Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Deparment of Telecommunications and Media Informatics

Dynamic Industrial Workflow Execution Supported by Service Discovery

Conference of BME Scientific Students' Association

Authors:

Tamás Mrázik
Kristóf Szabó
Bence Tóth

Supervisor:

Dr. Pál Varga

2020

Contents

Kivonat

Az ipari automatizálás, azon belül a dinamikusan változó munkafolyamat (workflow) automatikus, okos, valamint okos eszközökkel segített végrehajtása iránti igény növekedésével kapcsolatban a megoldási lehetőségek még korlátozottak. A jövő gyárainak a jelenlegi hatékonyság és megbízhatóság mellett a gyártási folyamatok flexibilis igényeire (lot-size-1 termékek) és a gyártási környezet gyors változásaira is tudni kell reagálnia. Ezt a munkafolyamat leírásának dinamikus végrehajtásával, az erőforrások rendelkezésre állásának figyelembevételével, és a környezet megváltozására történő megfelelő reakciókkal kell lekezelni. A munkadarabok megfelelő kezelését többek között gépi látást támogató technológiákkal, a használható eszközök körének változásához való alkalmazkodást pedig szolgáltatás-alapú architektúrák (Service Oriented Architecture, SOA) alkalmazásával érhetjük el. Dolgozatunk épp ezekre mutat egy keretrendszert és egy konkrétan működő, integrált példát.

Ahhoz, hogy kidolgozzunk egy lehetséges megoldást erre az új kihívásra, igénybe vettük az Arrowhead keretrendszer által nyújtott szolgáltatásokat, valamint továbbfejlesztettük a Workflow Choreographer nevű kiegészítő rendszerét is. Ezek mellett használtunk még mélytanuláson alapuló gépi látást, robotkarokat és egy saját fejlesztésű „okos" futószalagot az összetett gyártósor modellezésére.

Mint említettük, a gépi látást mélytanulással valósítottuk meg. Ehhez a TensorFlow-t vettük igénybe, amely egy általános, mesterséges intelligenciát és gépi- illetve mélytanulási folyamatokat támogató, teljességében nyílt forráskódú szoftvercsomag.

Az Arrowhead egy ipari automatizálást támogató keretrendszer, mely szolgáltatásorientált módon közelíti meg a problémát. Az Arrowheadben jelen lévő elemek önálló rendszert alkotnak; képesek lehetnek szolgáltatásokat használni és nyújtani, míg az Arrowhead - úgynevezett „rendszerek rendszere" (System of Systems, SoS) - szervezi ezen elemek munkáját, ütemezi az erőforrások használatát. Ezen felül a keretrendszer kihasználja a lokális, valamint a távoli felhőkben rejlő potenciált. A lokális felhőkön belül többek között garantálható a megfelelő válaszidő és az Arrowheadnek köszönhetően nem csak az egyes felhőkön belül, hanem közöttük is folytatható biztonságos kommunikáció.

Az Arrowhead keretrendszer Workflow Choreographer munkafolyamat-végrehajtást támogató modulja képes egy komplex gyártási folyamat dinamikus felügyeletére. Ezzel az eszközzel bebizonyítható a rendszerek rendszere megközelítés helyessége különféle gyártási végpontok használatlával. Az esettanulmányban szereplő robotkarok, valamint gyártássegítő eszközök (futószalag, kamera) mind a Workflow Choreographer segítségével kapják meg a dinamikusan változó munkakiosztást annak függvényében, hogy mely eszközök áll-

nak az adott időpillanatban rendelkezésre a feladat végrehajtásához. A rendszerek rendszere elven alapuló okosgyárak létjogosultságát és a bennük rejlő potenciált hivatott ezen dolgozat és a komplex demonstráció bemutatni.

Abstract

As the demand for industrial automation grows, including the automatic and smart implementations of a dynamically changing workflow, the availability of solutions are still limited. In addition to the current efficiency and reliability, the smart factories of the future must be able to respond to the flexible needs of production processes (lot-size-1 products) and also to the rapid changes in the production environment. This should be addressed through dynamic execution of the workflow description with the consideration of resource availability, and appropriate responses to changes in the work environment. To interact with new work pieces, technologies should be used that support machine vision and can manage the various range of tools present in the work environment, such as the Service Oriented Architecture (SOA) approach. Our dissertation presents a framework and a concrete, integrated example of these.

To develop a possible solution to this new challenge, we used the services provided by the Arrowhead Framework, as well as developed further a supporting system called the Workflow Choreographer. In addition to these, we also used machine vision, robotic arms and a self-developed "smart" conveyor belt to model a complex production line.

As mentioned above, machine vision was implemented using deep learning algorithms and neural networks. For this, we used TensorFlow, a fully open source software package that supports general artificial intelligence and deep learning processes.

Arrowhead is a framework that supports industrial automation approaching the problem in a service-oriented way. The elements present in Arrowhead form a self-contained system in which devices may be able to use and provide services, while Arrowhead - the so-called "System of Systems" (SoS) - organizes the work of these elements, scheduling the use of resources. In addition, the framework takes advantage of the potential of local as well as remote clouds. Within local clouds, among other things, the required response time can be guaranteed, and, thanks to Arrowhead, secure communication can take place not only within individual clouds, but also between them.

The Workflow Choreographer supporting system in the Arrowhead framework is capable of dynamically monitoring a complex manufacturing workflow execution process. With this tool, the correctness of the SoS approach can be demonstrated with using different manufacturing endpoints. In our case study we used robotic arms and other production aids (conveyor belt, camera) which use the dynamically changing task allocation provided

by the Workflow Choreographer, depending on which tools are available at the moment of task execution. This dissertation and the complex demonstration are intended to present the relevance and potential of smart factories based on the System of Systems principle.

Chapter 1

Introduction

In the past few years, as a result of technological advancements the consumer society has changed. In order to satisfy the new needs of the accelerated lifestyle – the increased consumption or environmentally friendly approaches – developing new eco-conscious industrial solutions that maximize productivity are essential.

Productivity today depends on many factors, such as the number of working employees, machines, the physical capacity of these machines, the supply chain and so on. Nearly all of these depend on tasks that are physically executed by people. Using technologies available today, industrial automation is a possible solution to increase productivity by utilizing intelligent systems capable of self organisation. These also can partly or fully replace human resources in future industrial environments.

Achieving industrial automation however is not as simple as it might sound. While modern machines are mostly programmable to some extent, it is inevitable to provide solutions to machines already used for decades, still serving their roles. Digital maturity of factories is diverse, and as of now there is a wide technological gap between Industry 2.0 and Industry 4.0 which means it is a great challenge to provide solutions for factories all around the globe. In both cases the goal is the same: improve productivity while paying attention to environment friendly principles.

In our paper first we will introduce workflow execution principles that are available today in Chapter 2 as well as image processing possibilities to help asset tracking in Chapter 3. Then Chapter 4 presents platform and application system support to achieve the goals described above. In Chapter 5 one solution to image processing is elaborated and in Chapters 6 and 7 we showcase a small size demonstrations emulating an Industry 4.0 production line, integrated into the further developed Arrowhead Framework. Finally in Chapter 8 we demonstrate some measurements in the systems to prove the adequacy of our concept.

In the working demonstration of the complex use case, automated dynamic workflow management is achieved through the utilization of the new Supporting Systems of the Arrowhead Framework – namely the redesigned Workflow Choreographer system and the new Workflow Executor system –, and the integration of every component of the Cyber-Physical System of Systems (CPSoS) into the Arrowhead Framework. As for the components of the demonstration, we used a custom conveyor belt and multiple robotic arms, supported by a computer vision system.

Our contributions to the state-of-the art are the following:

- The creation and validation of the new Workflow Choreography architecture, including the definition, creation and usage of Workflow Executors.
- The Workflow Choreography and Execution concept has been validated through a complex use case scenario, which includes a conveyor belt, a camera for deep learning-based image processing and several robotic arms controlled in sync with the camera-provided images.
- The use case demonstrates both the flexibility of the Arrowhead framework through Orchestrating various service producers and the dynamic execution of a production recipe by using the available resources at the production floor.
- The definition and extensive usage of the BPMN-based Recipe Description Language that fits into the Arrowhead systems communication structure with its JSON-based information representation.
- The demonstration on how an Engineering Toolchain with Device catalogues (Eclipse Vorto), SysML-based System of Systems representation (MagicDraw) and the Arrowhead Management Tool for deployment can ease systems engineering complex use-cases. As part of this, a plugin was developed for the MagicDraw visual modeling tool to make the creation of Choreographer compatible production recipes visual, more intuitive, hence easier.

Chapter 2

Related work

2.1 Workflow execution in manufacturing

Introducing IoT support for production facilities is often referred to as "digitalisation", and it is currently a highly challenging task in the industry. A system that manages the daily tasks in a factory has to be flexible and secure in its communication, and it has to be robust, scalable as well. This is especially true when communication endpoints use the Internet, more specifically the HTTP (or HTTPS) protocol to communicate with each other.

In related research several modeling tools are presented for workflow and business process modeling alternatives such as the BPMN 2.0, Event-driven Process Chains (EPCs) and the Unified Modeling Language (UML). It became clear that for our case of IoT applications using the BPMN 2.0 ISO standardized notation to make business processes executable is the best solution [1] by using process engines (e.g., Activiti, Bonita BPM, or jBPM). BPMN is also the most widely used industrial modeling language therefore the perfect candidate to bridge the communication gap often occurring between business process planning and implementation.

As it can be seen there are various concepts to support digital production – we propose to use the Arrowhead framework, one of the quasi-standards for creating cyber-physical system-of-systems in the industrial IoT domain. The main motivations for the Arrowhead framework include its design for real-time support, high security, flexibility, integrability with legacy systems, and that interopearbility is achieved by its Service Oriented Architectural approach. A comparative study of the Arrowhead framework and other industrial IoT platforms can be found in [2].

To achieve the desired control over communication between the various endpoints of the assembly lines we used three mandatory core systems of the Arrowhead Framework: the Orchestrator, the Service Registry and the Authorization. Further information about these core systems in general will be provided in the next section. Moreover, there is one

supporting core system of the framework which is essential in the execution of each task included in a workflow, namely the Workflow Choreographer.

The core systems supplemented by the Workflow Choreographer create a powerful tool for flexible and dynamic production control.

When compared to other solutions, the distinguishing feature of this approach is that the production recipe is executed depending on the providers available for the given service at a given step - rather than the resources being strictly associated with the tasks. The Choreographer has the recipe and controls its status, whereas the Orchestrator creates the matchmaking of service consumers and producers at each step. The details of this process and functions are provided in the coming sections.

2.2 The Arrowhead Framework

The Arrowhead Framework [3] as mentioned above supports the principles of the System Oriented Architecture (SOA) approach encapsulating geographically local automation clouds into Local Clouds as seen on Figure 2.1. It conforms to the producer-consumer pattern meaning that it assists establishing and maintaining connection between systems that produce services (providers) and systems that consume these services (consumers) forming System of Systems (SoS).

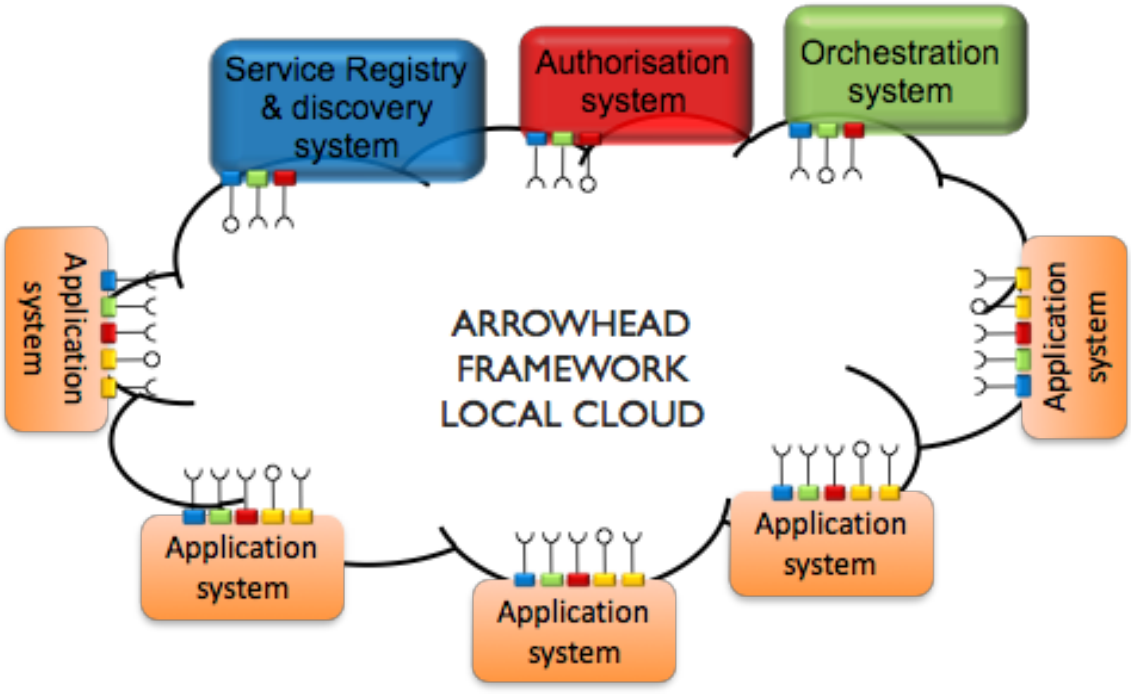

Figure 2.1: An Arrowhead Local Cloud with the Core Systems and Application Systems [4].

A simplistic example for a consumer system can be seen on Figure 2.2. Here System B consumes the service provided by System A. The advantages of using the SOA approach for interoprability purposes is its Late binding, Loose Coupling and Lookup features.

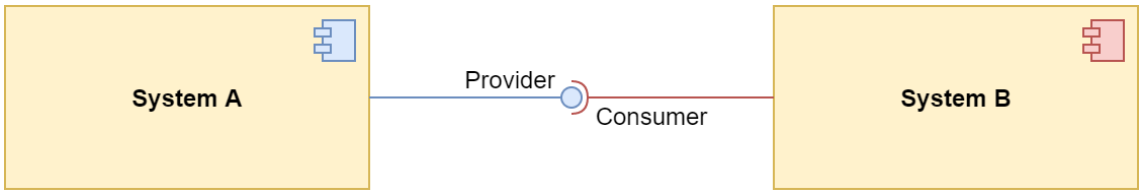

Figure 2.2: System B consuming service provided by System A.

2.2.1 The Service Registry core system

The Service Registry is one of the mandatory core systems in the Arrowhead Framework. There are two important services that the Service Registry provides for others: Service Publishing and Service Discovery (SD, lookup).

Through Service Publishing systems can register their services and publish them across a local or remote cloud enabling other systems to access them with SD. This way available services in the cloud can change dynamically upon registration and unregistration processes.

Service Discovery makes it possible for all systems in the cloud to request services that they need during execution. This request is not sent directly to the Service Registry but through the Orchestrator core system which, if the requested service is available in the cloud provides all the necessary information for consumers to establish connection with the corresponding providers.

Both of these services and the connection between the application systems and the core systems can be seen on Figure 2.3.

2.2.2 The Orchestrator core system

The Orchestrator as seen on Figure 2.3 communicates with almost every other core system and application system. If a consumer needs a service from a provider system, a request is sent to the Orchestrator to find such provider. Then the Orchestrator can use dynamic lookup or the Orchestration Store based on the request the consumer sent to find a match for it. The Orchestration Store contains records of matchmakings between providers and consumers defined by system operators before the beginning of the orchestration process. Previous to sending back data to the consumer about possible matches the Orchestrator also checks with the Authorization core system that the provider candidates are authorized to communicate with the requester system.

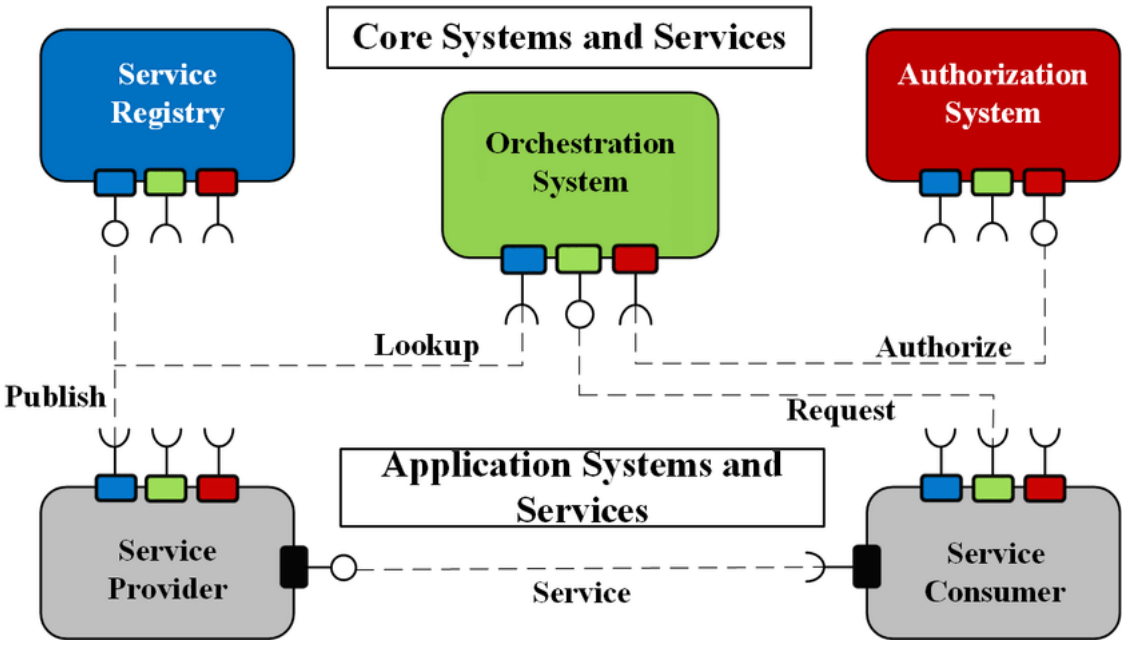

Figure 2.3: Arrowhead Core Systems and their services [5].

