit, durch die Erstellung eines OAs, die Steuerung und Komplexität aus der SPS

auslagern soll, passt es in das Konzept der Arbeit die Betriebsarten in den OA zu integrieren. Durch die Umsetzung in Node-RED kann die Funktionsweise des Orchestrators entsprechend der aktiven Betriebsart angepasst werden. Die Betriebsarten, die für den Anwendungsfall verwendet werden, sind der Automatikbetrieb (F1), der Initialzustand (A1), der Stillstand nach dem aktuellen Zyklus (A2), die Fahrt in den Initialzustand (A6) und der Notuas (D1).

Die Auswahl der Betriebsarten erfolgt über drei Schaltflächen auf dem Dashboard. Hierzu gehören ein switch Button und zwei Buttons. Die zwei Buttons lösen die Fahrt in den Initialzustand und den Notaus aus. Über den switch Button wird primär der Automatikbetrieb aktiviert, regelt allerdings zudem die Betriebsarten (A6) und (A1). Durch das Verschieben des switch Buttons auf die aktive Seite wird der Automatikbetrieb aktiv und ermöglicht die Durchführung der Prozesse. Diese Funktionalität wurde durch eine externe Node-RED *node* erreicht. Die verwendete *node* ermöglicht die klassische Verwendung von Logik Gattern wie AND, OR und XOR. Die *node* speichert den letzten Input-Wert einer Topic, ohne zusätzliche *nodes*. Innerhalb der *node* wird eine definierbare Anzahl an Topics verarbeitet. Die *node* ist AND, OR und XOR Gatter zugleich, wobei die gewünschte Funktion durch die Wahl des entsprechenden Output Knotens erfolgt. Aufgrund der Speicherung des booleschen Zustands einer Topic ist für jede Topic ein true und false Input erforderlich.

Durch die Verwendung der *node* soll erreicht werden, dass der Orchestrator nur agieren kann, wenn die Betriebsart auf Automatik gestellt ist. Das wird erreicht, indem der *flow* des „Start“ Buttons nicht direkt mit dem Orchestrator verbunden wird, sondern über die zusätzliche *node* geleitet wird. Der *flow* des switch buttons wird ebenfalls mit dieser *node* verbunden. Wird der AND-Output der *node* mit dem Orchestrator verbunden, starten die Prozesse nur, wenn bei Aktivierung des „Start“ Buttons der Automatikbetrieb aktiv ist. Während die Topic für den Automatikbetrieb manuell über das Ein- und Ausschalten gesetzt wird, wird das Zurücksetzen der Topic für den „Start Button“ automatisch über eine Zeitverzögerung nach der Betätigung des Buttons geregelt. Die Anordnung der *nodes* für diese Funktionalität kann der Abb. 47 entnommen werden. Der *flow* zeigt die Integration der Betriebsarten für den Custom Process. Die Umsetzung für die Auswahl über die Drop-down Liste erfolgt identisch. Durch die *flow-basierte* Eigenschaft von Node-RED hat der switch Button zudem die Funktion, dass, wenn der switch deaktiviert wird, der aktuelle Prozess beendet wird, aber kein Neuer begonnen werden kann. Diese Eigenschaft entspricht der Betriebsart „Stillstand angefordert nach Ende des aktuellen Zyklus“. Der Button *„Initial State“* löst, wie im Kapitel *Grapical User Interface* bereits beschrieben die Fahrt in die Grundstellung aus. Dieser *flow* ist direkt mit dem Orchestrator verbunden, da

die Betriebsart nicht abhängig vom Automatikbetrieb ist. Bei Aktivierung des *„Initial State“* Buttons wird parallel der switch Button in die inaktive Position gebracht. Demnach ist das System bei aktiviertem switch button im Automatikbetrieb und bei deaktiviertem switch button in der Grundstellung.

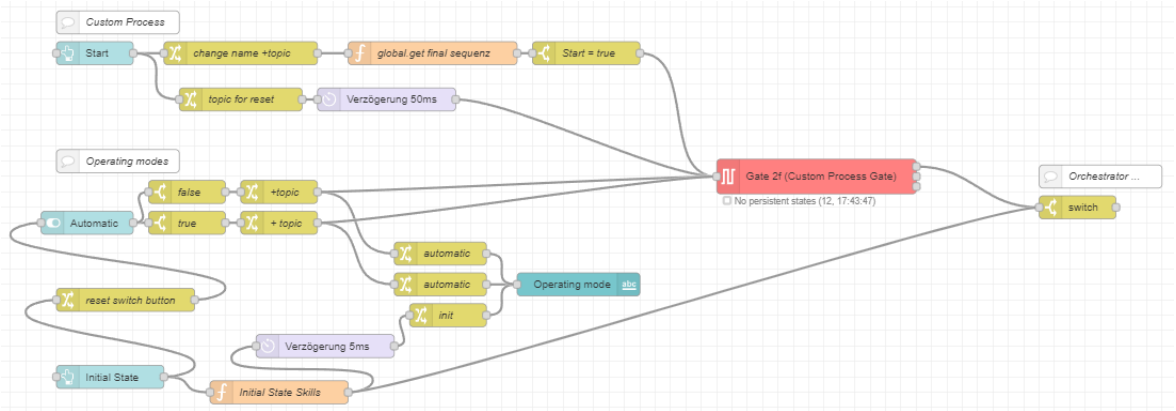


Abbildung 47: Übersicht des flows zur Integrierung von Betriebsarten

Die Betriebsart für den Notaus wurde für den Anwendungsfall implementiert, obwohl die Variante in der Form nicht für den reellen Einsatz geeignet ist. Durch die Betätigung des *„Emergency Stop“* Buttons werden alle Skill-Bausteine manuell beendet und der Array *„sequenz“* wird mit einem leeren Array überschrieben. Somit werden alle Skills angehalten und kein neuer gestartet. Zusätzlich wird durch Betätigung des Notaus der Automatikbetrieb deaktiviert. Die Aktivierung des Notaus erfordert, dass nachfolgend nur die Fahrt in die Grundstellung aktiviert werden kann, um das System wieder in eine kontrollierte Ausgangslage zu bringen. Um das zu erreichen, werden bei Aktivierung des *„Emergency Stop“* Buttons alle Eingabeflächen auf dem Dashboard ausgeblendet, außer die Bereiche für die Betriebsarten. Sobald der Button *„Initial State“* betätigt wird sind alle Komponenten erneut eingeblendet. Dadurch ist es nicht möglich neue Prozesse zu starten, sofern das System nicht initialisiert wurde. Das Ausblenden erfolgt über die Eingabe folgendes *„msg.payload“* Werts in eine *ui control node*:

„{"group":{"hide":["CustomProcess_Process_Customization","CustomProcess_Product_Selection","CustomProcess_Variation"]}}“.

Für das Einblenden wird *„hide“* durch *„show“* ersetzt. Der graphische Aufbau des *flows* für den Notaus kann der Abb. 48 entnommen werden.

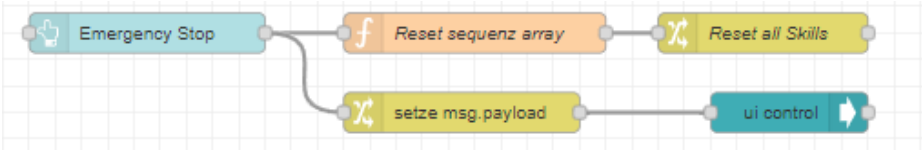


Abbildung 48: flow zur Umsetzung eines Notaus

5.4 Kommunikation

Die Umsetzung der Kommunikation zwischen der SPS und Node-RED sollte über MQTT als Publish/Subscribe Modell und OPC UA als Server/Client Modell erfolgen. Durch den direkten Vergleich können die Protokolle auf deren Eignung, vor allem hinsichtlich der Durchlaufzeit, geprüft werden. Aufgrund von Lizenzproblemen bei der Implementierung von OPC UA im Wago System, kann im Rahmen dieser Arbeit OPC UA nicht implementiert werden.

Die Umsetzung der Kommunikation zwischen der SPS und Node-RED über MQTT erfolgt folgendermaßen. Zu Beginn werden die Teilnehmer festgelegt. Die SPS und Node-RED agieren als Clients, die Publishen und Subscriben können. Als MQTT-Broker wird *„mqtt://mq.jreichwald.de:1883“* verwendet. Anschließend muss definiert werden, welche Informationen übertragen werden sollen. Zu den erforderlichen Informationen gehören die Skill Calls, die Zeitvariablen für die Zeitsteuerung, die Entscheidungsvariablen für die Variantenauswahl und die Statusvariablen der Skills. Zu den Informationen, die für die aktuellen Abläufe nicht erforderlich aber für eine visuelle Abbildung der Prozesse im Dashboard notwendig sind, gehören die Zustandsvariablen der Aktoren und Sensoren. Die Festlegung des QoS ist eine weitere erforderliche Bedingung für die Kommunikation. Hierfür wird QoS1 verwendet, da diese Variante eine gute Kombination aus Zuverlässigkeit und Übertragungsgeschwindigkeit bietet. Für die Kommunikation müssen die Clients eingerichtet werden.

Die Einrichtung des Node-RED Clients erfolgt über die standardmäßig integrierten *MQTT nodes* für das Publishen und Subscriben. Die *nodes* verfügen über vorgefertigte Eingabeflächen für die erforderlichen Informationen. Dazu gehört die Serveradresse des Brokers (*„mqtt://mq.jreichwald.de:1883“*), die Topic, die für jede *node* verschieden gesetzt werden kann, und der QoS. Die *Subscribe nodes* verfügen zudem über die Fähigkeit eine definierte Topic zu subscriben oder alle Topics dynamisch zu subscriben. Für das Versenden von Nachrichten über die *Publish nodes* müssen die Informationen in *„msg.payload“* als JSON gespeichert werden. Sobald der *flow* die *Publish node* erreicht, wird der Inhalt von *„msg.payload“* mit der definierten Topic an den Broker gepublished. Die *Subscribe nodes* sind so eingestellt, dass eine definierte Topic subscribed wird. Sobald eine Nachricht mit dieser Topic am Broker eintrifft, wird diese Nachricht als *„msg.payload“* in Node-RED erfasst. Der Eingang einer Nachricht startet zudem den *flow*, falls weitere *nodes* verbunden sind.

Die Einrichtung der SPS als Client erweist sich als umfangreicherer Prozess, da hierbei keine standardisierten Bausteine verfügbar sind. Zudem muss zusätzlich die

„WagoAppCloud“ Library, die die *„Native MQTT“* Library enthält, hinzugefügt werden. Die Bibliotheken sind erforderlich für die Funktion als MQTT-Client. Hierbei gibt es Konflikte durch unterschiedliche Versionen von Bibliotheken. Für die Fehlerbehebung muss die Platzhalterbibliothek *„WagoTypesCommon“* von der aktuellen Version 1.6.0.0 auf die Version 1.5.3.0 gebracht werden. Die Adresse des Brokers wird über das Web-based Management definiert, der restliche Anteil wird in der SPS direkt definiert. Die erforderlichen Schritte für das Publishen und Subscriben unterscheiden sich in der SPS. Die Implementierung von MQTT erfolgt individuell für jede Ressource. Für die Ressourcen Conveyor1 und Stamping1 im Subsystem 2 wird die Publish und Subscribe Fähigkeit umgesetzt. Zudem werden die Skill States gepublished. Am Beispiel des Conveyor1 wird zunächst auf den Prozess der Einrichtung der Publish Funktionalität eingegangen: Für das Publishen sind folgende Komponenten erforderlich: Eine allgemeine DUT „sdtPayload“, eine allgemeine DUT „sdtTag“ und einen Programmbaustein je Ressource, der einen Aktionsbaustein für jeden Sensor und Aktor enthält. Unter sdtPayload wird für jede Information, die gepublished wird, eine individuelle Variable definiert. Für den betrachteten Conveyor sind das zwei Aktoren, zwei Sensoren und zwei States. Jeder der Variablen ist der Datentyp „sdtTag“ zugeordnet. (siehe Abb.49.).

```
LB1_C1_Sub2 : sdtTag; // Lichtschranke vorne
LB2_C1_Sub2 : sdtTag; // Lichtschranke hinten
//Aktoren
M1_C1_Sub2  : sdtTag; // Motor zur Stampinmachine
M2_C1_Sub2  : sdtTag; // Motor zum Übergabepunkt
//States
M1_St_C1_Sub2   :sdtTag;  // State aktiver Skill forward
M2_St_C1_Sub2   :sdtTag; // backward
```