2.2.3 The Authorization core system

As mentioned above the Authorization core system is mainly responsible for defending against unauthorized access by using certificates, authorization tokens and other authentication methods. A consumer can only get the detailed information of the provider from the Orchestrator if it has the appropriate token or certificate to communicate with it. Therefore the Authorization system is essential for secure communication in local and between remote clouds.

2.3 The Principles of Workflow Choreography in Arrowhead

In order to manage and execute workflow recipes in production and logistics, a new supporting core system was defined within Arrowhead. There are a lot of requirements and restraints the Workflow Choreographer faces upon the management and execution of workflows, such as constraints of workstations that are used, the sequencing (sometimes parallelisms) of various tasks and services, and further aspects of production plants. It was first introduced as an engine that controls automated production and in doing so was planned to greatly reduce the difficulty of management challenges in a production facility.

The features and services that the Workflow Choreographer provides can be seen on Figure 2.4. First it processes the Production Order (PO) acquired from various Enterprise Resource Planning (ERP) systems and creates the Production Recipe (PR) based on this data. Furthermore, it manages the different Workflow Executors (WE) to accomplish the tasks at hand either by instantiating them or sending messages to already instantiated

WEs. After getting information about the production steps in a recipe the WEs execute the related activities by coordinating the workstations to perform the modifications described in the PR. From the Arrowhead Framework's point of view the Choreographer pushes requests to the Orchestrator and subscribes to events reported by other application systems [6].

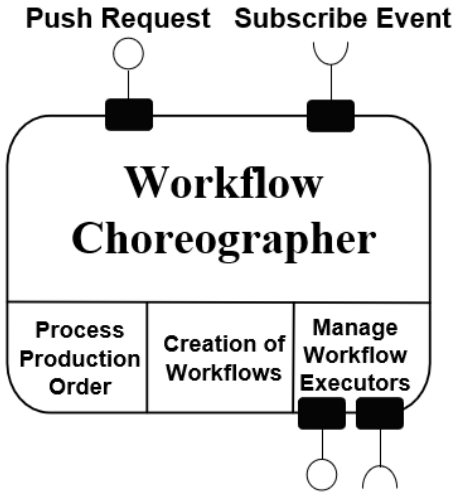

Figure 2.4: The services and features of the Choreographer [6]

The Workflow Choreographer needs to store information about every workflow running in the system to ensure their proper execution by monitoring, detecting and handling possible errors occurring during production. To achieve this, two core systems support the workflow Choreographer, namely the Orchestrator and the Event Handler (EH). The former enables the Choreographer to pair application systems (service consumers and providers) together while the latter delivers messages across the system if something goes wrong or a workstation finished task execution successfully.

Figure 2.5 shows how the Choreographer configures the Orchestrator before pairing the two application systems for starting task execution. Then the WE takes control and manages the application systems to achieve the behaviour described by the production recipe. This is followed by a callback to the Choreographer with the result (either if the task is done or some unexpected error happened that needs handling) [7].

2.4 Research gaps

It became clear during the study of the related work presented earlier that there are multiple ambiguous aspects of the behaviour of the Workflow Choreographer in practice.

First of all, a powerful description language was needed to enable handling production recipes both by the Choreographer itself and by possible humans involved in production. The chosen language follows the JSON (JavaScript Object Notation) standard because it is easily readable, supports the sufficient complexity and is also used by other core

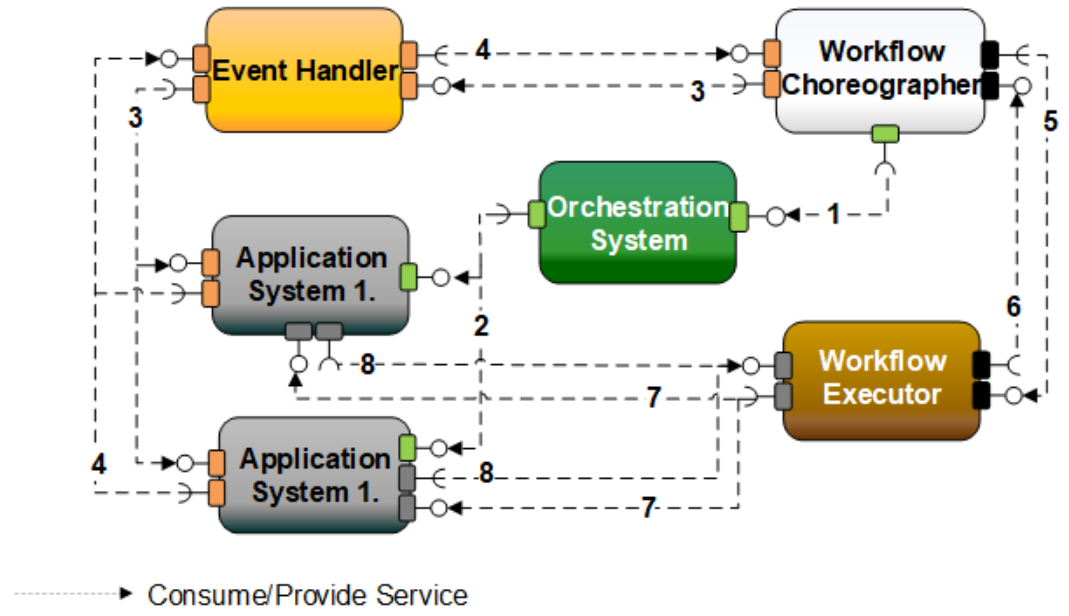

Figure 2.5: The environment of the Workflow Choreographer in Arrowhead [7]

systems in the AH Framework as a Data Description Language (DDL) for lightweight data-interchange.

It was also not obvious where the previously mentioned Workflow Executors stand in the framework: on the side of the core systems or on the same level as the application systems. Both design choices have their advantages and disadvantages but ultimately only one clarified solution should be supported by the Arrowhead Framework to avoid ambiguity.

Besides both of these complex problems that need a lot of testing and well thought through designs and implementations, the storage and forwarding of data between application systems also emerged as an issue. It became important to address the possible solutions for continuous data-flow during the execution of production recipes.

Chapter 3

A Brief Overview of Computer Vision-related Technologies

3.1 Image processing and Machine Learning

This chapter provides a general survey of image processing and machine learning techniques, methods and functions that are used (or needed as a background) for the complex industrial-motivated use-case presented in this document.

3.2 The role of cameras in the industry

The role of camera systems in an industrial environment has a quite wide range. They have been serving safety, security purposes for a long time, moreover with the utilization of the image processing algorithms of today, they can provide information for processes being on a higher level of abstraction. Industrial camera systems are capable of supporting high precision tasks, and quality management can be partially - if not totally - automatized with the proper usage of algorithms based on the data produced by such systems.

3.3 A brief introduction to image processing algorithms

In the early days of image processing, due to the computational and technical limitations, most of the image processing operations were only meant to assist human operators in their work. For example, applying an edge detector filter to an image or different color related transformations helped humans to see better on the recorded images, but the decision making was still the job of the person. These transformations were calculated using the central processor unit (CPU), but the complexity of the later algorithms required a higher computational capability and efficiency than CPUs could provide. Nowadays, most of the state of the art computer vision technologies and systems use graphics processing

unit (GPU) accelerated algorithms as GPUs have specific physical architecture and are capable of performing vector and matrix operations much faster than CPUs. CPUs are not designed to perform such operations, hence, for example, calculating a matrix multiplication takes just as many operations as it would take for a human to calculate it by hand, only that the CPUs are much faster at calculating the basic operations required to perform such operations. With this extra functionality of GPUs, mathematical models and methodologies created for image processing became computationally more efficient and more applicable in real-time scenarios.

3.4 Artificial Neural Networks in general

Artificial Neural Networks (ANNs), as the name suggests, are trying to artificially - digitally replicate the behaviour of a non-artificial neural network - the brain. As such, ANNs are constructed of artificial neurons layered, stacked and connected to create a network capable of performing different tasks. For different tasks, usually a differently constructed network is required or is more optimal; an expert is able to build a custom neural network for a specific task. The process of learning in ANNs is based on forward and back propagation methods, including gradient descent based on cost/loss functions, adjusting the weights assigned to artificial neurons, and many other concepts which will be described in the next sections and paragraphs.

3.4.1 Neurons

Artificial neurons are replicating the behaviour of biological neurons. Hence, some basic knowledge of biological neurons is helpful in understanding basic concepts.

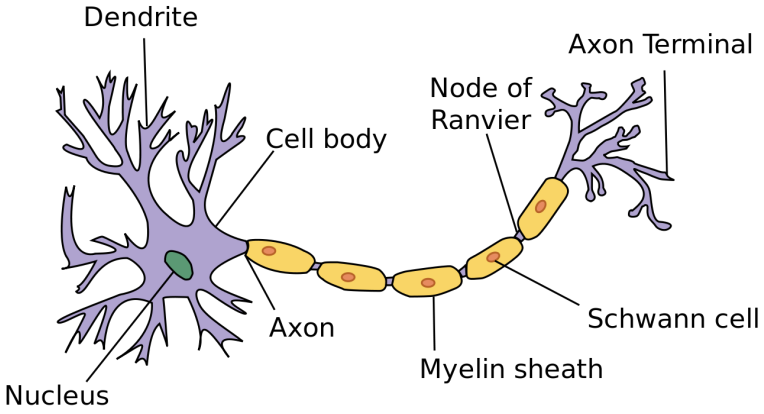

Figure 3.1: The figure of a biological neuron.
Dendrites and the Axon terminals are the major points of similarity between artificial and biological neurons [8]

As shown on Figure 3.1, biological neurons have Dendrites and Axon terminals, and these can be looked at as the inputs and outputs of a biological neuron. Basically biological

neurons work with very low voltages, and are connected with each other, so in general, a neuron has other neurons as inputs and other neurons as outputs. When the input neurons transfer signals to the neuron, the signals accumulate, and cause an action potential if they reach a threshold, which is basically a binary reaction, as it can only happen or not happen. It is also worth noting that as action potentials are a binary reaction, there is no meaning of different voltages, it either happened or did not. This depends on complex electrical and chemical reactions, however, a simple mathematical model created to represent artificial neurons have a function called activation function which determines the output of the artificial neuron.

3.4.2 Activation functions

Step function

The step function is one of the first activation functions as it represents the binary decision quite intuitively. The codomain of the function is Y = 0, 1, which shows the handiness of the function as biological neurons also provide binary output, this can be seen on Figure 3.2. If the accumulated input values reach a certain threshold it gives the output 1 - True -, and 0 - False - otherwise. For binary classification this is useful, but it can not produce n-ary output, hence is inappropriate for multi-class classification. Also, due to the mathematical operations performed on the activation functions, it is advised to have differentiable activation functions having non-zero derivatives.

$$\phi(x) = 1, if x \geq 0$$

$$\phi(x) = 0, if x < 0$$

$$\phi(x) \longrightarrow [0; 1]$$

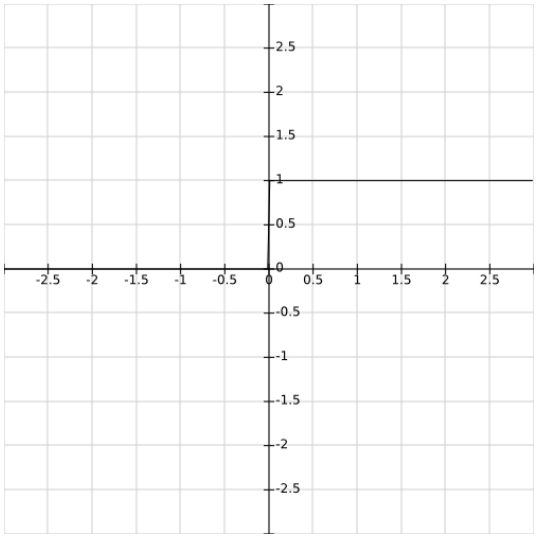

Figure 3.2: The step function

Sigmoid function

The sigmoid activation function shown on Figure 3.3(a) also maps the input to the interval Y[0;1] mimicking the behaviour of biological neurons, but it can give any value in between zero and one as output, this makes the sigmoid function appropriate for multi-class classification. The function itself is nonlinear, hence the decision boundary is nonlinear as well. As Figure 3.3(a), nicely demonstrates, if the argument of the function is lower than -3 or higher than +3, the change in outputs - predictions - is so small that it causes problems in the process of learning, as we would multiply very small numbers with very small numbers, resulting in even smaller numbers as weights. This problem is called the vanishing gradient problem, which is described in detail later on.

sigmoid	**tanh**
$\sigma(x) = \frac{1}{1+e^{-x}}$	$tanh(x) = \frac{(e^{2x}-1)}{(e^{2x}+1)}$
$\sigma(x) \longrightarrow [0;1]$	$tanh(x) \longrightarrow [-1;1]$

Tanh function

The hyperbolic tangent function is similar to the sigmoid function. The main difference between the two functions is the codomain, which in this case is Y[-1;1], while in case of the sigmoid function it was Y[0;1], as seen on Figure 3.3(b). This codomain makes the outputs of the function centered around 0, which is generally preferred. Having relatively strong negative or positive values is useful in specific scenarios where strong negative, strong positive and neutral inputs are expectable. The main strengths and weaknesses of the function are the same as in case of the sigmoid function.

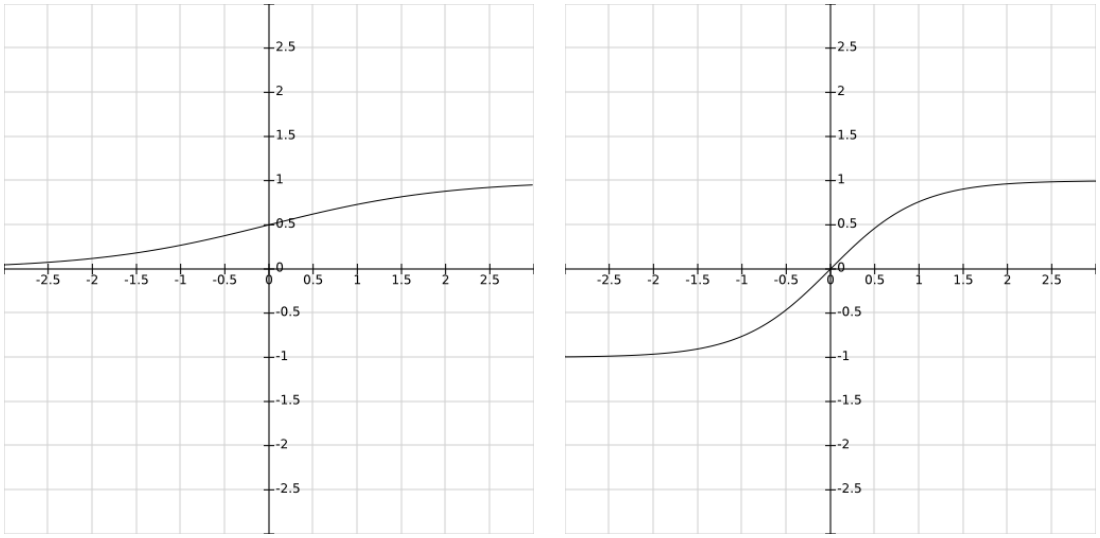

Figure 3.3: (a)Left: The sigmoid function
(b)Right: The tanh function

ReLU function

ReLU is a short for Rectifier Linear Unit. It maps the input values to the interval Y[0, ∞], so with the correct rules applied, it can be used for binary or multi-class classification as well. As shown by Figure 3.4(a), ReLU is a not completely differentiable function – having a corner at zero, where the derivative is technically not defined. The values greater than zero always have a positive gradient, this makes the training of neural networks much more efficient, and with efficiency comes speed. As explained in subsection 3.4.5, this activation function serves as a solution to the problem, as the derivatives of the function on the positive interval are positive. Half of the function is zero however, so are the derivatives on the negative interval. This causes a phenomenon called the Dead neuron problem. Luckily, experiments show that the right side of the function being non zero is 'good enough', the function performs well as an activation function.

Leaky ReLU function

Leaky ReLU is a modified version of the ReLU function, where the negative interval of the function has negative values and not zero, as Figure 3.4(b) shows. This modification was made to avoid the Dead Neuron problem. This function maps the input to the interval Y[$-\infty; \infty$], and the derivatives of it are always positive, just like in case of the sigmoid or the tanh activation functions, meaning that the outputs can not be centered around zero.