Abbildung 49: Zuordnung des Datentyps „sdtTag“

```
TYPE sdtTag :
STRUCT
    Component       : STRING;
    TypeOfMachine   : STRING;
    Status          : BOOL;
END_STRUCT
END_TYPE
```

Abbildung 50: Vorlage des Datentyps „sdtTag"

Der Datentyp *„sdtTag“* wird im DUT *„sdtTag“* definiert. Dort wird eine Vorlage definiert, wie die einzelnen Nachrichten aufgebaut werden, wenn diese gepublished werden. Die Nachrichten werden in *„Component“*, *„TypeOfMachine“* und *„Status“* untergliedert. *„Status“* enthält die Information, die verarbeitet wird, während *„Component“* und *„TypeOfMachine“* Informationen über die Herkunft der Nachrichricht enthalten (siehe Abb. 50). Im Programmbaustein „Publish_Sub2_Conveyor1“ bzw. in den darin enthaltenen Aktionen erfolgt der Publish Prozess. Für jeden Sensor und Aktor der Ressource existiert eine „BuildPayload…“ Aktion. Wenn die einzelnen Aktionen vom übergeordneten Programm aufgerufen werden, werden durch die Aktionen die zu publishenden Informationen in das JSON-Format umgewandelt. Die Abb. 51. zeigt beispielhaft den Aufbau einer Aktion anhand der Lichtschranke „LB1_C1_Sub2“.

```
T1_LB1_C1_Sub2(IN := NOT T1_LB1_C1_Sub2.Q, PT := T#100MS);

asParameters_LB1_C1_Sub2[0] := GVL.Payload_Init.LB1_C1_Sub2.Component;
asParameters_LB1_C1_Sub2[1] := GVL.Payload_Init.LB1_C1_Sub2.TypeOfMachine;
asParameters_LB1_C1_Sub2[2] := BOOL_TO_STRING(NOT Modbus.SUBSYSTEM_2_tcp.dIn_5);

MyWriter_LB1_C1_Sub2(
    sJSON_BaseFrame:= sTemplate_LB1_C1_Sub2,
    oStatus=> myStatus_LB1_C1_Sub2,
    xError=> xWriter_Error_LB1_C1_Sub2,
    xDone=> xWriter_Done_LB1_C1_Sub2,
    aParameterValues:= asParameters_LB1_C1_Sub2,
    xTrigger:= xTrigger_LB1_C1_Sub2,
    sOutput := sPayload_LB1_C1_Sub2);
```

Abbildung 51: Aktion zur Bündelung der Informationen als JSON

Die Zeilen zwei bis vier bilden den Inhalt der zu bildenden JSON ab. Die Indexe [0] bis [2] geben die Reihenfolge an, die sich auf die sdtTag Vorlage mit den Namen *„Component“*, *„TypeOfMachine“* und „Status“ beziehen. *„Component“* und *„TypeOfMachine“* werden in einem anderen Baustein definiert und hier als globale Variable gelesen. *„Status“* ist jedoch mit dem physischen Input der Lichtschranke verbunden und repräsentiert somit den aktuellen Zustand. Vor dem Publish Prozess wird der Wert negiert und in einen String umgewandelt. Die nachfolgenden Zeilen sind vergleichbar mit anderen Funktionsbausteinen, wie beispielsweise TON, allerdings in der Form eines strukturierten Texts anstatt FUP. Der Funktionsbaustein implementiert die drei Informationen in ein JSON-Objekt. Während die Erstellung des JSON-Objekts für die einzelnen Sensoren und Aktoren separat in den Aktionen erfolgt, wird der Prozess des Publishens für alle Komponenten einer Ressource im selben Programm geregelt. Zusammenfassend wird der Programmabschnitt für jeden Aktor/Sensor in drei Abschnitte gegliedert. Der erste Abschnitt legt fest, wann bzw. wie oft die Nachricht gepublished werden soll. Die Wiederholrate wird hierbei für alle zu publishenden Nachrichten über eine globale Variable gesteuert. Im zweiten Teil wird die Aktion für den entsprechenden Sensor/Aktor aufgerufen und somit das JSON-Objekt erstellt. Im letzten Abschnitt werden dem JSON-Objekt die erforderlichen Informationen wie die Topic und der QoS beigefügt. Anschließend wird die Nachricht gepublished. Für die Aktivierung der einzelnen Skills der SPS müssen die Skill Calls empfangen werden.

Ähnlich wie für den Publish Prozess sind mehrere Komponenten für das Subscriben erforderlich. Dazu zählen globale Variablen, und ein Subscribe Programmbaustein je Ressource. Dieser Programmbaustein enthält jeweils eine Aktion. Für jeden Skill Call, jede Zeitvariable und Entscheidungsvariable wird eine globale Variable erstellt. Diese dienen als interne Verknüpfung zwischen der eingehenden MQTT-Nachricht und der Implementierung im SPS-Programm. Der Ablauf des Subscribe Prozesses wird am Beispiel des Conveyor1

beschrieben. Im Programmbaustein „Subscribe_Sub2_Conveyor2“ wird zunächst über eine globale Variable vorgegeben wie oft die eingehenden Nachrichten geparsed bzw. analysiert werden. Parsen beschreibt das Auslesen und Zuweisen der Informationen aus dem JSON-Objekt. Nachdem die Topic *(„OA“)* und der QoS *(„QoS1“)* definiert wurde, wird die Aktion „JSON_StringParse“ ausgeführt (siehe Abb.52).

```
T1(IN := NOT T1.Q, PT := GVL.Refresh_Subscribe); // Control how often message is parsed
IF T1.Q THEN


    xParserBuildup := TRUE;


END_IF

oFbSubscribe1
(xSubscribe := TriggerMethod, sTopic := 'OA', eQoS := eQualityOfService.QoS1, aPayloadData := aSubscriptionData1);

MemCopySecure(pDest:= ADR(sJSON), udiDestSize:= 255, pSource:= ADR(aSubscriptionData1), udiSourceSize:= oFbSubscribe1.dwRxNBytes, bPadding := );

JSON_StringParse();
```

Abbildung 52: Programmbaustein für das Subscriben von MQTT-Nachrichten

In der Aktion erfolgt die Umwandlung des JSON-Objekts zu einzelnen Strings, die auf die globalen Variablen geschrieben werden. Die Aktion besteht aus drei Abschnitten. Zunächst wird der Baustein für das Parsen des JSON-Objekts in Form des strukturierten Texts definiert. Nachfolgend wird über die Anzahl an *„xParserTrigger[#]“* die Anzahl der zu parsenden Informationen definiert. Im dritten Abschnitt wird für jede Information eine Parse-Methode erstellt (siehe Abb.53). Hierbei erfolgt über „sPointer“ eine Unterscheidung in untergegliederte Topics. Entsprechend der identifizierenden Topics wird die empfangene Information aus dem JSON-Objekt der gewünschten globalen Variable zugeordnet. Parallel wird definiert in welcher Form von Datentyp die globale Variable beschrieben werden soll. Das ist wichtig, da manche Funktionsbausteine nur bestimmte Datentypen als Input verwenden. Die globalen Variablen können an den entsprechenden Stellen im Programmbaustein „PRG_Sub2_C1undSM1“ implementiert werden, um z.B. Skills auszulösen oder Prozessparameter zu definieren.

```
JSON_Parser(
    pData:= ADR(sJSON),             // JSON string to parse
    udiSizeData:= length(sJSON),
    oStatus=> oParserStatus,
    xError=> xParserError,
    xDone=> xParserDone,
    diToken=> diParserToken,
    xTrigger:= xParserBuildup);     // Set xParserBuildup TRUE to execute the function block

IF xParserDone THEN
    xParserTrigger[0] := TRUE;      // Set xParserTrigger TRUE to exectue the parser method
    xParserTrigger[1] := TRUE;
    xParserTrigger[2] := TRUE;
    xParserTrigger[3] := TRUE;
END_IF

//2LBConveyor_forward
pJSON := JSON_Parser.GetValueByPath(sPointer := '/Time_2LBConveyor_forward_UDINT_Sub2_C1', xTrigger := xParserTrigger[0]);
GVL.Time_2LBConveyor_forward_UDINT_Sub2_C1 := STRING_TO_UDINT(pJSON.sValue);

pJSON := JSON_Parser.GetValueByPath(sPointer := '/Time_2LBConveyor_forward_time_Sub2_C1', xTrigger := xParserTrigger[1]);
GVL.Time_2LBConveyor_forward_time_Sub2_C1 := STRING_TO_TIME(pJSON.sValue);

pJSON := JSON_Parser.GetValueByPath(sPointer := '/SkillCall_2LBConveyor_forward_Sub2_C1', xTrigger := xParserTrigger[2]);
GVL.SkillCall_2LBConveyor_forward_Sub2_C1 := STRING_TO_BOOL(pJSON.sValue);

//2LBConveyor_backward
pJSON := JSON_Parser.GetValueByPath(sPointer := '/SkillCall_2LBConveyor_backward_Sub2_C1', xTrigger := xParserTrigger[3]);
GVL.SkillCall_2LBConveyor_backward_Sub2_C1 := STRING_TO_BOOL(pJSON.sValue);
```

Abbildung 53: Aktion zur Konvertierung und Zuordnung der JSON zu globalen Variablen

Die Implementierung von MQTT als Kommunikationsprotokoll zwischen der SPS und dem Demonstrator ermöglicht den zuverlässigen Austausch aller Informationen für die Prozesse. Der SPS-Client ist bezüglich der Durchlaufzeit beschränkt durch die Frequenz der Publish und Subscribe Prozesse. Die Refreshrate wird für alle Publish und Subscribe Programmbausteine über je zwei globale Variablen gesteuert. Durch die Umstellung zu einer ereignis-basierten statt einer zeitabhängigen Kommunikation, könnte die Durchlaufzeit verbessert werden.

Des Weiteren sollte ein Vergleich zwischen der Durchlaufzeit eines Prozesses bei Implementierung mit MQTT und mit OPC UA durchgeführt werden, um das optimale Kommunikationsprotokoll festzulegen bzw. um zu analysieren, wo die Vor- und Nachteile der Varianten liegen. Aufgrund von Komplikationen, während der Erstellung des Informationsmodells für die OPC UA Implementierung, konnte im Rahmen dieser Arbeit kein Vergleich zwischen den Kommunikationsprotokollen erstellt werden. Die Umsetzung von OPC UA kann nicht durchgeführt werden, da die Schaltfäche für den Upload des Informationsmodells im Web-based Management nicht angezeigt wird. Somit kann die PLC nicht auf das erstellte Informationsmodell zugreifen und der Server kann die Variablen nicht zur Manipulation verfügbar machen.

6. Fazit und Evaluation

Inhalt dieser Arbeit ist die Erstellung eines OAs in Node-RED, der fähig ist Skills in der Wago-SPS aufzurufen. Durch dieses Konzept soll eine Variante zur klassischen Schrittkette erstellt werden, die es ermöglicht die Steuerung des Systems aus der SPS auszulagern. Die erhofften Effekte sind eine höhere Modularität und Flexibilität als die konventionelle Version. Die praktische Umsetzung hat gezeigt, dass es über Node-RED möglich ist das Konzept umzusetzen und darüber die zusammenhangslosen Skills der SPS zu orchestrieren. Im Rahmen dieser Arbeit wurden die erforderlichen Skills für das Teilsystem-Wago erstellt, die Umsetzung der Ansteuerbarkeit beschränkt sich auf den Conveyor und die Stampingmachine im Subsystem 2. Das Design des SPS-Programms und des OAs ermöglichen die simple, modulare Erweiterung des restlichen Systems auf Basis des aktuellen Stands. So können, bis auf die Benennung der Variablen, die Funktionsbausteine für den Conveyor aus Subsystem 2 exakt gleich für den Conveyor aus dem Subsystem 3 verwendet werden.