ReLU	**Leaky ReLU**
$R(x) = max(0, x)$	$R(x) = max(0, x),\ if\ x\ \geq\ 0$
	$R(x) = min(0, a \times x),\ if\ x\ <\ 0,$
	$where\ a\ \geq\ 0\ is\ a\ constraint$
$R(x) \longrightarrow [0; \infty]$	$R(x) \longrightarrow [-\infty; \infty]$

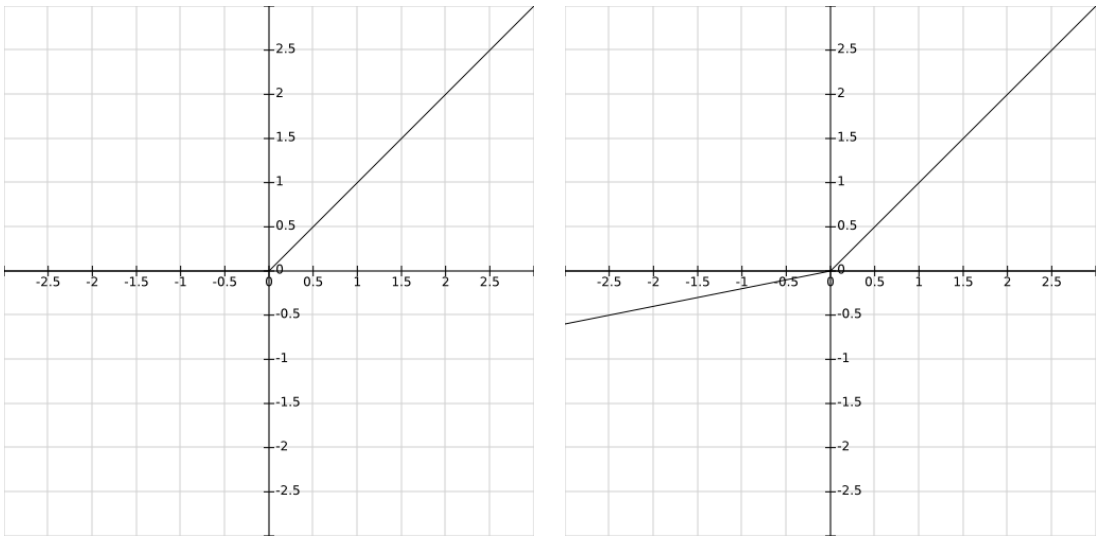

Figure 3.4: (a)Left: The ReLU function
(b)Right: The leaky ReLU function

ELU function

ELU is a short for Exponential Linear Unit. This function, as seen on Figure 3.5(a), has a different shape in the negative interval than ReLU and the Leaky ReLU, as unlike ReLU it decreases, and decreases more steadily than the Leaky ReLU. This function maps the inputs to the interval Y$[-\infty; \infty]$, meaning that it is possible for the outputs to have a mean close to zero, which is generally preferred in machine learning. This activation function is able to make the training process much faster in some cases than it would be with ReLU, but in some cases ReLU is a better choice.

Softplus function

The Softplus function has a similar shape as the ReLU, this can be observed on Figure 3.5(b). The visible difference is that Softplus has a curve, not a corner, which caused ReLU to be not completely differentiable. Having large inputs, the image of the function looks close to linear, as we take the logarithm of the exponent. When using ReLU and Softplus functions as the activation function, Vanishing gradient problem occurs, but as mentioned above, these functions seem to be viable options and are well performing activation functions.

ELU	**Softplus**
$E(x) = x,\ if\ x\ >\ 0$	$S(x) = log(1 + e^x)$
$E(x) = a \times (e^x - 1),\ else\ ,$	
$where\ a\ \geq\ 0\ is\ a\ constraint$	
$E(x) \longrightarrow [-\infty; \infty]$	$S(x) \longrightarrow [0; \infty]$

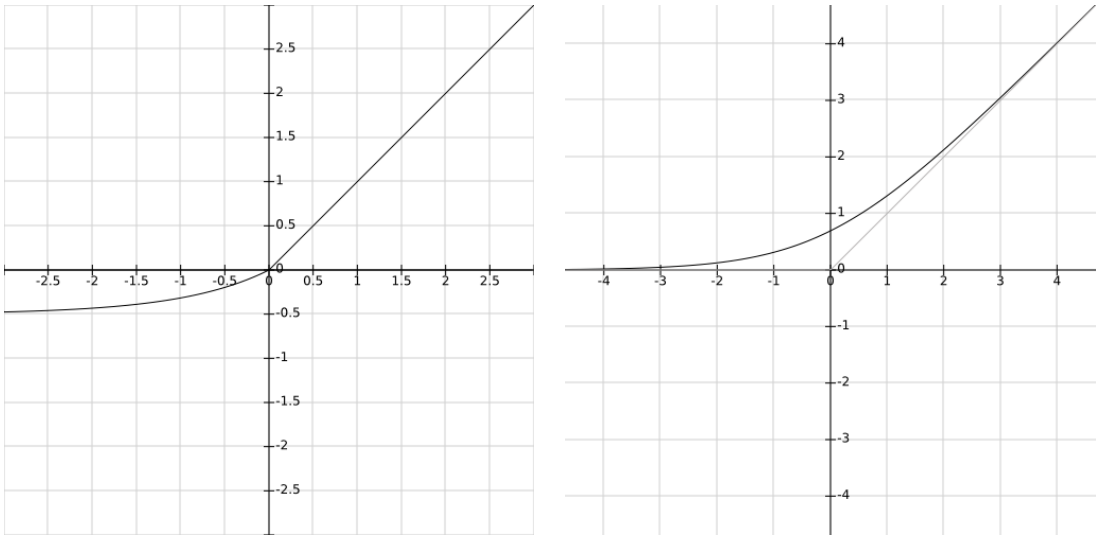

Figure 3.5: (a)Left: The ELU function
(b)Right: The softplus function

3.4.3 Loss functions

In the process of training neural networks, we have to apply specific metrics to express the goodness of the current state of a model. For this purpose, there are several mathematical equations expressing how far a prediction is from the expected prediction. A common loss function is categorical cross-entropy [9], which can be mathematically expressed as:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Where:
M - number of classes (dog, cat, fish)
log - the natural log
y - binary indicator (0 or 1) if class label **c**
the correct classification for observation **o**
p - predicted probability observation **o** is of class **c**

3.4.4 Gradient descent

Gradient descent is an iterative algorithm that minimises loss, given a cost function. It is iterative, so in an optimal case, with every iteration, the cost of gets lower and lower, the neural network gets more and more trained. The required loss value depends on the task, the architecture of the neural network and on the cost function as well. A representation of Gradient descent is to be seen on Figure 3.6.

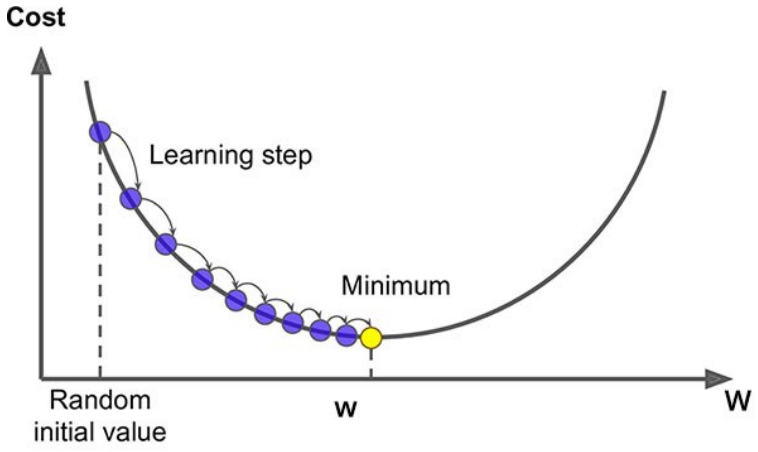

Figure 3.6: The gradient descent converges to the cost minimum with each iteration [10]

3.4.5 Vanishing gradient problem

As mentioned in the introduction of Sigmoid and hyperbolic tangent functions, researchers had to face the problem of vanishing gradients. This problem occurs because the derivative of these and similar functions are very close to zero on a big interval, and the maximum of the derivative is low as well, 0.25 in case of the sigmoid function. The low absolute value of derivatives cause problems because of the high number of layers in a neural

network. The mathematical equation describing the behaviour of a deep neural network is a composite function of nested Sigmoid and/or hyperbolic tangent functions. Under the training process, these composite functions result in the usage of the chain rule. When using the chain rule, composite functions become multiplications, and the multiplication of small (≥ 0.25) numbers results in even smaller numbers. As neural networks get deeper and deeper, the neurons close to the input are nearly not trained at all, as the weights assigned to them are very low. One solution to this problem is called greedy layer-wise pretraining, but the easier and in many cases more feasible solution is to just not use such functions [11].

3.5 Artificial Neural Networks for image processing

3.5.1 Preprocessing images for Machine Learning

For simple, formatted numerical data an appropriate data structure is viable in most of the cases. For image processing, however, it is essential to preprocess the images given to neural networks as inputs. The size of the image matters, since training time of neural networks increases exponentially as the size of the images grow, so resizing the images helps reducing the training time. On the other hand, we may lose some detail features: the accuracy of the model might get worse if trained on half size images instead of full ones. Another often used technique is color space transformation, which might be necessary because of implementational compatibility (e.g., in case of RGB-BGR color spaces), or to reduce the dimensions of an image. In this case, reducing the dimensions only means that we use the gray-scale version of an image, not the RGB form, so the dimensions of an image becomes Width $\times$ Height $\times$ 1, instead of Width $\times$ Height $\times$ 3 [12].

3.5.2 Filters

Filters, or kernels in image processing are N $\times$ K shaped matrices, used to find features of a H $\times$ W sized image by shifting the matrix on the image and calculating a score for the current area. By convention, these are square shaped (N = K), and small, typical sizes are 3×3, 5×5, 7×7. It is usually better to work with small sized filters, but a lot of them, with fewer but orders of magnitude greater sized filters as these matrices will be multiplied, and the cost of matrix multiplication grows exponentially with the size growing. For the filters to behave the same way in the middle of an image than on the edge or in the corners, a technique called padding is used to make the filter able to go farther than the edges of the image, so the same effect takes place on the whole image, the filter can work consistently [12].

3.5.3 Convolution in Artificial Neural Networks

Convolution in ANNs is an essential operation, as it is used to calculate the result of a filter applied to an image. Here, B is a feature vector, the output of an A image being convolved with an ω filter [12]. The mathematical form of convolution is the next:

$$B(i,j,c) = \sum_{i'=1}^{K} \sum_{j'=1}^{K} \sum_{c'=1}^{C} A(i+i', j+j', c') \times \omega(c', i', j', c)$$

Where:

$B = A * \omega$ $\quad$ $shape(\omega) = C_1 \times K \times K \times C_2$
$shape(A) = W \times H \times C_1$ $\quad$ $shape(B) = W \times H \times C_2$

3.5.4 Convolutional Neural Network architecture

Convolutional Neural Networks (CNN-s) contain several kinds of layers. These kinds are shown in the chart below.

Type of layer	**Description**
Convolutional layer	Capable of creating feature maps.
Pooling layer	Capable of downsampling, maintains the most important features.
Fully connected input layer	Flattens the output of previous layers into vectors which later layers can work on.
Fully connected layer	Applies weights to the feature analysis to predict a more accurate label.
Fully connected output layer	Determines the class prediction for the image.

Convolutional and pooling layers are usually repeated after each other to get strong features trough feature maps. Then, through fully connected layers, the output is produced [13].

3.5.5 Feature Pyramid Network architecture

Feature Pyramid Networks (FPN-s) combine low-resolution, strong features with high resolution weak features using bottom-up pathway done by a **backbone CNN** providing convoluted and pooled inputs, and top-down pathways with lateral connections present in the FPN, as seen on Figure 3.7. This architecture results in strong feature maps on all scales [14].

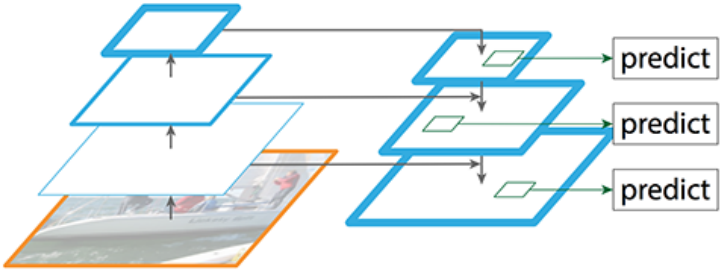

Figure 3.7: The FPN architecture showing the top-down pathway, with predictions made on each level [14]

3.6 Technologies utilized for image processing in our complex use-case

3.6.1 Python

Python [15] is a high level programming language, using an interpreter, hence it is often referred to as a script language. The syntax of Python is unique, targeting simplicity. This language is known for its quick prototyping capability and proof-of-concept attitude, as it has wrapper classes and packages for libraries implemented in more efficient programming languages, mostly in C or C++.

3.6.2 OpenCV

OpenCV [16] is an open source computer vision library, implemented in C/C++. The library has a python API, trough which we could utilize the methods implemented in it. OpenCV is mostly used for classical computer vision tasks, such as applying filters e.g. an edge detection filter, colour space transformation, and much more. In our case, OpenCV was mainly used for colour space transformation, drawing on the images and displaying them.

3.6.3 TensorFlow 2

TensorFlow 2 [17] is a widely used framework for machine learning. In the earlier version, it had a complicated syntax and was mostly used by experienced researchers, nowadays the TensorFlow 2 provides a much simpler syntax and an other machine learning library, called Keras [18] has been built into it, providing pre-constructed models, well known datasets and more. TensorFlow 2 provides tools to build custom neural networks, train existing ones on custom datasets, both with GPU acceleration, and many more tools and functions.

Chapter 4

Choreography Integration with the System of Systems

4.1 Choreography supported by Workflow Executors

As introduced in Section 2.3 and visualized by Figure 2.4, the Workflow Choreographer is a workflow governing unit in the Arrowhead Framework. It can create workflows from the Production Orders and allocate tasks to the Workflow Executors for execution while logging relevant data about every phase of the production.

The initial concept of the Workflow Choreographer in Arrowhead has aimed to execute only predefined tasks in static environments. In our research we went further and added new functions and features to the system making it capable of working in dynamic scenarios where occasionally some devices go offline even during execution and an other device must step in to accomplish the task.

The main new element introduced in this part of our work is the creation of Workflow Executors (WEs) and their integration with the Workflow Choreographer. WEs make it possible to establish connection between the AH core systems (Orchestrator, Choreographer) and the Application Systems (application service providers, consumers). In our previous work [7] and ven in the recent recent proceeding [19] the providers communicated with the Choreographer directly, which introduced significant dependency from the AH's point of view. Services provided by the Arrowhead Framework core systems should not introduce future implementation difficulties for the Application Systems therefore the independence between the core systems and the various application service providers was cut to almost zero by using WEs.

To highlight our new contribution, Figure 4.1 shows the high-level architecture of a Local Cloud with the Arrowhead Core Systems, the Application Systems, including the Workflow Executors and the various services they provide each other. To accomplish the execution of each task in the recipe, the Choreographer sends a request to the suitable Executor. In this case the WEs can be comprehended as special consumers by pushing the request they

get from the Choreographer to the corresponding providers, in other words forcing these providers to execute tasks the Choreographer previously allocated to them according to the production recipe by service consumption. The WE can monitor the status of the tasks under execution in various ways e.g., by periodically checking the blinking of a led sensor on the device, polling the providers regularly or waiting for callback messages from the providers executing the task.

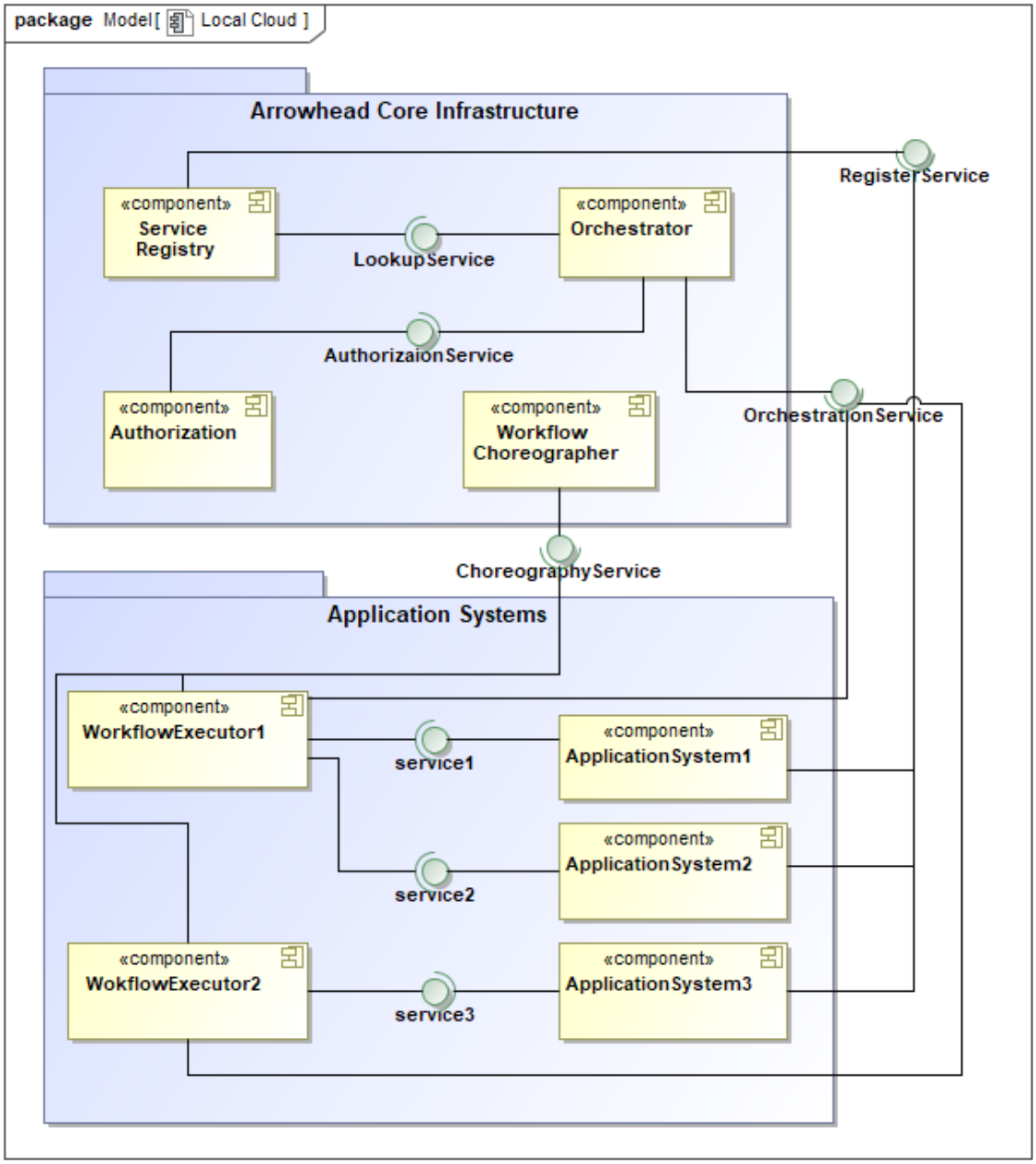

Figure 4.1: The connection between the WE, the AH Core Systems and the various Application Systems in an Arrowhead Local Cloud

A more complex behavior is depicted on Figure 4.2 which shows the sequence of messages and method calls during workflow execution.