Die drei Forschungsfragen aus dem Kapitel 1.2 können folgendermaßen beantwortet werden: Die Registrierung der Skills erfolgt in Node-RED über repräsentative Bausteine der Skills der SPS. Diese Bausteine können vom OA aufgerufen und mit Prozessparameter beschrieben werden. Die Skill-Bausteine werden über die Statusvariablen der SPS-Skills aktualisiert. Die Steuerungslogik wird vollständig aus der SPS ausgelagert. Das bedeutet, dass die Skills bei einem Aufruf ihre Funktion eigenständig ausführen können, allerdings sind die Skills im SPS-Programm untereinander nicht verknüpft. Die Betriebsarten sind ebenfalls vollständig über Node-RED implementiert. Daraus folgt eine absolute Trennung zwischen der ausführenden Ebene und der Steuerungsebene, da die SPS ohne Anbindung an Node-RED keine Aktoren aktivieren kann. Die Steuerung des OAs basiert auf der Verwendung eines Arrays, der die Skill Calls in der korrekten Reihenfolge enthält. Anhand dieses Arrays werden die repräsentativen Skill-Bausteine aufgerufen, die die Information an die reellen Skills in der SPS weitergeben. Kommunikationsschnittstellen sind bei dem System nur zwischen der SPS und Node-RED erforderlich. Es sind keine weiteren Schnittstellen erforderlich, da anstatt eines Knowledge Graphen das GUI als Vorgabe bzw. als Werkzeug für die Erstellung der Prozesse verwendet wird. Zudem ermöglicht das Node-RED-interne Dashboard die Visualisierung und Steuerung des Systems. Die Kommunikation zwischen SPS und Node-RED erfolgt über das Kommunikationsprotokoll MQTT.

Die Bewertung der Umsetzung wird in folgende Aspekte untergliedert: Die Umsetzung des OAs und der Betriebsarten in Node-RED, das SPS-Programm nach dem Ansatz des SkE und die Kommunikation über MQTT.

Die Umsetzung des OAs über Node-RED funktioniert. Die Vorteile der Nutzung von Node-RED sind, dass über vorgefertigte *nodes* die Einrichtung von MQTT simpel und schnell ist. Zudem ermöglicht das integrierte Dashboard durch die *ui nodes* eine einfache Steuerung der *flows* bzw. Visualisierung der Zustände. Die *flow-basierte* Funktionsweise kann allerdings für Probleme sorgen. Standardmäßig sind keine klassischen Logik-Gatter möglich, die beispielsweise die Meldung einer Statusveränderung registrieren können. Problematisch ist, dass die Funktionsweise von *function nodes* für größere Skripts eingeschränkt ist, da nach Start von beispielsweise *node-internen Loops* keine Änderungen von globalen Variablen registriert werden. Der OA ist so konzipiert, dass er über die einheitliche Bereitstellung der Informationen individuell zusammengestellte Prozesse und vordefinierte Services verarbeiten kann. Hierbei können die zu Grunde liegenden Skills in beliebiger Anzahl mit individuellen Parametern aufgerufen werden. Die Information über die Prozesse erfolgt aktuell allerdings nicht über einen Knowledge Graph, sondern über einen Array. Somit erfolgt in Node-RED die Orchestrierung und die Erstellung der Prozesse. Die Umsetzung der Betriebsarten über Node-RED zeigt, dass es möglich ist die Betriebsarten aus der SPS auszulagern. Während der Automatikbetrieb und der Initialzustand prozesssicher über Node-RED gesteuert werden kann, ist die Funktionalität des Notaus nicht ausreichend umgesetzt. Für einen reellen Anwendungsfall muss das menschliche Eingreifen alle Befehle des OAs aushebeln können, um die Prozesse sicher zu gestalten. Zudem muss der Notaus am physischen System installiert werden und darf keinen variablen Zykluszeiten unterliegen.

Das ausführende SPS-Programm ist nach dem Ansatz des SkE aufgebaut. Das bedeutet, die Skills werden baukastenartig erstellt und können individuell durch den OA angesteuert werden. Die Skills sind in der Form von composite skills angelegt und können entsprechend in verschiedenen Varianten ausgeführt bzw. durch unterschiedliche Bedingungen zurückgesetzt werden. Jede Ressource wird in dem Programm als eigener Programmbaustein instanziiert. Skills können dadurch einmal definiert werden und in verschiedenen Ressourcen verwendet werden. Des Weiteren bietet das Programm so die Möglichkeit der modularen Erweiterung von Ressourcen. Das Programm kann um ein Subsystem erweitert werden, während die bestehenden Komponenten nicht angepasst werden müssen.

Bei der Umsetzung der Kommunikation über MQTT ist zwischen den beiden Clients zu unterscheiden. Die Verbindung von Node-RED mit dem Broker kann innerhalb weniger

Minuten aufgebaut und getestet werden. Die standardmäßigen *MQTT nodes* vereinfachen diesen Prozess. Der Verbindungsaufbau zwischen der SPS und dem Broker erweist sich als aufwändiger. Dafür muss für jede Ressource ein Programm für das Publishen und ein Programm für das Subscriben erstellt werden. Die Durchlaufzeit des Systems kann verbessert werden, indem das Subscriben und Publishen nicht in einer definierten zeitlichen Rate, sondern Event-basiert erfolgt. Das bedeutet, dass Nachrichten beispielsweise nur gepublished werden, wenn sich die zugrunde liegenden Informationen ändern. Dadurch kann die Menge der übertragenden Daten und die Zeit zwischen Änderung der Information und Versenden der Nachricht minimiert werden. Zudem wird ein Vergleich der Durchlaufzeiten eines MQTT-basierten und OPC UA-basierten Systems empfohlen.