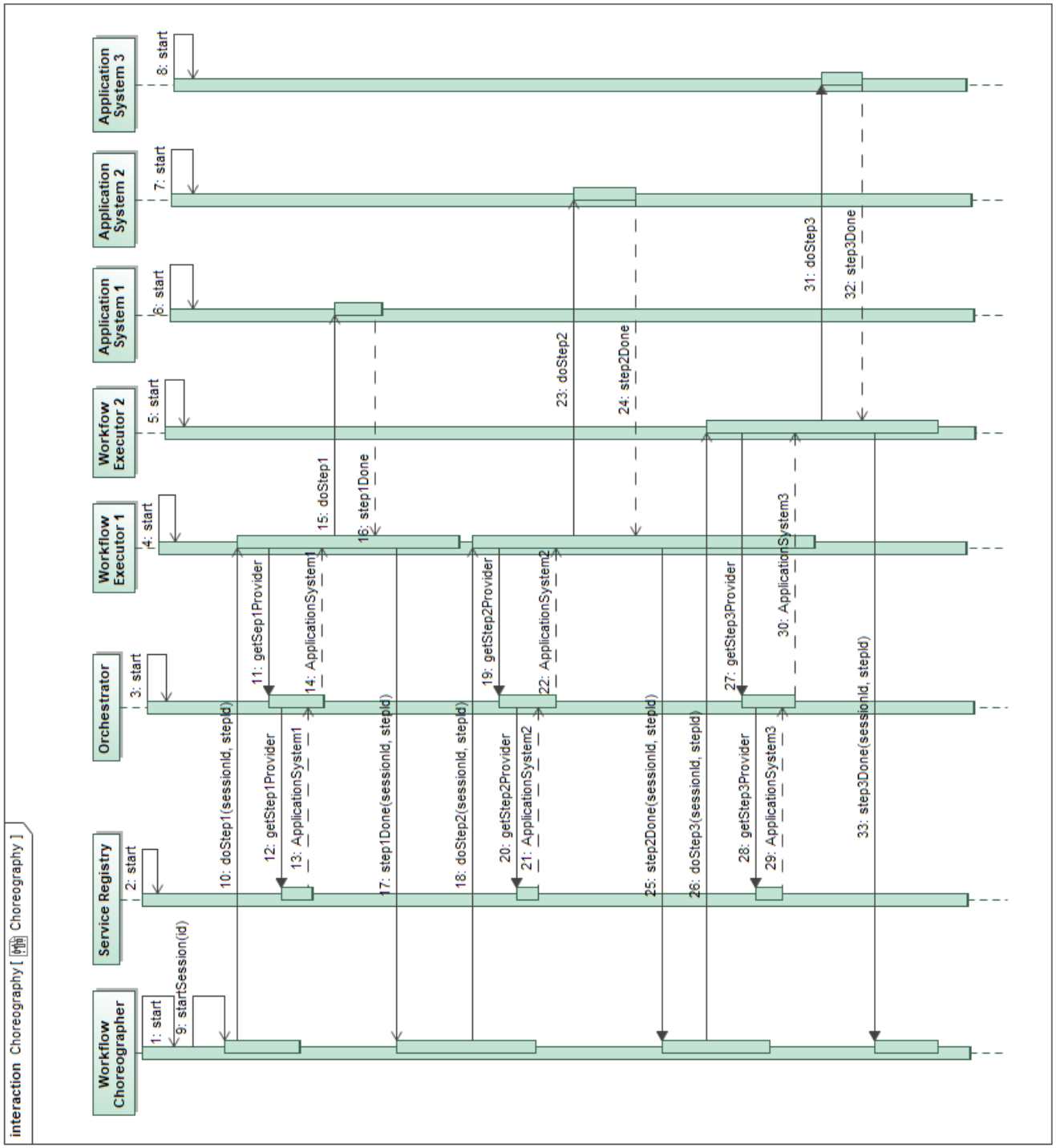

Figure 4.2: Sequence diagram showing workflow execution with Workflow Executors

When the Choreographer is tasked to manufacture a product, first it checks if the Local Cloud contains all the necessary providers and Workflow Executors according to the recipe. If it finds that the product could not be manufactured under the current circumstances then a system operator is warned to fill in the gaps by either adding the missing provider systems to the environment or by extending the functions of the current WEs. In some cases even adding and implementing new WEs may be needed.

If everything is set then plan (recipe) execution can be started by the Choreographer. In the simplest example case the plan has only three steps: *Step1*, *Step2* and *Step3* in this order. Let us use the example where *Step1* and *Step2* is realized by *Workflow Executor 1*, *Step3* by *Workflow Executor 2*. See the sequence diagram of this generic example in Figure 4.2. The Choreographer searches for the suitable WE that can force the execution of *Step1*. As *Workflow Executor 1* is eligible to perform this step, the Choreographer sends the information which the Executor has to send back if execution is done or an unexpected error happened to the Choreographer for the identification of the running step in the database. The next task is for the Executor to request a provider from the Orchestrator to have it perform the step which the Choreographer forwarded. If execution is successful then the WE calls back to the Choreographer to start the possible next step (in this case *Step2*) or next steps if there are multiple. The manufacturing of a product is only successful if the execution of all steps in the recipe ended successfully. In the example case shown on the figure if the execution of *Step3* finished without error then the workflow choreography was successful.

4.2 Recipe Description Language

To assist recipe creation we invented our very own Recipe Description Language for the Workflow Choreographer which was based on the widely used Business Process Model and Notation (BPMN) representation. Table 4.1 shows the BPMN representations and their JSON counterpart in the Choreograper's own language. Here the question is rightfully asked why do we need a separate language if BPMN is so similar to ours. The answer is that all Arrowhead core systems use the JSON format to communicate between each other, furthermore our JSON although similar to BPMN has a much greater expressive power by the inclusion of additional information. Therefore BPMN should be only used as a skeleton for production plans and completed afterwards.

BPMN representation	**JSON language element**
Model	Plan
SubProcess	Action
Task	Step

Table 4.1: Mapping the BPMN representations to their JSON DL counterpart

As building a new language from scratch is a complex and time consuming task, we used JavaScript Object Notation (JSON) to create, visualize and forward recipes in a lightweight data format. The main goal was to create a description for the recipes which is easily understandable for man and machine equally.

Figure 4.3 summarizes the elements of the language and the relationships between them. As it can be seen the production recipes are called *Plans* which contain *Actions* and *Actions* contain *Steps*. An *Action* can have several first steps but a maximum of one following *Action* as opposed to *Steps* which can be followed by either zero, one or several other *Steps*. It is also mandatory to define a service by which the related production step will be performed. Services can have versions defined either by stating the version or describing the minimum and maximum versions. This will be necessary to find WEs that match the version of the service description. The users can also add a quantity in a *Plan* to each *Step*. The syntax of the completed language is visualized on Figure 4.4.

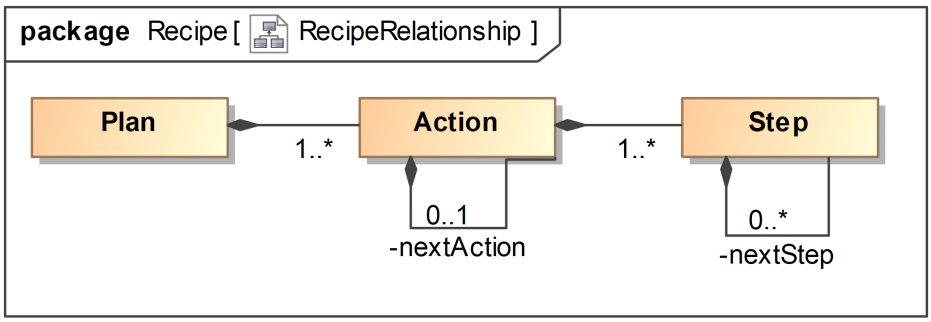

Figure 4.3: Relationship of the Recipe Decription elements

There are other features worth mentioning regarding to this Description Language such as solutions to include Orchestration requests in the recipe or precondition constraints.

Orchestration request inclusion is a cornerstone of this design. By adding data about the possible request to the recipe the WEs can without a doubt find the appropriate provider needed for the given step. This also enables system operators to add a list of preferred providers, orchestration flags, Quality of Service (QoS) requirements and other additional metadata to a recipe.

Precondition constraints mean that other orchestration requests can be added to a step besides the main service request. These services should be called strictly before executing the main service call and it serves to acquire possible information needed for the main consumption e.g., sensor data or camera data for asset tracking.

```
{
  "name": "String",
  "firstActionName": "String",
  "actions": [
    {
          "name": "String",
      "firstStepNames": [
        "String"
      ],
      "steps": [
        {
          "name": "String",
          "nextStepNames": [
            "String"
          ],
          "quantity": 1,
          "usedService": {
            "requestedService": {
              "maxVersionRequirement": "Number",
              "minVersionRequirement": "Number",
              "serviceDefinitionRequirement": "String"
            }
        "..."
}
```

Figure 4.4: The Choreographer Recipe Description Language with basic functions enabled

4.3 Supporting external tools

A tool is a software or a hardware entity that supports engineering activities operated either in design-time or in run-time. Although the phases of the engineering principle can be managed without tools, most engineers probably use some for their tasks. A toolchain is a collection of tools, potentially organized in chain-based or parallel structures.

Figure 4.5 introduces categories of tasks that tools can support in connection with the AH Framework. For the given requirements, Functional Design can be supported by MagicDraw [20], Papyrus [21], or other SoS Modeling tools. Among others, the Vorto language can assist Procurement and Engineering by device catalogs, while for Deployment and Management tasks the Arrowhead Management Tool has various services. Operation and Management is governed by the Arrowhead Framework itself. The connection between AH and these tools will be elaborated in the next section.

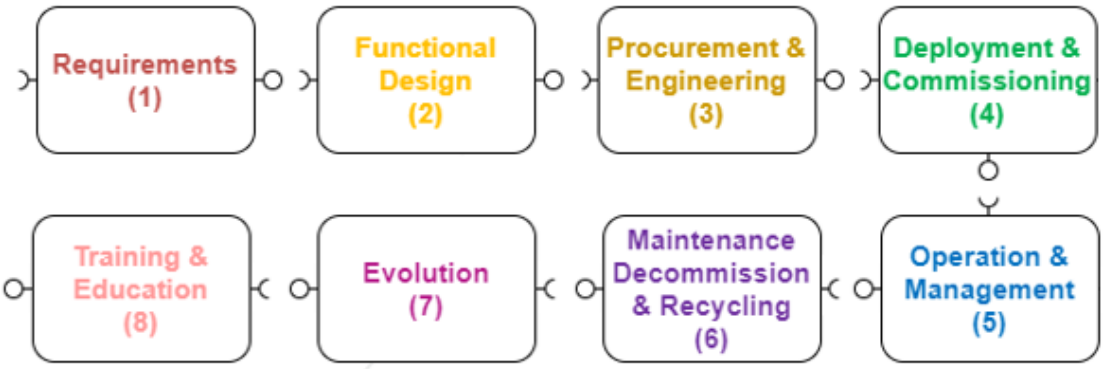

Figure 4.5: The Engineering Toolchain extended by Arrowhead Tools

4.3.1 Generating production plans from SysML with MagicDraw

MagicDraw is a visual UML, SysML, BPMN, and UPDM modeling tool designed for business analysts and program developers. To create connection with the AH Framework, a plugin was developed in MagicDraw with the help of IncQueryLabs of the Arrowhead Tools project. This plugin can convert Business Process Model and Notation (BPMN) to the JSON format of the Recipe Description Language in the Workflow Choreographer. BPMN is a widely used modelling tool in factories and production facilities to generate, customize and test out workflow. With this plugin conversion between MagicDraw and the Arrowhead Framework it is only one click for the system operators to create production recipes compatible with the Choreographer.

An example BPMN diagram can be seen on Figure 4.6. In this scope *Actions* are realized by Sub-processes and *Steps* by Tasks inside a Sub-process and a Model is equal to a *Plan*. Figure 4.7 demonstrates part of the JSON created by the plugin. It can be seen that the name of the *Plan* is the same as the name of the Model and *Action1* has two first steps: *Step1* and *Step2* just like on its BPMN counterpart shown on Figure 4.6.

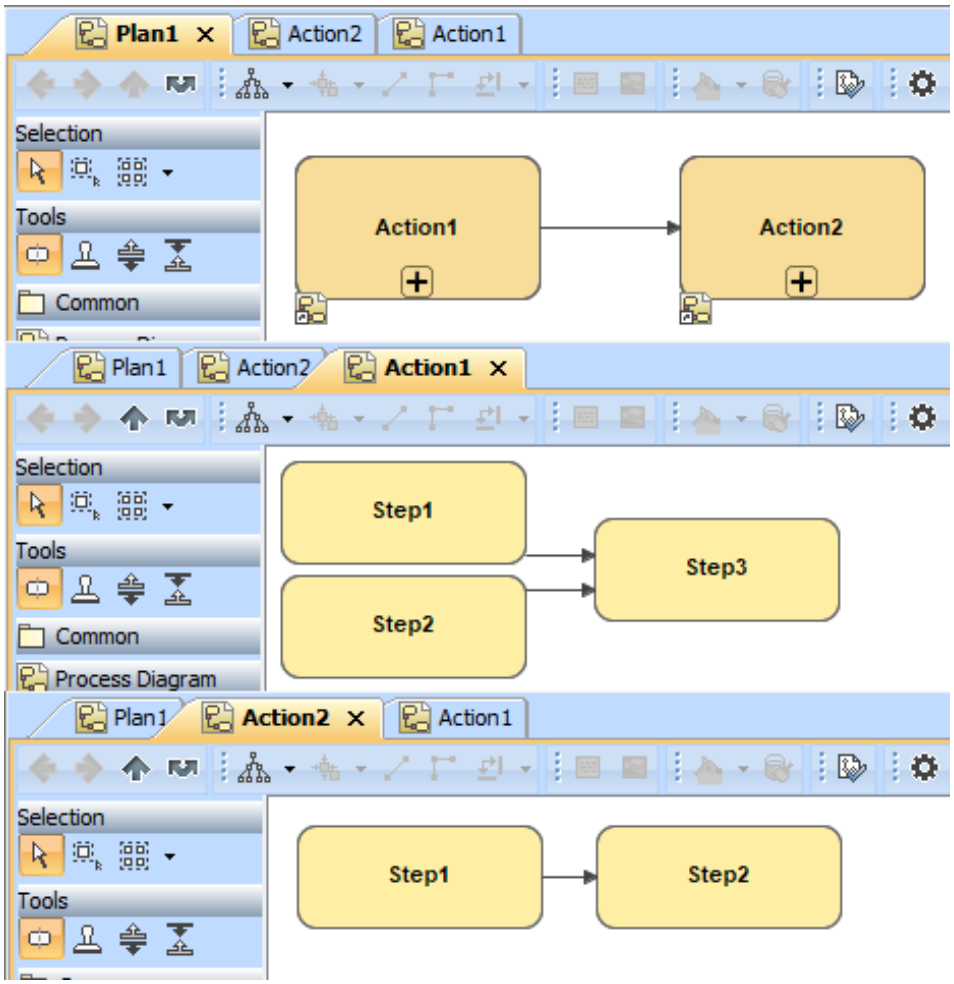

Figure 4.6: The BPMN diagram of a simple task.

```
{
  "firstActionName": "Action1",
  "name": "Plan1",
  "actions": [
    {
      "name": "Action1",
      "nextActionName": "Action2",
      "firstStepNames": [
        "Step1",
        "Step2"
      ],
      "steps": [
        {
          "name": "Step1",
          "nextStepNames": [
            "Step3"
          ],
          "quantity": 1,
          "usedService": {
            "requestedService": {
              "interfaceRequirements": [
                "HTTP-SECURE-JSON",
                "HTTPS-SECURE-JSON",
                "HTTP-INSECURE-JSON"
              ],
              "versionRequirement": 1,
              "serviceDefinitionRequirement": "service1"
            }
        "..."
}
```


Figure 4.7: Part of the Choreographer Recipe created from the BPMN description on Figure 4.6

4.3.2 Eclipse Vorto integration possibilities

Vorto is a domain-specific language (DSL) developed by the Eclipse Foundation based on other well known languages such as Java focusing on expressing device functionality. With Vorto the capabilities and functionalities of devices can be described as an Information Model. These consist of re-usable, technology independent Function Blocks. It can be widely used in IoT applications and solutions as a repository for the devices in the system [22].

With the integration of Vorto to Arrowhead through MagicDraw device catalogs can make it simpler to address providers by consumers and use each others interface and exchange data between them. This significantly shortens implementation times of each Arrowhead

Application System furthermore helps choreography tremendously by clearly describing how data is exchanged between service providers and the corresponding Workflow Executors. As a solution for automatic model data exchange, it is one of the biggest upcoming challenges for the Workflow Choreograper that Vorto device catalogs should be used during recipe execution planning.

An example of an Information Model for the DOBOT Magician robotic arm (that we use in our complex case study) can be seen on Figure 4.8.

```
infomodel DobotMagicianRoboticArm {
        functionblocks {
            headType as HeadType "the currently used head on the arm"
            interactWithItem as InteractWithItem
            armState as ArmState
        }
}
```

Figure 4.8: Vorto Information Model of the DOBOT Magician

The *InteractWithItem* Function Block with its operations and events is demonstrated on Figure 4.9. The operations include grabbing, dropping or moving an item to a position, the events describe what happened during the execution of this function block. When the robotic arms grabs or releases an item the first two events are triggered respectively if an item and the time of the operation is present. The third event type triggers when the robotic arm is moving but only if both the starting and goal locations of the robotic arm are available.

```
functionblock InteractWithItem extends
  vorto.private.ahtoolsplugin.CurrentItem {
    operations{
        breakable grabItem(location as Location) returns boolean
        "grabs the item from the given location"
        dropItem (location as Location) returns boolean
        "drops the item to the given location "
        moveTo (position as Location)
        "the arm's head moves to the given location"
    }

    events {
        ItemGrabbed {
            mandatory item as Item
            mandatory timestamp as dateTime
        }
        ItemDropped {
            mandatory item as Item
            mandatory timestamp as dateTime
        }
        ArmMoving {
            mandatory start as Location
            mandatory goal as Location
        }
    }
}
```

Figure 4.9: The InteractWithItem Function block with the operations and events.

4.3.3 The Arrowhead Management tool

The Arrowhead Management tool has complex means to manage the core services of the Arrowhead Framework. It supports adding Orchestration rules, Service Registry entries and intracloud or itercloud Authorization rules. It has a component for the Choreographer as well, giving a user a polished Graphical User Interface (GUI) to add workflows (Plans) to the local cloud and executing them.

Figure 4.10 illustrates the GUI of the Management Tool with the Choreographer option on the left. Here the user can expand a *Plan* to see its details by clicking on it or either start or delete it with the corresponding button. Clicking on the *Play* button enables the user to start multiple plans at once and the *Add* button is for adding new *Plans* to the

system. Figure 4.11 demonstrates the form to be filled by the user when adding new *Plans* to the database.

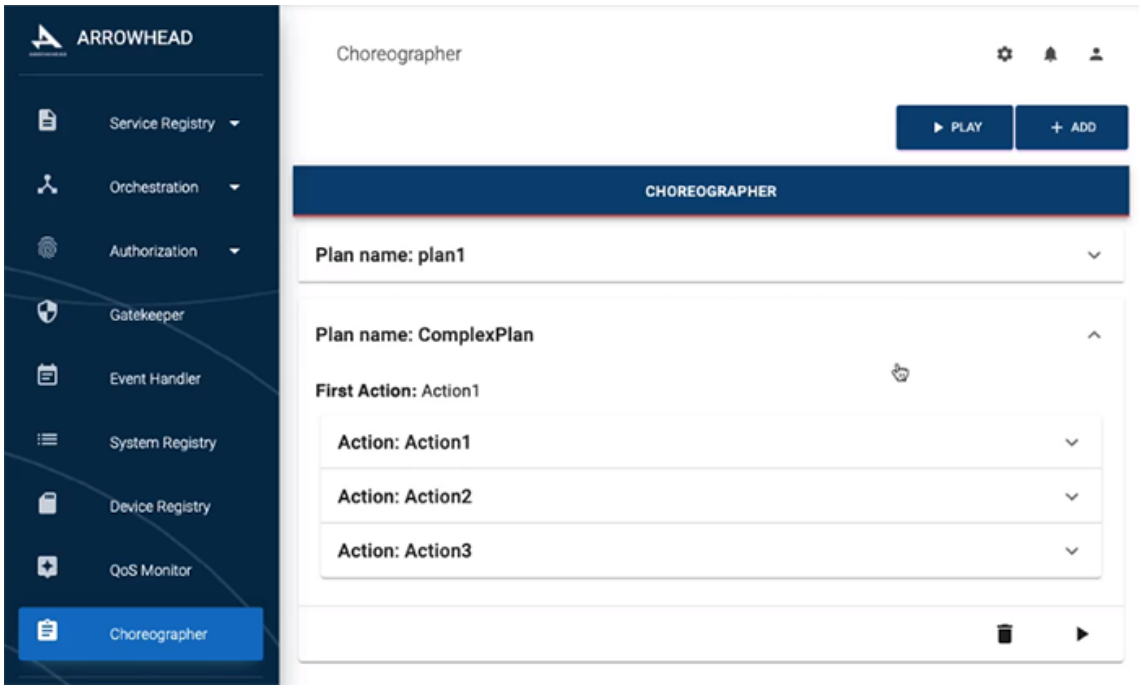

Figure 4.10: The graphical user interface of the management tool

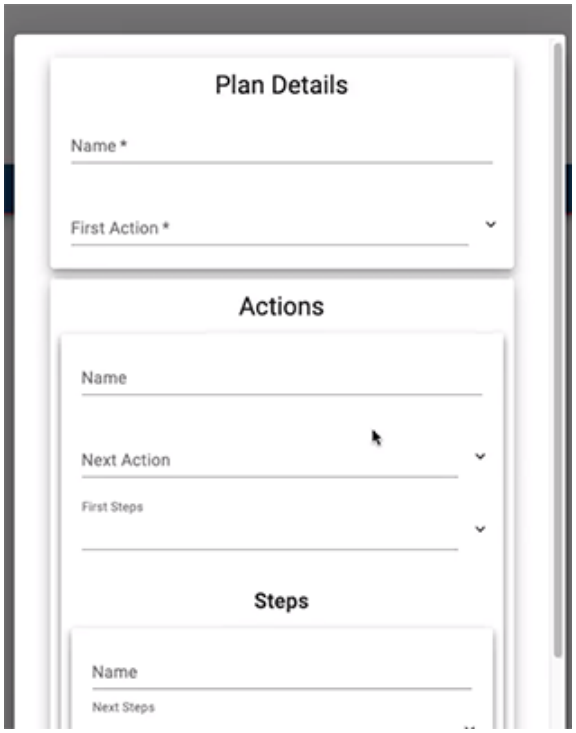

Figure 4.11: The form of adding a new *Plan* to the Choreography System

4.4 Conclusions for Choreography Integration

To summarize, this section introduced a new approach to workflow execution in industrial environments, more specifically in those implemented as Arrowhead Local Clouds. With the Workflow Executors the system became much more flexible and the bottleneck around the Workflow Choreographer has been eliminated by evening the tasks between the new Executors and the Choreographer.

As the number of Executors in a local cloud are not limited, this solution significantly lowers the centralization in the System of Systems. The Choreographer does not have to use the other core systems, only govern the Executors to execute the tasks on their appropriate workstations. With the BPMN-based, newly expanded Recipe Description Language the choreography of workflows became simpler, more manageable and more customizable in every way.

There were also external tools introduced which support engineering activities through the whole system. The newly developed MagicDraw plugin enables to convert BPMN diagrams to workflow plan skeletons and Eclipse Vorto integration will make it possible to organize devices in a local cloud. The Arrowhead Management tool supports the Choreographer as well: it has a graphical user interface, making it easier for users to interact with the Choreographer and other core systems.

Chapter 5

Image Processing Support for the Local Cloud

5.1 Video and Image processing systems and subsystems

For the complex use-case scenario presented in Chapter 7, we had to implement a video and image processing system and integrate it as an Arrowhead Framework compatible Application System. This task consists of subtasks such as the implementation of a physical system capable of producing and transporting video feed, and the creation of a high level logic which then can be implemented by programs and scripts. These subtasks have their corresponding subsystems in the complex video processing system of the demonstration.

5.2 The video stream provider in the demonstration

The first step in video processing is getting a video feed. In our case, the source of video, therefore the source of the data to be processed, was a Raspberry Pi 4 model B, with the official Raspberry Pi Camera Module v2 connected to it, which can be seen on Figure 5.1.

The Raspberry Pi based camera system was integrated into the Arrowhead Framework, serving as the provider of video feed. To implement an Arrowhead Framework compatible system, the first mandatory step was to create a provider system with the capability of producing video stream, then came the creation of a service which can be later consumed by other Arrowhead compatible systems. The Raspberry Pi has a quite convenient python API for handling the camera module, hence a simple python script could solve the video feed creation. To be able to provide this feed to consumers, we had to implement an Arrowhead provider system which can be done using the Java programming language and the corresponding Arrowhead Framework libraries. In the complex use-case scenario, this was the source of data provided to the also Arrowhead Framework compatible image processing system.

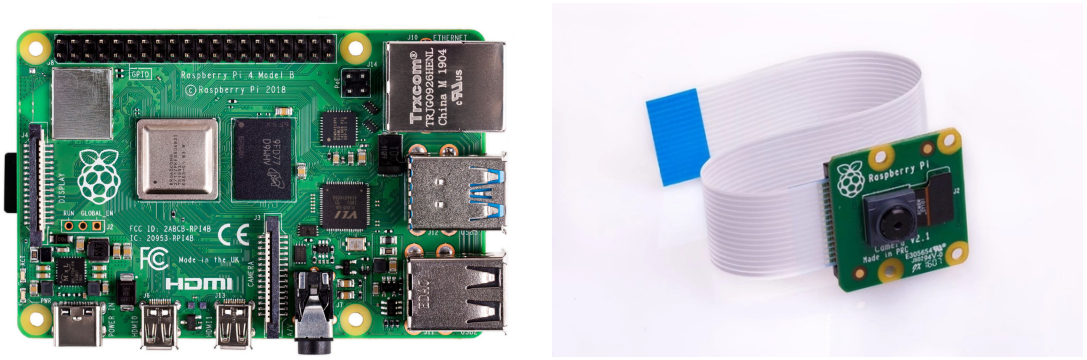

Figure 5.1: (a)Left: The Raspberry Pi model used in the demonstration.
(b)Right: The official camera module used in the demonstration

5.3 The implemented Image Processing System

As mentioned in Section 5.1 and is shown on Figure 5.2, after the production of the video feed, our next task was to process it. For this purpose, we used the highest-level approach possible, and as this solution includes neural networks, this high-level approach resulted in faster prototyping and development of the image processing system, which is an important aspect in most technology researches, as dead ends can be detected in the early phases of development.

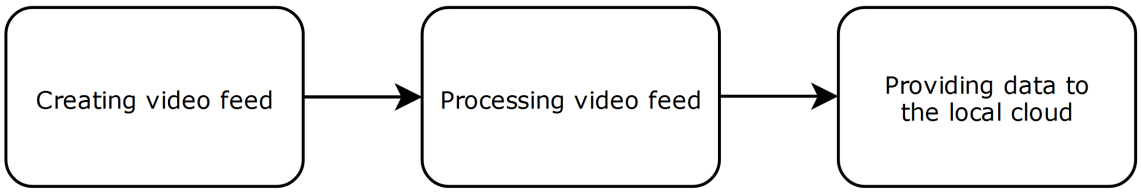

Figure 5.2: The main steps of creating a computer vision support for local clouds.

5.3.1 The process of training a neural network

The main tool used to train the neural network in our case was TensorFlow 2 [17]. With the built-in tools of TensorFlow 2, it was much simpler to train a model, especially custom training a previously trained model which was available in the TensorFlow 2 Detection Model Zoo [23]. One important step in training neural networks is gathering and labeling data which then is split into training and testing data. As our use case scenarios extensively use rubber ducks as "production objects" –that are handled with robotic arms–, these steps were done by hand, as not many databases have thousands of labeled pictures of rubber ducks. The preprocessing of the collected images was done by a tool called LabelImg [24], the main GUI of which is shown on Figure 5.3, the main functions are labeling the image in Pascal/VOC or YOLO format - in our case Pascal/VOC was the correct format.

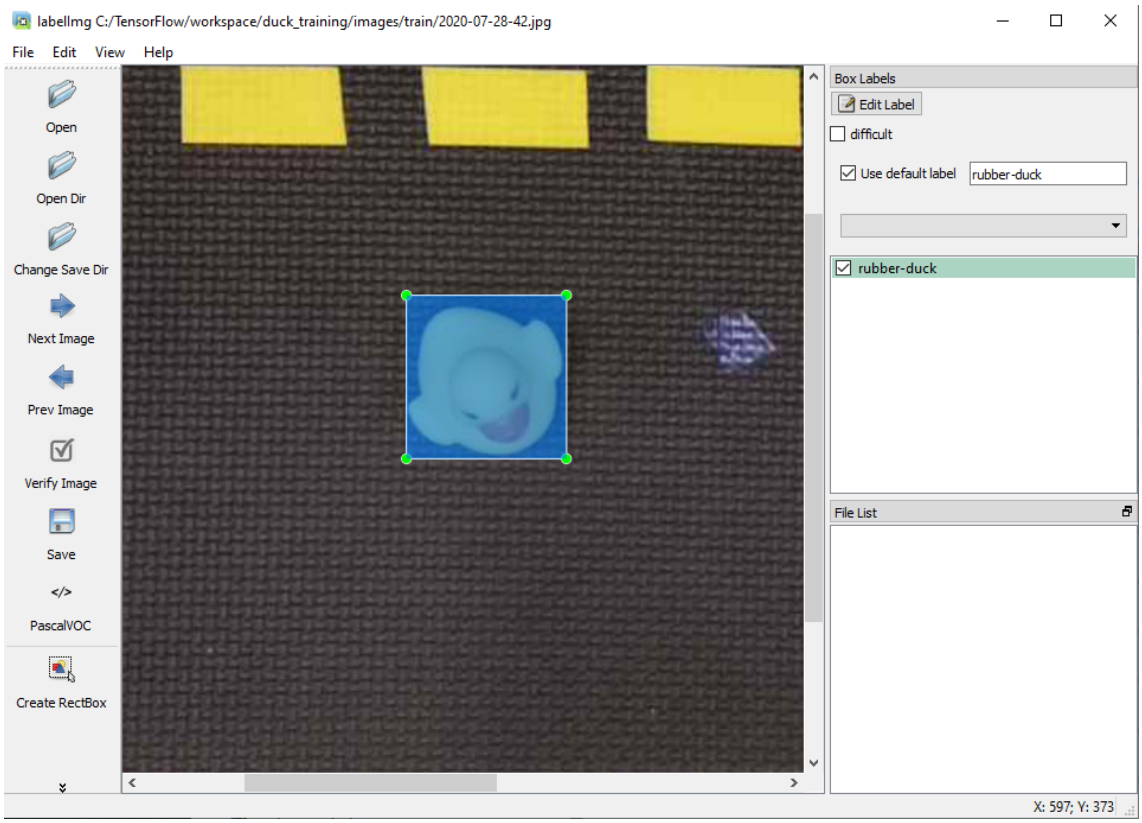

Figure 5.3: The GUI of the LabelImg tool, showing the main functions and options.

As soon as we have prepared the required datasets, a TensorFlow 2 function can be used to create the annotation files corresponding to our labeled train and test datasets. An other TensorFlow 2 function is then used to train the model.

When the training was done, we implemented a program which can produce the required position data, and is capable of providing real-time video feed of the detection. As the custom trained model was basically a single class classifier, labeling the images was unnecessary. Single and multi-object detection records are shown on Figure 5.4(a) and (b).

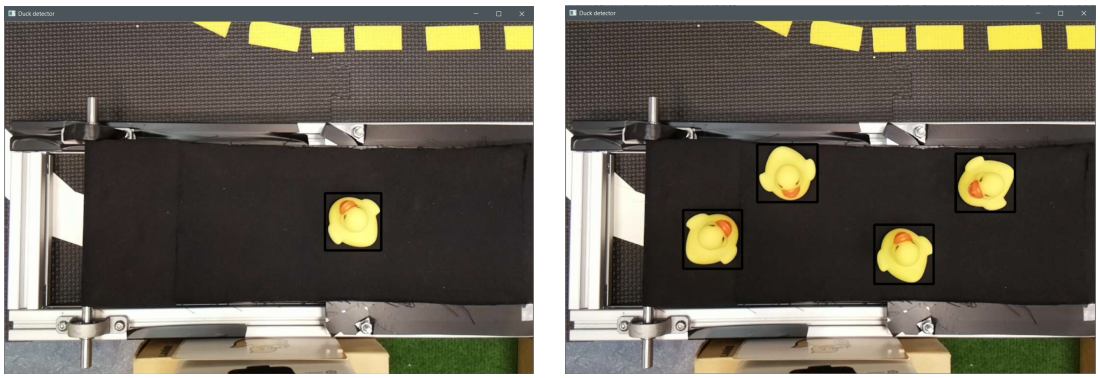

Figure 5.4: (a)Left: Single object detected by the neural network. (b)Right: Multi-object detection by the neural network.