7. Ausblick

Neben der Erweiterung des Systems um die OPC UA Funktionalität ist die standardmäßige Definition der Prozesse ein Verbesserungsaspekt. Die Prozesse sollten nicht über den OA erstellt werden, wie es aktuell der Fall ist, sondern über einen Knowledge-Graph bereitgestellt werden. Der Knowledge-Graph dient der Festlegung der Reihenfolge der Skills und ordnet diesen die korrekten Prozessparameter zu. Im Rahmen der Hardwareunabhängigkeit und Interoperabilität ist die Einbindung von BPMN-Prozessen denkbar. Die Beschreibung von Prozessen und Abläufen über BPMN ist eine etablierte Praktik. Über eine automatische Umwandlung von BPMN-Modellen in einen Knowledge-Graph können Prozesse ohne Zwischenschritte aus der Konzeption in das Produktionssystem aufgenommen werden.

Ein weiterer Schritt ist die Einbindung des OAs in ein MAS. Hierbei wird ein BDI-Agent in den OA integriert. In einem MAS, in dem jede Ressource über einen OA verfügt, können durch die Kommunikation zwischen den Agenten Prozesse im gesamten Produktionssystem orchestriert werden. Die Agenten sind in der Lage untereinander zu kommunizieren, welcher Agent über welche Ressourcen verfügt. Durch Abgleich des Produktionsauftrags und den freien bzw. verfügbaren Ressourcen kann autonom der Prozess gebildet werden. Wenn bestimmte Ressourcen nicht verfügbar sind, ist das MAS fähig denselben Prozess über alternative, geeignete und verfügbare Ressourcen zu modellieren. In diesem Szenario ist es möglich in einem Produktionssystem Produkte mit verschiedenen Spezifikationen gleichzeitig zu produzieren.

Das Statusmodell des OAs besteht aktuell aus zwei Zuständen, „on“ und „off“. Hinsichtlich der Steuerung des Normalbetriebs sind die Zustände ausreichend. In der Realität ist der Anteil des Normalbetriebs kleiner und wird von einer Vielzahl an verschiedenen Fehlerzuständen unterbrochen. Dementsprechend ist es sinnvoll das System mit einem umfangreichen Statusmodell zu erweitern, um den OA zu befähigen, präziser auf reale Zustände des Systems zu reagieren.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

Blanco, Phil/Kotermanski, Rick/Merson, Paulo (2007): Evaluating a Service-Oriented Architecture. Fort Belvoir, VA: Defense Technical Information Center.

Bommer, Benjamin: Konzept einer shared production für eine Festo MPS Station mittels eines Agentensystems.

Bundesministerium für Wirtschaft und Klimaschutz (2023): Was ist Industrie 4.0? URL: https://www.plattform-i40.de/IP/Navigation/DE/Industrie40/WasIndustrie40/was-ist-industrie-40.html (20.07.2023).

Clerissi, Diego et al. (2018): Towards an approach for developing and testing Node-RED IoT systems. In: Proceedings of the 1st ACM SIGSOFT International Workshop on Ensemble-Based Software Engineering. New York, NY, USA. New York, NY, USA: ACM.

Dorofeev, Kirill (2020): Skill-based engineering in industrial automation domain. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings. New York, NY, USA. New York, NY, USA: ACM.

González García, Cristian et al. (2017): A review about Smart Objects, Sensors, and Actuators. In: International Journal of Interactive Multimedia and Artificial Intelligence, 4. Jg. (3), S. 7. URL: https://www.researchgate.net/profile/cristian-gonzalez-garcia/publication/307638707_a_review_about_smart_objects_sensors_and_actuators.

HiveMQ (2015): What is MQTT Quality of Service (QoS) 0,1, & 2? URL: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

Jennings, Nicholas R. (2000): On agent-based software engineering. In: Artificial Intelligence, 117. Jg. (2), S. 277–296. URL: https://www.sciencedirect.com/science/article/pii/s0004370299001071.

Jungbluth, Simon et al. (2023): Developing a skill-based flexible transport system using OPC UA. In: at - Automatisierungstechnik, 71. Jg. (2), S. 163–175.

Küveler, Gerd/Schwoch, Dietrich (2003): Das ISO/OSI — Schichtenmodell der Datenkommunikation. In: Informatik für Ingenieure: Vieweg+Teubner Verlag, Wiesbaden, S. 446–451. URL: https://link.springer.com/chapter/10.1007/978-3-663-01369-3_22.

Lasi, Heiner et al. (2014): Industrie 4.0. In: WIRTSCHAFTSINFORMATIK, 56. Jg. (4), S. 261–264. URL: https://link.springer.com/article/10.1007/s11576-014-0424-4.

Laskey, Kathryn B./Laskey, Kenneth (2009): Service oriented architecture. In: Wiley Interdisciplinary Reviews: Computational Statistics, 1. Jg. (1), S. 101–105.

Lober, Andreas et al.: Flexible Skill-based Production Systems through novel OPC UA Design Approaches.

Madakam, Somayya/Ramaswamy, R./Tripathi, Siddharth (2015): Internet of Things (IoT): A Literature Review. In: Journal of Computer and Communications, 03. Jg. (05), S. 164–173. URL: https://www.scirp.org/html/56616_56616.htm.

Mishra, Biswajeeban/Kertesz, Attila (2020): The Use of MQTT in M2M and IoT Systems: A Survey. In: IEEE Access, 8. Jg., S. 201071–201086.

Ooms, Jeroen (2014): The jsonlite Package: A Practical and Consistent Mapping Between JSON Data and R Objects.

OPC Foundation: Unified Architecture. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.

OpenJS Foundation & Contributors: Node-RED. URL: https://nodered.org/about/.

Pisarić, Milan et al. (2020): Towards a Non-disruptive System for Dynamic Orchestration of the Shop Floor. In:. IFIP International Conference on Advances in Production Management Systems: Springer, Cham, S. 469–476. URL: https://link.springer.com/chapter/10.1007/978-3-030-57997-5_54.

Suleiman, Zhanybek et al. (2022): Industry 4.0: Clustering of concepts and characteristics. In: Cogent Engineering, 9. Jg. (1).

(2015): The internet of things: An overview.

Viroli, Mirko/Denti, Enrico/Ricci, Alessandro (2007): Engineering a BPEL orchestration engine as a multi-agent system. In: Science of Computer Programming, 66. Jg. (3), S. 226–245. URL: https://www.sciencedirect.com/science/article/pii/s0167642307000378.