5.3.2 The pre-trained model used for the project

There are dozens of pre-trained models in the TensorFlow 2 Detection Model Zoo [23], so after deciding to use neural networks and deep learning for computer vision tasks, a wide spectrum of models is available. Different models are optimized for different purposes or are designed to operate in different environments (desktop or mobile environment, embedded systems, etc.). In our case, a model designed to function in desktop environment was convenient as the hardware we were working with met the hardware requirements of such models, but a performance-wise efficient model was required due to hardware limitations. After training and testing multiple models, we decided to use the SSD ResNet50 V1 FPN 640x640 (RetinaNet50) where SSD means Single Shot Detector. We decided to use this model because it is a lightweight model in its category, but has an acceptable accuracy after training on our hand-collected dataset.

5.3.3 Testing the trained model

Testing the model begins while training the model. The training process provides real time logs showing important information about the process: number of current step (every hundredth), per-step-time, loss. A line of the outputted log can be seen on Figure 5.5.

```
I1024 12:58:47.572535 13008 model_lib_v2.py:649] Step 10000 per-step time 0.725s loss=1.578
```

Figure 5.5: A line of the training log. Date, number of current step, per-step-time and loss is shown.

These numbers can tell us whether the training goes correctly or something is going wrong. The general starting loss value was around 30-35, and this decreased to around 1-2 after 10.000 steps. Per-step-time can tell us if the model is struggling with the calculations, in which case we should probably configure the training better. This value of this indicator was between 0.7 and 0.9 in our case. To be able to visualize these indicators, TensorFlow 2 has a built-in tool, called Tensorboard. Tensorboard shows real-time data, but is also capable of opening saved logs to process them later, the format and style can be seen on Figure 5.6.

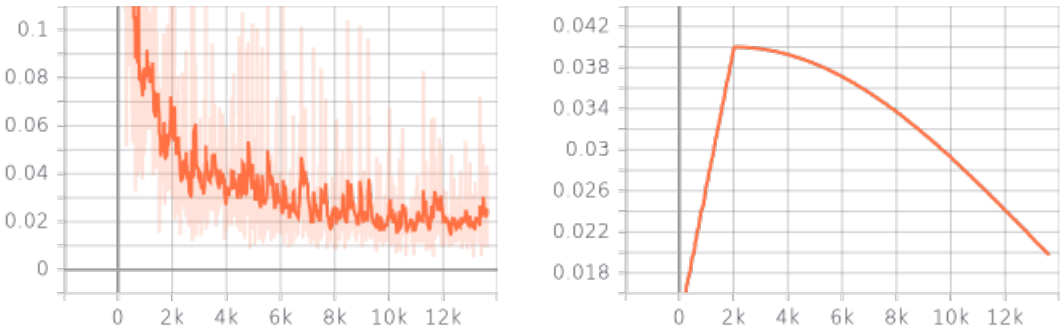

Figure 5.6: The appearance of diagrams automatically produced by Tensorboard.

5.4 The Arrowhead Framework compatible systems created for image processing

Previously, we presented the creation of the video and image processing system, starting with the physical construction, then the creation of the video processing algorithm using neural networks. Now is the time to present the Arrowhead compatible system which will consume the previously produced video feed service, process the incoming data, then provide data to the local cloud of the complex demonstration. This high-level view can be seen on Figure 5.7.

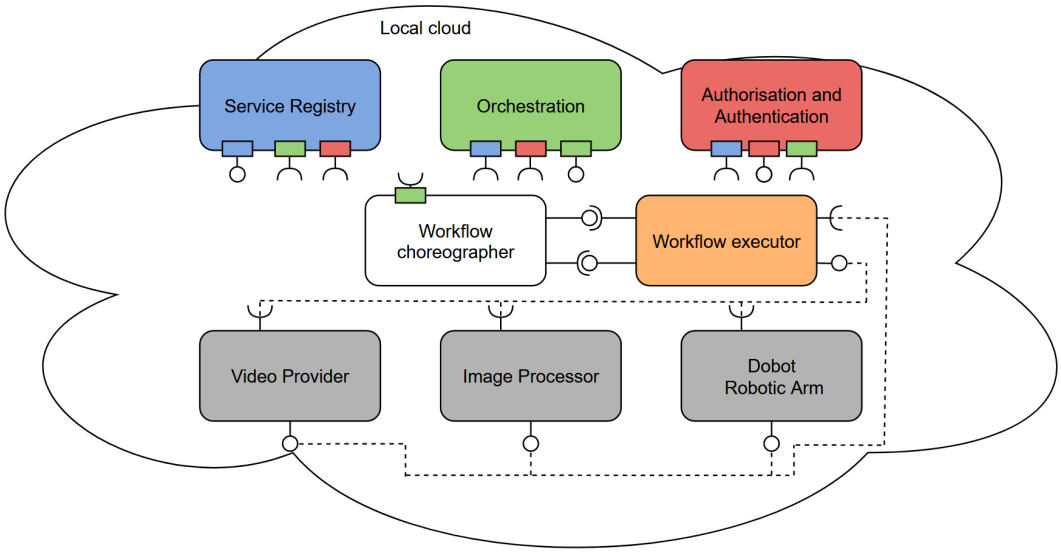

Figure 5.7: A system-level representation of the Video and Image processing support for the Local Cloud.

In an industrial environment, a similar relation is present in case of a camera recording the production line for a computer vision based quality management system, or in case of a camera on a production line machine which is directly connected to the machine and the control algorithm of it is based on the video feed provided by the camera. In these cases the camera is mostly directly wired to the computers or machines as real-time requirements are present and direct connection is a reliable, fast connection. In our case however, the local cloud architecture grants a viable response time amongst the systems present in it, hence there is no need for wired connections.

Physical independency, modularity and dynamic organizability of industrial systems and machines can result in higher cost-efficiency as the service of the machines can be scheduled better and the machines not working can be replaced dynamically. This modularity in our case is supported by independent systems. These systems are capable of consuming the data of any other system providing the required service and they also are capable of producing data or providing other kind of service for other systems or the user.

The Video Stream Provider does not consume any application level service, however is physically capable of creating a video stream into a video feed service which this system

provides. The Image Processing System is in the middle of this part of the process as seen on Figure 5.7. This system consumes the service of the previous one to get images to process. After an image has been processed, the position of the searched object - or a value meaning it was not find - is available. The next step is when the robotic arms get a command to move. When the command arrives to the Arrowhead compatible system of the robotic arms, it consumes the position data provided by the Image Processing system, and acts according to the implemented logic of the robotic arms. In the current implementation of the Arrowhead Workflow Executor system, the video data and the position data travels trough it, avoiding hard-coded messaging solutions, e.g. a hard coded IP address in an application level system.

Chapter 6

Simple Use Cases

To test the initial version of the Workflow Choreographer, some use cases were developed, in which typical workflow steps – such as packing and moving – are modelled. All of the use cases are executed by DOBOT Magician (hereafter: DOBOT) [25], which is a multi-functional desktop robotic arm, installed with different end-tools. These use cases were also presented (by us) at IEEE IECON in 2020 [19].

There are three workstations in the two use cases: A, B and C. All of these have different coordinates in the DOBOT's coordinate system. In these cases, the DOBOT can move any objects (in the current case rubber ducks) between these workstations in any order, defined by the production recipe. The next sections describe these thoroughly with the related sequence diagrams and illustrative figures. In addition, the demonstration videos of the use cases are also available online [26].

6.1 Use case 1 - Object packing

In this use case, first, the robot grabs the object from workstation 'A', moves it to workstation 'C' then grabs the object from workstation 'C' and moves it to workstation 'B' and so on. Before executing each step, the Workflow Choreographer needs a device that provides the corresponding service which had been defined in the production recipe. For this purpose the Workflow Choreographer asks for the appropriate service provider from the Orchestration System and if there is one registered in the Service Registry the plan execution can continue. Unfortunately, it is also possible that the necessary service is not available or there is no service in the system at all that the Workflow Choreographer asked for. In this case, the plan execution is aborted, and all users get notified through logging the error. After execution, each provider must call back to the Workflow Choreographer that the step execution is done. This enables the Workflow Choreographer to maintain its internal information about each running plan and run the following steps or finish the execution of a plan with no more steps to run. The workstation movement A $\rightarrow$ C $\rightarrow$ B $\rightarrow$

A → B can be followed by the sequence diagram shown in Figure 6.1 and by the related illustrative process collage in Figure 6.2.

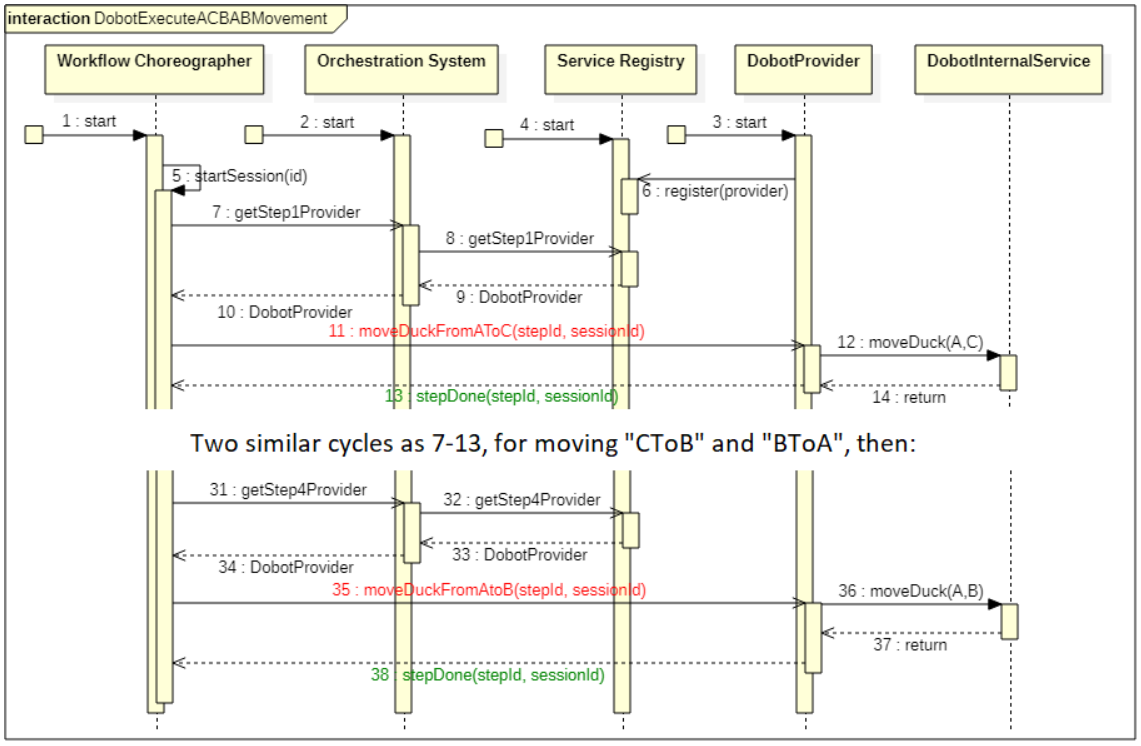

Figure 6.1: The sequence diagram of use case 1.

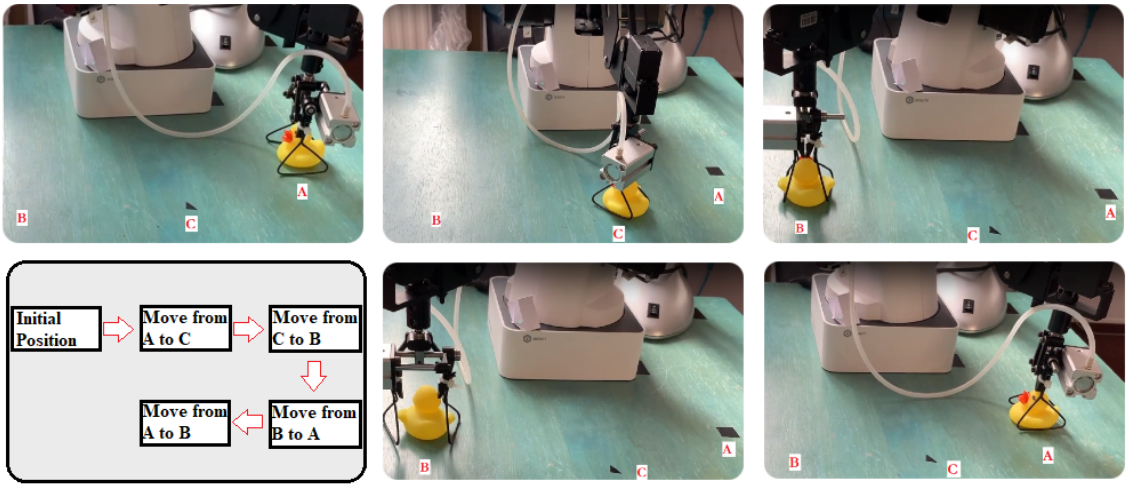

Figure 6.2: Picture collage showing plan execution in the first use-case.

6.2 Use case 2 - Shuffle the objects

In this case, there are two objects on workstation 'A' and 'C' and based on the production recipe, the task is to switch the objects where the steps are as follows:

1. Object1: A → B;
2. Object2: C → B;
3. Object1: B → C.

The sequence diagram of Use Case 2 is shown in Figure 6.3, and the related illustrative process collage can be seen in Figure 6.4.

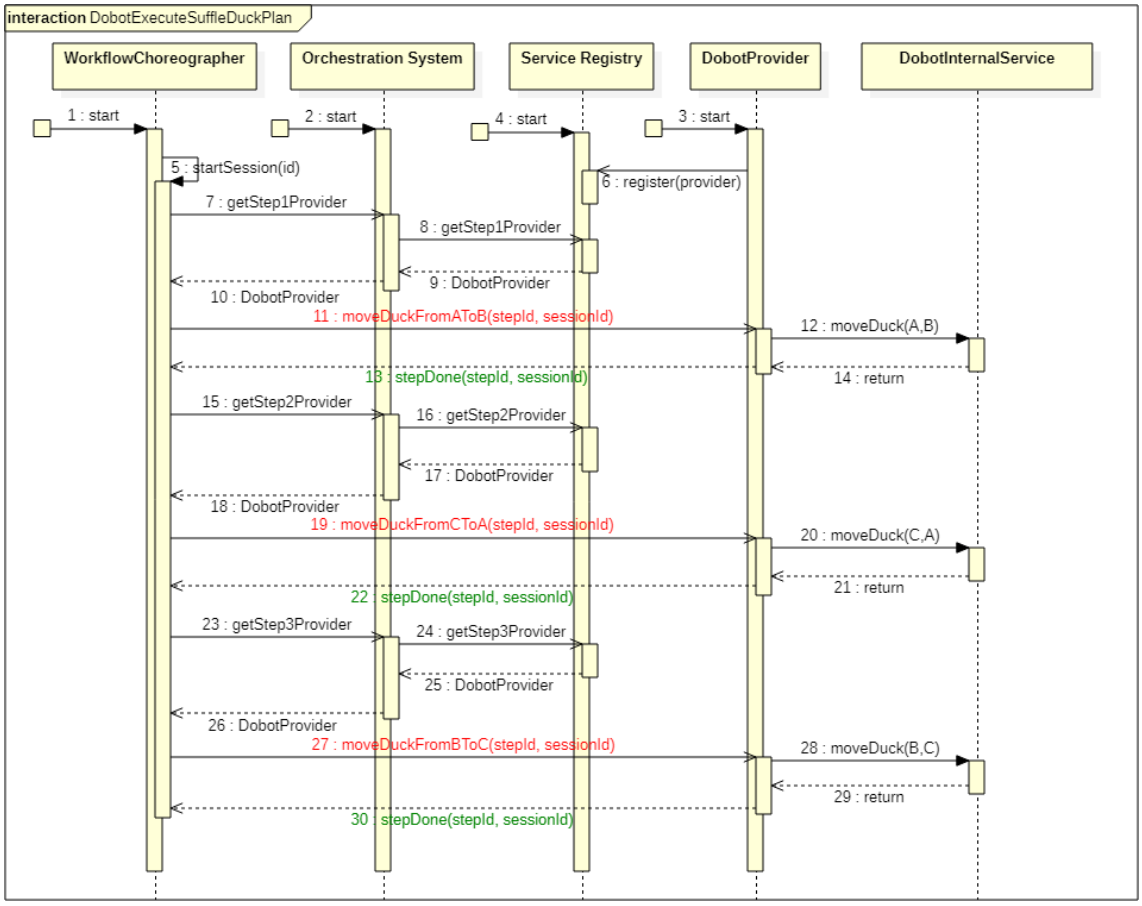

Figure 6.3: The sequence diagram of use case 2.

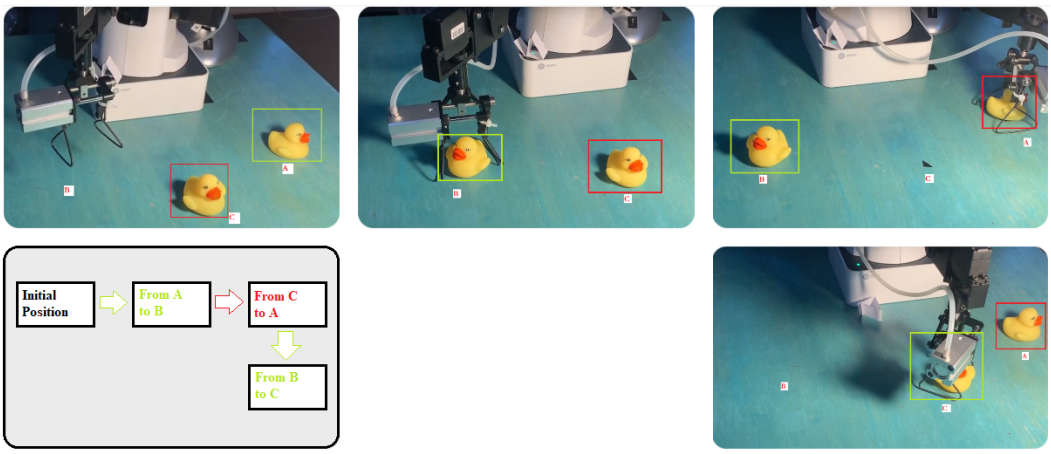

Figure 6.4: Picture collage showing plan execution in the second use-case.

Chapter 7

A Complex Use-case Scenario

7.1 Overview

We wanted to make a scenario that has close resemblance in complexity to a real production line. We gathered as many - and various - robotic arms, used a custom IoT conveyor belt (designed by our peer colleague, Balázs Riskutia), made them all Arrowhead Framework compatible and added our image processing system to complete the demonstration which can be seen on Figure 7.1.

Two small robots – one of which we can disable to demonstrate the dropout detection and dynamic orchestration features of the Arrowhead Framework – place rubber ducks (as production objects) on the conveyor belts. The belt drives the duck into the reach of the Dobot arm. The image processing system recognizes the duck and returns its position. The Dobot arm then lifts the duck off the conveyor belt. Each step is controlled by the Choreographer and Executor supporting systems.

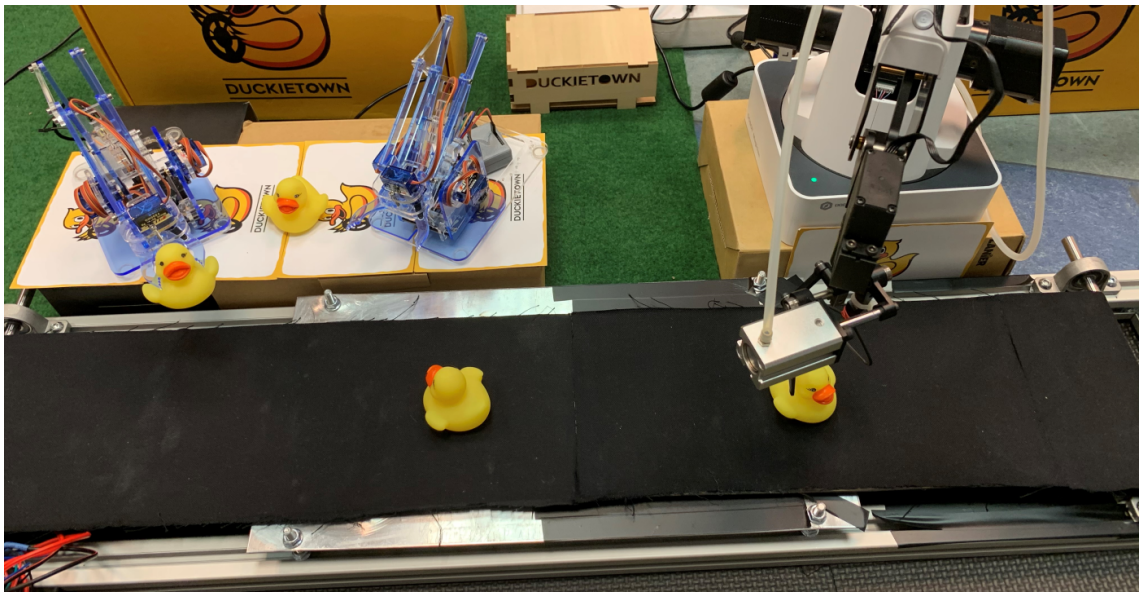

Figure 7.1: The devices taking part in the demonstation.

7.2 The elements of the dynamic use-case process

7.2.1 Image processing system

In the complex use-case scenario, we used the video and image processing system described in details in Section 5. The devices used to provide video feed were a Raspberry Pi 4 model B, with the official Raspberry Pi Camera v2 camera module connected to it. Integrating the system into the Arrowhead framework was done using both the Python and Java programming languages, as the device has a Python API for convenient camera usage and the Arrowhead Framework compatible provider was created using Java language. This is explained in more detail in Section 5. For processing the video feed provided by the previously described system, stronger hardware was required, as the Raspberry Pi 4 model B with 3 GB RAM was not capable of running the video processing algorithm. The video process algorithm was based on machine learning and neural networks, using TensorFlow 2 [17], which supports GPU acceleration for better performance.

7.2.2 Robotic Arms

For the project we used two kinds of robotic arms. One of which is the Dobot Magician that is seen on Figure 7.2(a), the other is the MeArm DIY robot on Figure 7.2(b), of which we had 2 sets.

7.2.2.1 Dobot Magician robotic arm

Its claimed accuracy is 0.2 mm. It has 4 axes of motion, and can reach for up to 32cm-s away from its base in 180 degrees [27]. To make it compatible with the Arrowhead Framework we had to implement a translator to communicate using RESTful communication methods over the network. The Dobot Magician can be controlled trough its Python API, so we had to create a program that connects each control method of the robot to a REST endpoint. This program is capable of executing universal commands received from the Excecutor, which commands are in the workflow plan constructed by the Choreographer system of the Arrowhead Framework. Although the arm has a "gripper", it is not suitable for our use-case. It had to be equipped with a custom clamp (Figure 7.3) in order for it to be able to lift the rubber ducks.

7.2.2.2 MeArm DIY robots

For simpler tasks we use two MeArm DIY robots, which can be bought, or 3D printed using open source blueprints, available trought their official website [28]. They are much smaller, a lot less accurate, but are capable of executing less delicate motions. They also have 4 axes of motion, but can only reach out for about 10 centimeters. Controlling of the robot is done with Arduino libraries on an ESP 8266 board, which can connect to the

internet trough Wi-Fi and can communicate using RESTapi trough https, similarly to the Dobot's translator program. Fortunately the default clamps can pick up the rubber ducks without much of a hassle but given the small size of the robot, usage is limited.

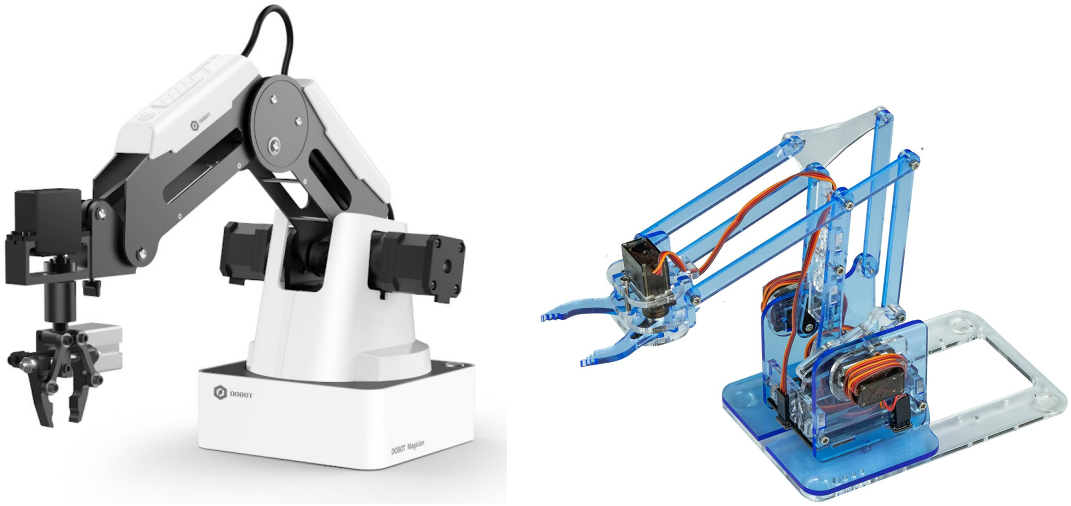

Figure 7.2: (a)Left: Dobot Magician with the "gripper" attachment [29]
(b)Right: MeArm DIY robotic arm [30]

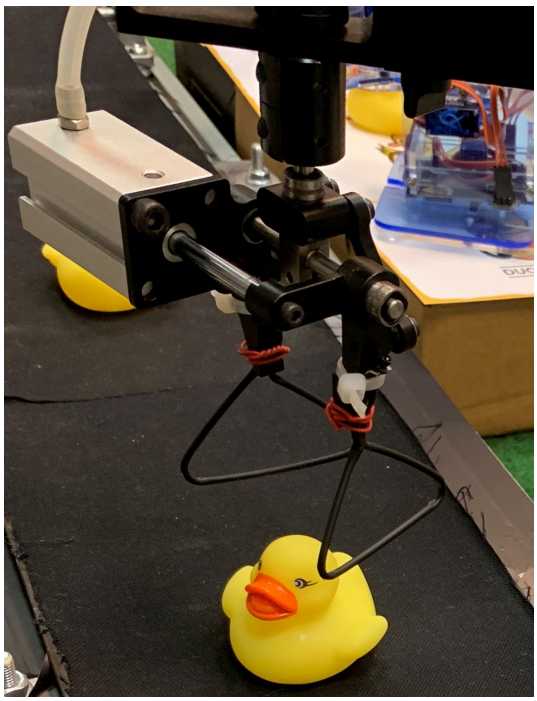

Figure 7.3: The custom clamp that allows the robot to grab rubber ducks

7.2.3 Conveyor Belt

For the complex use-case scenario, we thought that it is necessary to have a conveyor belt to demonstrate the workflow in an industrial environment. As the demonstration took place in a quite specific scenario and environment, we decided to create our own, hence our colleague, Balázs Riskutia has designed and manufactured one. To be able to execute precise movements with the conveyor belt, we used the popular NEMA-17 stepper motor which is widely used in 3D printers. For controlling the stepper motor and for Arrowhead Framework integrability, the microcontroller used in this case was the same as detailed in Section 7.2.2.2, the ESP 8266. After the physical construction of the conveyor belt we created a simple controller program with custom functionalities. These functionalities and the conveyor belt itself was integrated into the Arrowhead Framework using the device in an Arrowhead Framework compatible provider system.

7.3 The Arrowhead Local Cloud

In Section 4.1 we have presented the high-level, abstract architecture and components of an Arrowhead Local Cloud which can be seen on Figure 4.1. On Figure 7.4, a similar composition can be observed, but with exact components and Arrowhead Framework application systems present in the complex use-case scenario, during the demonstrations.

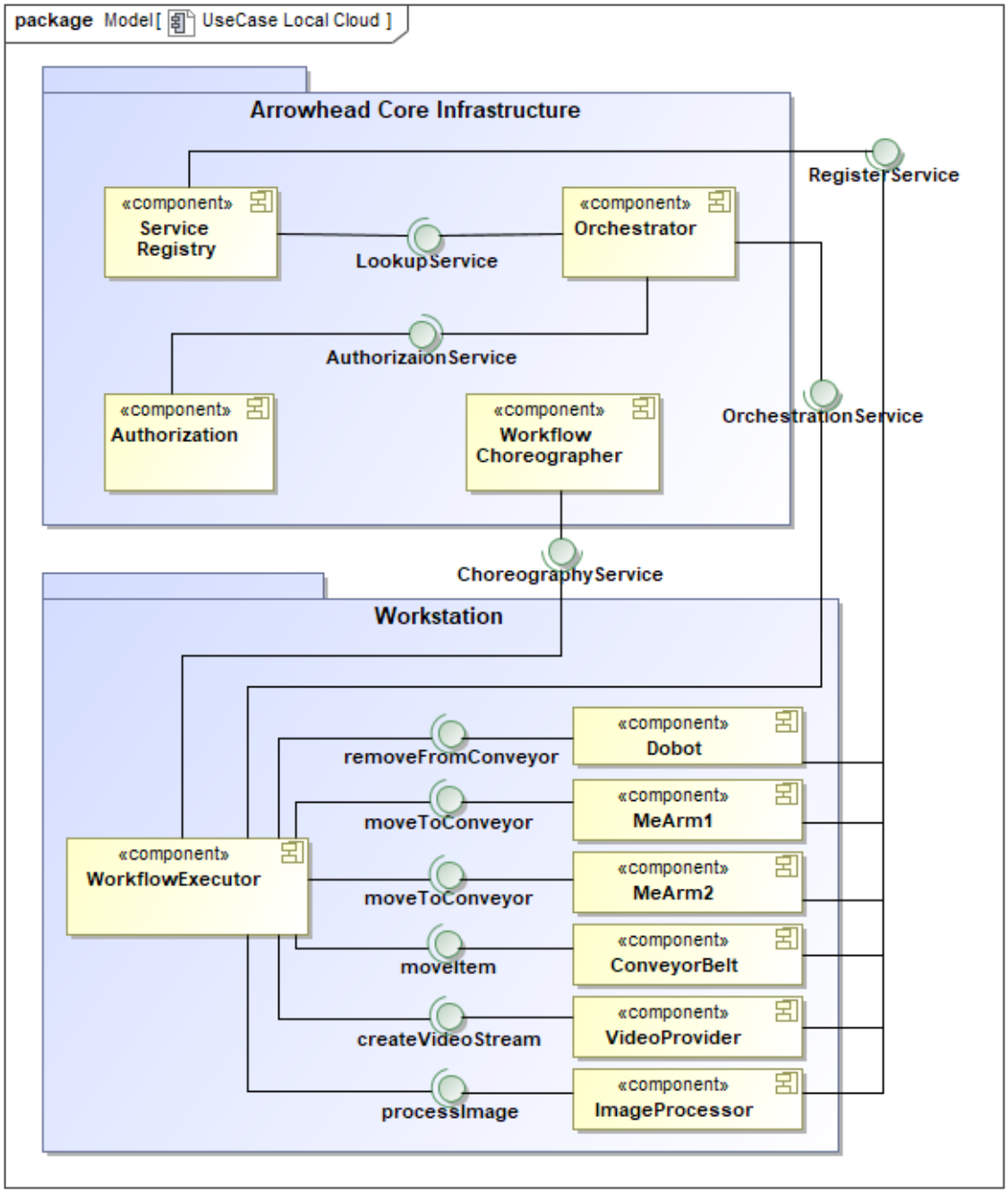

Figure 7.4: The Arrowhead Local Cloud Architecture with concrete devices of the complex use-case scenario.

7.4 Dynamic Process Sequences

For further understanding of the process, Figure 7.5 below shows a sequence diagram detailing the flow of plan execution. After all systems have booted up, the Workflow Choreographer starts the plan on a dedicated thread. The Choreographer only communicates directly with the new Workflow Executor system. As the WE receives each step it uses the Orchestration Service to find the corresponding providers in the cloud. The Orchestrator uses the lookup service of the Service Registry then returns a list of all providers that can produce the needed service for the task. The Orchestrator filters out the unnecessary services that do not meet the requirements defined in the orchestration rules and sends them to the Executor. The Executor chooses one provider from the list then sends the command to lift a duck onto the conveyor belt. While it is working another parallel task is carried out by the Choreographer. The same steps follow, however, since the MeArm1 is still in use MeArm2 gets returned as the chosen provider. After both steps are done, the Choreographer starts the execution of the next task. Similarly to the former steps, a provider is selected and returned to the WE. For this part of the production only the conveyor belt is suitable to move one of the ducks into the camera's field of view. To remove the rubber duck from the conveyor belt two services are required to work together. First the Image Processor unit has to determine the correct position of the rubber duck and then forward the information to the Dobot robotic arm. This is where the multiple service consumption and precondition support described in Section 4.2 come in handy. The image processing can serve as a precondition to the rubber duck movement this way the Dobot knows exactly where the item that is to be moved off from the conveyor belt is. The last two steps are repeated once more to remove the second duck from the conveyor belt to finish the execution of the recipe.