Wooldridge, Michael: An Introduction to MultiAgent Systems.

Wooldridge, Michael/Ciancarini, Paolo (2001): Agent-Oriented Software Engineering: The State of the Art. In:. International Workshop on Agent-Oriented Software Engineering: Springer, Berlin, Heidelberg, S. 1–28. URL: https://link.springer.com/chapter/10.1007/3-540-44564-1_1.

Zimmermann, Patrick et al.: Skill-based Engineering and Control on Field-Device-Level with OPC UA.

Anhang

Anhang 1:

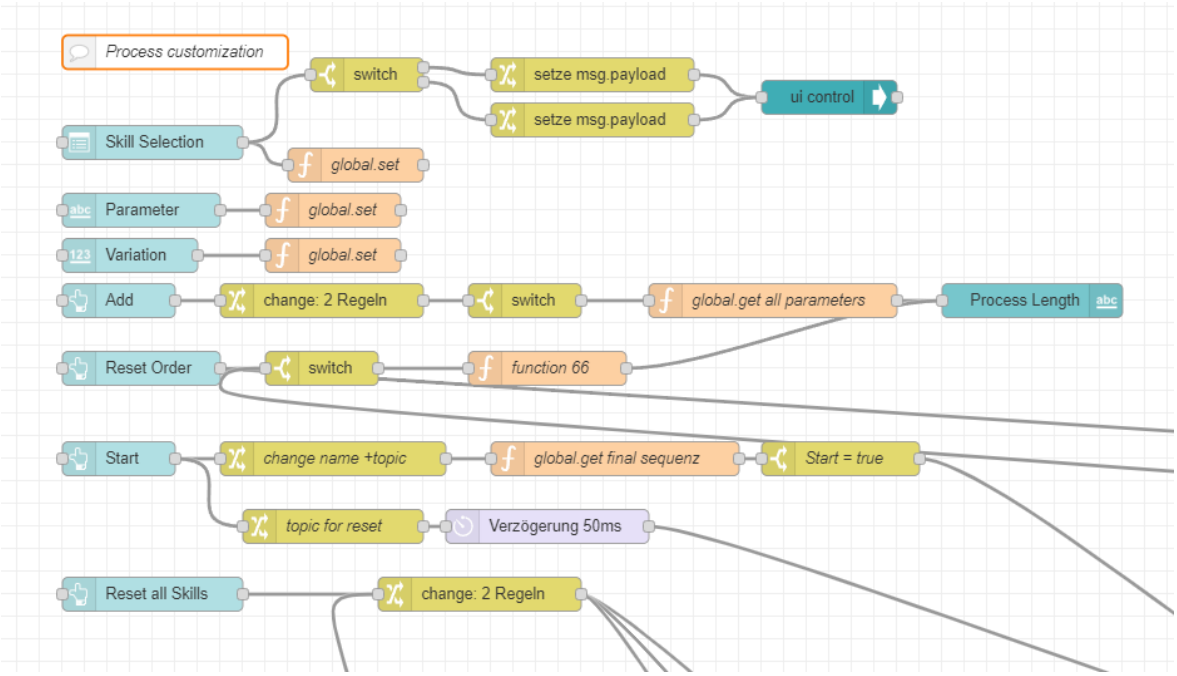
Übersichttabelle über die Sensoren und Aktoren des Demonstrators (Teilsystem Wago)

Subsystem	Modul	Kategorie	Typ	Name	Zustand
2	Conveyor1	Sensor	Light barrier	Light barrier 1	i.O.
2	Conveyor1	Sensor	Light barrier	Light barrier 2	i.O.
2	Puncher1	Sensor	Push button	Button Up	i.O.
2	Puncher1	Sensor	Push button	Button Down	i.O.
2	Conveyor1	Actuator	Motor	Motor	i.O.
2	Puncher1	Actuator	Motor	Motor	i.O.
2	Crane1	Sensor	Push button	Button Rotation	i.O.
2	Crane1	Sensor	Push button	Button Back	i.O.
2	Crane1	Sensor	Push button	Button Up	i.O.
2	Crane1	Actuator	Motor	Motor Rotation	i.O.
2	Crane1	Actuator	Motor	Motor horizontal	i.O.
2	Crane1	Actuator	Motor	Motor vertical	i.O.
2	Crane1	Actuator	Compressor	Compressor	i.O.
2	Crane1	Actuator	Magnetvalve	Vacuum valve	i.O.
3	Conveyor2	Sensor	Light barrier	Light barrier 1	i.O.
3	Conveyor2	Sensor	Light barrier	Light barrier 2	i.O.
3	Puncher2	Sensor	Push button	Button Up	i.O.
3	Puncher2	Sensor	Push button	Button Down	i.O.
3	Conveyor2	Actuator	Motor	Motor	i.O.
3	Puncher2	Actuator	Motor	Motor	i.O.
3	Crane2	Sensor	Push button	Button Rotation	i.O.
3	Crane2	Sensor	Push button	Button Back	i.O.
3	Crane2	Sensor	Push button	Button Up	i.O.
3	Crane2	Actuator	Motor	Motor Rotation	i.O.
3	Crane2	Actuator	Motor	Motor horizontal	i.O.
3	Crane2	Actuator	Motor	Motor vertical	i.O.
3	Crane2	Actuator	Compressor	Compressor	i.O.
3	Crane2	Actuator	Magnetvalve	Vacuum valve	i.O.
1	Pusher1	Actuator	Motor	Motor Pusher	i.O.
1	Pusher2	Actuator	Motor	Motor Pusher	i.O.
1	Conveyor1	Actuator	Motor	Motor	i.O.
1	Conveyor2	Actuator	Motor	Motor	i.O.
1	Conveyor3	Actuator	Motor	Motor	i.O.
1	Conveyor4	Actuator	Motor	Motor	i.O.
1	Conveyor1	Sensor	Light barrier	Light barrier 1	i.O.
1	Conveyor1	Sensor	Light barrier	Light barrier 2	i.O.
1	Conveyor2	Sensor	Light barrier	Light barrier	i.O.
1	Conveyor3	Sensor	Light barrier	Light barrier	i.O.