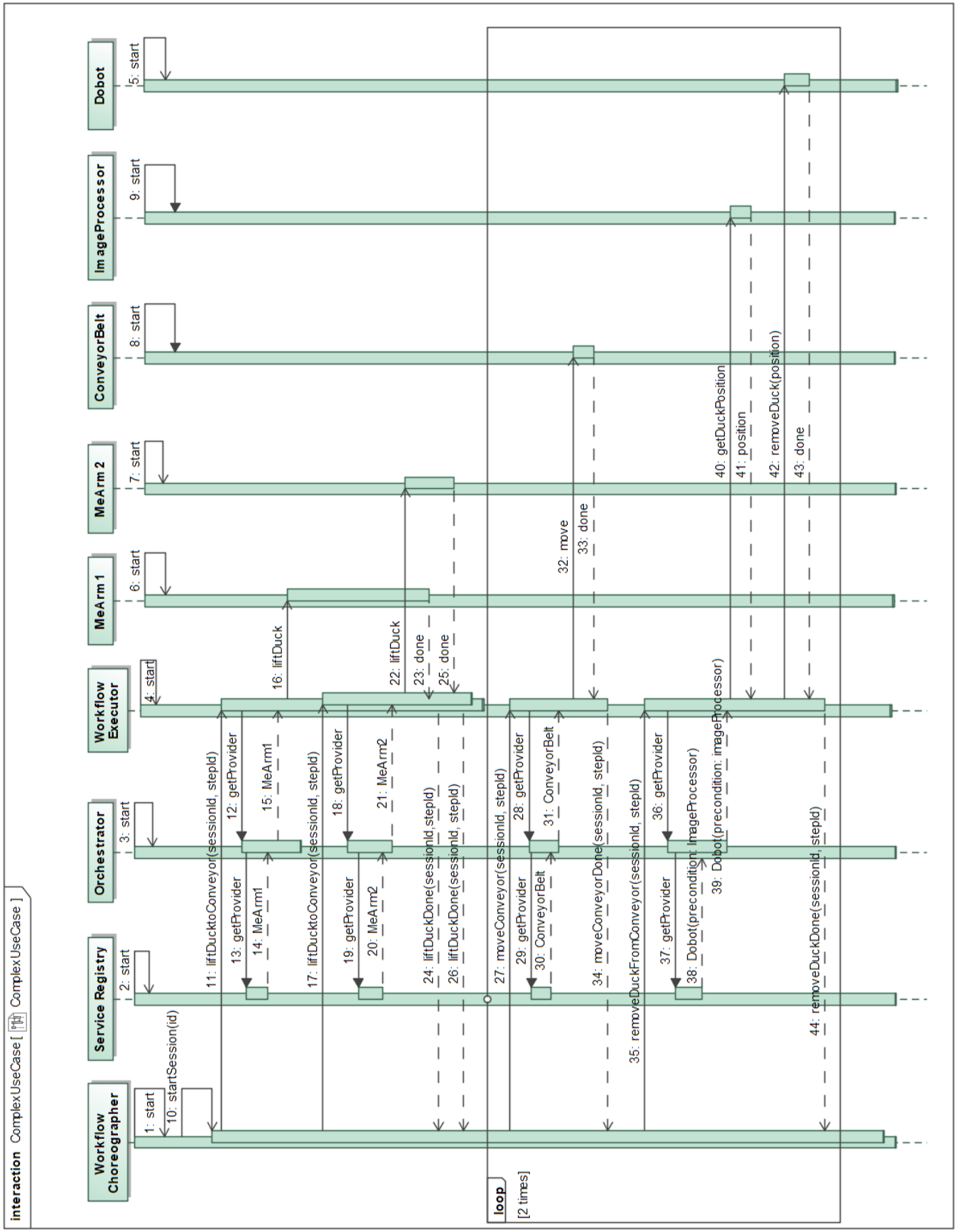

Figure 7.5: Sequence diagram showing this complex use case process.

Chapter 8

Measurement results

8.1 Computer Vision Measurements

Measuring the accuracy of the trained neural network was mainly done by the built-in function of the TensorFlow 2 training tool. Before training, we had to provide both training and test samples to the training tool, so it could automatically measure the accuracy of the model. The total loss was shown every hundredth steps in the real-time command line logs, but a detailed and graphical view was also available using the Tensorboard. Computer vision measurements show the classification loss, localization loss, normalized total loss, regularization loss and total loss of the model, from the training to the final state of the model. These indicators give a good prediction of the goodness of the model and show whether the training was correct or faulty.

The measurements shown on Figure 8.1 are both showing a correct learning process. The random initialisation of the neural networks can be observed at the early phase, as one training started from a lower total loss, the other started from a higher total loss, but both showed a constant, steady decrease in total loss while getting more and more trained.

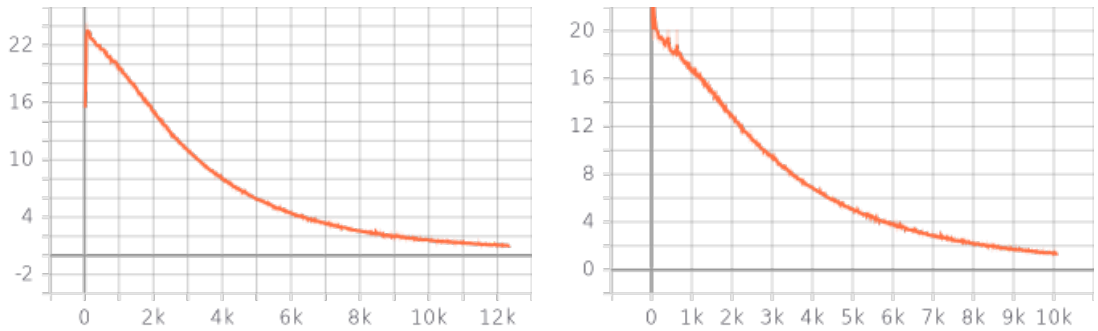

Figure 8.1: Graphs of total loss. On the left, it starts around 16, increases, then decreases. On the right, it starts from over 20 and descends steadily.

Further measurements of the model was done using the same method, but with a more natural video feed, on which the accuracy was much lower than on the test dataset. This

was the expected outcome of the measurement, as the training and testing dataset only contained few hundred images. A dataset of this scale is sufficient for training a neural network for specific environments, so the accuracy of the model was acceptable during the complex use-case scenario, but this model is not capable of providing a reliable object detection in a natural environment.

During the training process, different indicators are present and are measurable. One is the learning rate, which basically defines the 'braveness' of the neural network. It is a parameter that might have changing value as seen on Figure 8.2(a), to adapt to the state of the neural network over time. Total loss of a neural network can be observed on Figure 8.2(b), which shows nicely how our model gets more accurate over time.

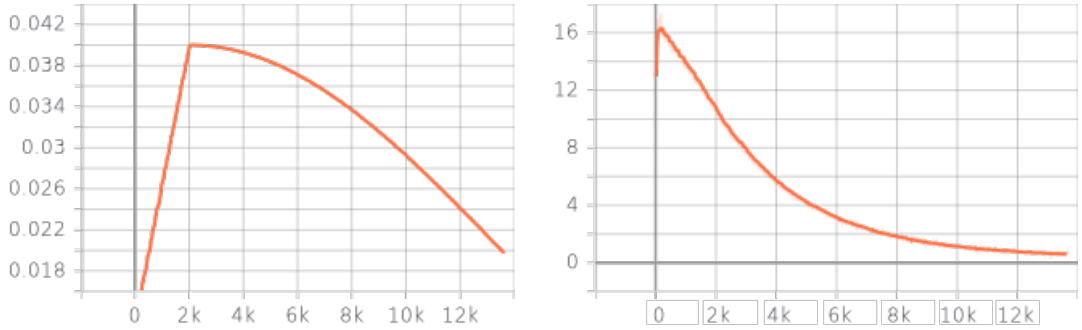

Figure 8.2: (a)Left: The learning rate of the neural network
(b)Right: Total loss of the neural network

Same tendency is observable on Figure 8.3, where specific losses, classification loss on Figure 8.3(a) and localisation loss on Figure 8.3(b) is present, showing that the model has a better localisation accuracy, but the average mean deviation is lower in classification, as the lighter colour shows.

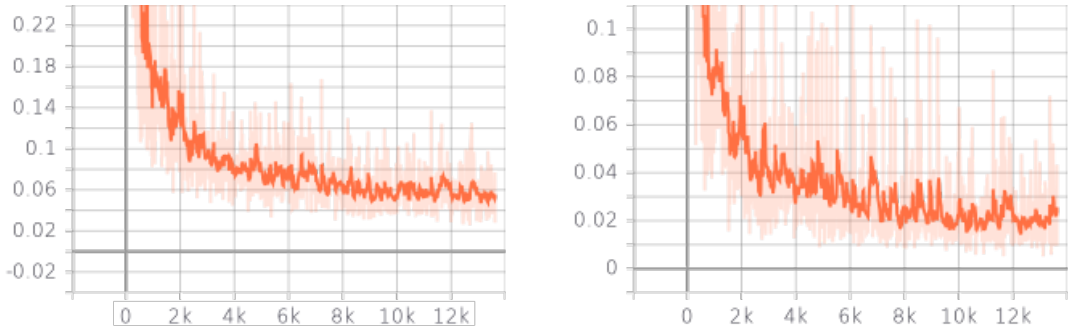

Figure 8.3: (a)Left: The classification loss of the neural network
(b)Right: The localisation loss of the neural network

8.2 System-level image processing Measurements

To be able to measure the efficiency of the image processing system, we created an implementation of the image processor which measures the time required for making a prediction for each frame of the incoming video feed. The results can be seen on Figure 8.4. The very first frame takes a little longer, because the system variables need to be set, but after a few hundred images, a stabilized performance is observable, processing one frame takes around 0.15 seconds, so the system can give six to seven new position data per second.

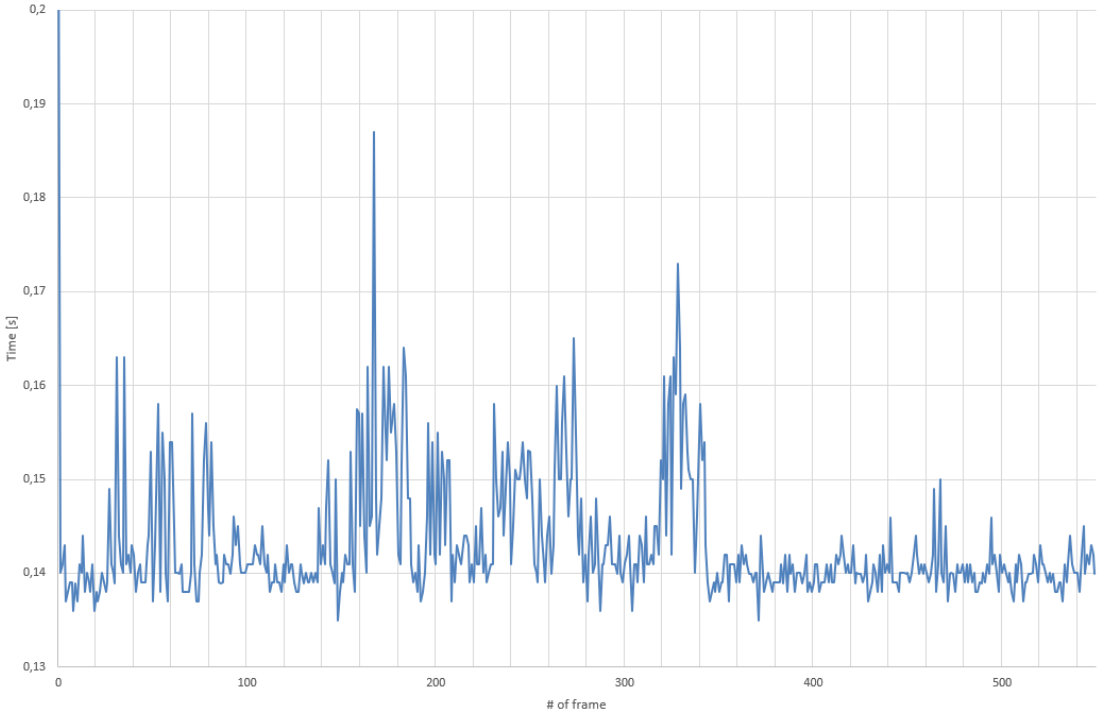

Figure 8.4: The measured values of the frame processing time.

Chapter 9

Conclusion

To frame what we have presented in this thesis, let us summarize the goals and results of our project. The main goal was to showcase a possible solution of improving industrial productivity and proving our approach with demonstrating a flexible workflow execution. This was achieved by exploiting the possibilities acquired by integrating the systems into the Arrowhead Framework.

The Arrowhead Framework implements service oriented architecture and applies the System of Systems concept. It is now also capable of meeting the requirements of real-time dynamic workflow management, as a result of our further development regarding to the Workflow Choreographer and the introduction of the Workflow Executors. The former is a supporting system developed further in charge of resource handling and task scheduling, which are essential capabilities for an intelligent, self organising system, the latter is a special application system governed by the Choreographer forcing providers to execute the tasks on hand. We have designed and realised a complex case study emulating the behaviour of an intelligent production line and integrated it into the Arrowhead Framework. This production line includes a deep learning based computer vision implementation, a custom made intelligent conveyor belt and multiple robotic arms controlled dynamically by the Arrowhead Framework using the data acquired from the computer vision system.

Our solution, despite its physical limitations, has successfully utilized and demonstrated the efficiency and applicability of dynamic workflow management, taking advantage of the opportunities provided by the Arrowhead Framework. Further development of this complex system is possible as well as adding new elements to it using the options provided by the Arrowhead Framework.

As for what can be expected in the near future, we hope that the potential of dynamic workflow management systems will be utilized as they provide a solution for factories to achieve cost-efficiency, eco-friendliness and automated operation, which are in favor of us, the people, and also in favor of a brighter and greener future.

Acknowledgements

Thank you Márton Juhász for providing us the Vorto Information Models and Function blocks making it possible to demonstrate the advantages of using Eclipse Vorto.

We also thank Balázs Riskutia in the development of the MeArm robots and the smart conveyor belt.

Bibliography

[1] Dániel Kozma, Pál Varga, and Felix Larrinaga. Dynamic Multilevel Workflow Management Concept for Industrial IoT Systems. *IEEE Transactions on Automation Science and Engineering*, 2020.

[2] Cristina Paniagua and Jerker Delsing. Industrial Frameworks for Internet of Things: A Survey. *IEEE Systems Journal*, 2020.

[3] Pál Varga, Luis Lino Ferreira, Jens Eliasson, and Fredrik Blomstedt. Making system of systems interoperable – The core components of the arrowhead framework. *Journal of Network and Computer Applications*, 81, 2016.

[4] Jerker Delsing. *Iot automation: Arrowhead framework*. CRC Press, 2017.

[5] Attila Frankó, Gergely Vida, and Pál Varga. Reliable Identification Schemes for Asset and Production Tracking in Industry 4.0. *Sensors*, 20, 2020.

[6] Dániel Kozma, Pál Varga, and Felix Larrinaga. Data-driven Workflow Management by utilising BPMN and CPN in IIoT Systems with the Arrowhead Framework. *IEEE 24th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 09 2019.

[7] Pál Varga, Dániel Kozma, and Csaba Hegedüs. Data-Driven Workflow Execution in Service Oriented IoT Architectures. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 203–210. IEEE, 2018.

[8] Wikipedia contributors. Neuron. `https://simple.wikipedia.org/w/index.php?title=Neuron&oldid=6877839`. (Accessed: 2020. 10. 25.).

[9] Loss Functions. `https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy`, 2017. (Accessed: 2020. 10. 25.).

[10] Saugat Bhattarai. An Introduction to Gradient Descent. `https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/`. (Accessed: 2020. 10. 25.).

[11] Chi-Feng Wang. The Vanishing Gradient Problem. `https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484`. (Accessed: 2020. 10. 25.).

[12] Tensorflow 2.0: Deep Learning and Artificial Intelligence. https://www.udemy.com/course/deep-learning-tensorflow-2/. (Accessed: 2020. 10. 25.).

[13] Convolutional Neural Network Architecture: Forging Pathways to the Future. https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-architecture-forging-pathways-future/. (Accessed: 2020. 10. 25.).

[14] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, 2017.

[15] Python. https://www.python.org/. (Accessed: 2020. 10. 25.).

[16] OpenCV-Python. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_table_of_contents_imgproc/py_table_of_contents_imgproc.html#py-table-of-content-imgproc. (Accessed: 2020. 10. 25.).

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[18] Francois Chollet et al. Keras. https://github.com/fchollet/keras. (Accessed: 2020. 10. 25.).

[19] Dániel Kozma, Pál Varga, and Kristóf Szabó. Achieving Flexible Digital Production with the Arrowhead Workflow Choreographer. *The 46th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 10 2020.

[20] MagicDraw software overview. https://www.nomagic.com/products/magicdraw. (Accessed: 2020. 10. 27.).

[21] Eclipse Papyrus. https://www.eclipse.org/papyrus/. (Accessed: 2020. 10. 27.

[22] Eclipse Vorto. https://www.eclipse.org/vorto/. (Accessed: 2020. 10. 25.).

[23] https://github.com/tombstone, https://github.com/srjoglekar246, https://github.com/pkulzc, and https://github.com/khanhlvg. Tensorflow 2 detection model zoo. https://github.com/tensorflow/models/blob/master/

research/object_detection/g3doc/tf2_detection_zoo.md, 2017. (Accessed: 2020. 10. 23.) commit: 23e26542c4f444e913372cf52ee7ac82885dae00.

[24] Tzutalin. Labelimg. https://github.com/tzutalin/labelImg, 2015. (Accessed: 2020. 10. 23.).

[25] Yang W. *Dobot Magician User Manual*. Shenzhen Yuejiang Technology Co. Ltd, 2016.

[26] Kristóf Szabó, Tamás Mrázik, Dániel Kozma, and Pál Varga. Workflow choreographer use-cases (1-3). https://www.youtube.com/playlist?list=PLXiXEGpr0Zwd5X8Tt9XXdUSQD2yF-Cqi, 2020. (Accessed: 2020. 10. 28).

[27] Dobot Magician Specifications. https://www.dobot.cc/dobot-magician/specification.html. (Accessed: 2020. 10. 24.).

[28] MeArm official website. http://learn.mearm.com/. (Accessed: 2020. 10. 27.).

[29] Dobot Magician stock photo. https://www.digitalissuli.hu/dobot-magician-basic-robotkar-521. (Accessed: 2020. 10. 24.).

[30] MeArm stock photo. https://www.amazon.com/MeArm-Robo6tic-Arm-Maker-Blue/dp/B01680T1B4. (Accessed: 2020. 10. 24.).