1	Conveyor4	Sensor	Light barrier	Light barrier	i.O.
1	Pusher1	Sensor	Push button	Button Out	i.O.
1	Pusher1	Sensor	Push button	Button In	i.O.
1	Pusher2	Sensor	Push button	Button Out	i.O.
1	Pusher2	Sensor	Push button	Button In	i.O.

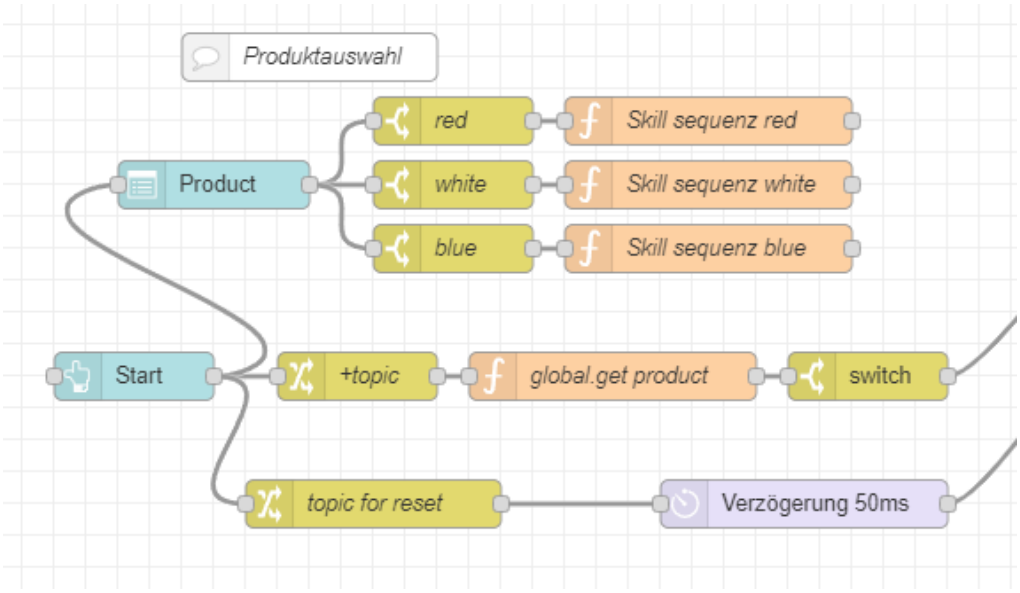
Anhang 2:

Node-RED Abschnitt für die Erstellung individueller Abläufe



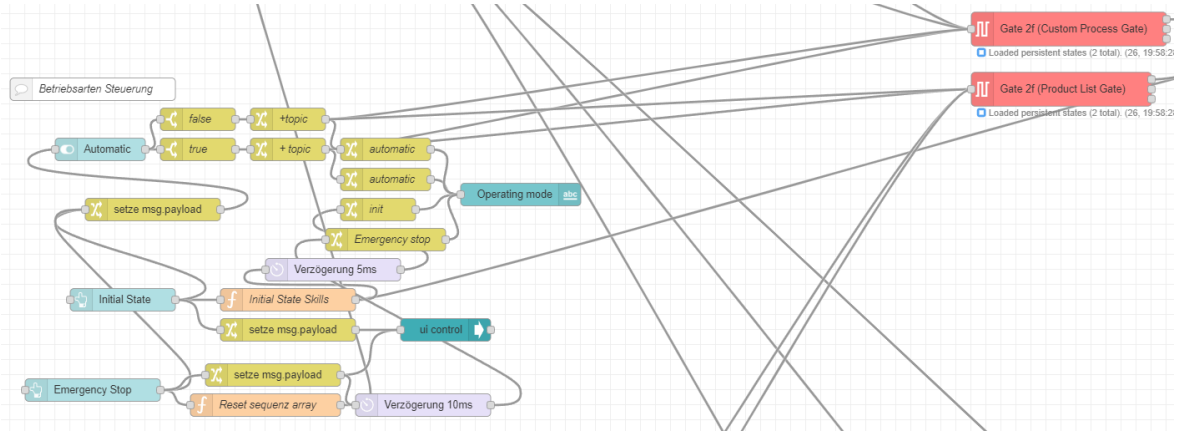
Anhang 3:

Node-RED Abschnitt für die Produktauswahl



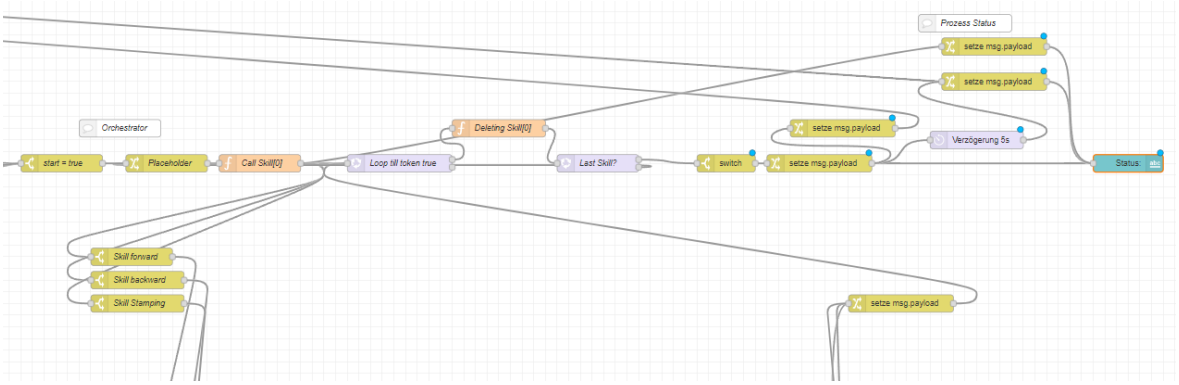
Anhang 4:

Node-RED Abschnitt für die Betriebsarten



Anhang 5:

Node-RED Abschnitt für den Orchestrator



Anhang 6:

Node-RED Abschnitt für die Skill-Bausteine