



SAPIENZA
UNIVERSITÀ DI ROMA

Digital Twins Composition via Markov Decision Processes

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Luciana Silo
ID number 1586010

Thesis Advisor

Prof. Giuseppe De Giacomo

Co-Advisor

Prof. Francesco Leotta

Academic Year 2020/2021

Thesis defended on 23th July 2021
in front of a Board of Examiners composed by:

Prof. Daniele Nardi (chairman)

Prof. Roberto Capobianco

Prof. Luigi Cinque

Prof. Marco Console

Prof. Giuseppe De Giacomo

Prof. Luca Iocchi

Prof. Simone Scardapane

Prof. Marco Schaerf

Digital Twins Composition via Markov Decision Processes

Master's thesis. Sapienza – University of Rome

© 2021 Luciana Silo. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: silo.1586010@studenti.uniroma1.it

Abstract

The use of Digital Twins is key in Industry 4.0, in the Industrial Internet of Things, engineering, and manufacturing business space. For this reason, they are becoming of particular interest for different fields in Artificial Intelligence (AI) and Computer Science (CS). In this thesis we focus on the orchestration of Digital Twins. We manage this orchestration using MDP, given a specification of the behavior of the target service, to build a controller, known as an orchestrator, that uses existing stochastic services to satisfy the requirements of the target service. The solution to this MDP induces an orchestrator that coincides with the exact solution if a composition exists. Otherwise it provides an approximate solution that maximizes the expected discounted sum of values of user requests that can be serviced. We formalize stochastic service composition and we present a proof-of-concept implementation, and we discuss a case study in an Industry 4.0 scenario.

*To my Brother Emanuele, that is much more than that.
To my Mother Annunziata, the light of my days.
To all my family.*

*It is not so much what we do, but how much love we put into doing it.
It is not so much what we give, but how much love we put into giving
Mother Teresa of Calcutta*

Acknowledgments

At the end of this exciting university course I would like to acknowledge all the people who have helped me during these years and during the thesis. First of all, I am sincerely grateful to the advisor of the thesis Prof. Giuseppe De Giacomo, for all his availability, for his patience and for giving me the opportunity to develop such an interesting project. Then, I also want to thank my co-advisor Prof. Francesco Leotta for his helpfulness and for his precious advice, and Prof. Massimo Mecella, for having made available not only his advices but also all his experience. A big thank also goes to PhD student Marco Favorito for having made a huge contribution to the development of the thesis and for all the support during these months. Thank you, I have learned a lot from each of you.

Then, I want to acknowledge all my family. My mother Annunziata, thank you for all the values transmitted and for never having stopped believing in me. My father Vincenzo thank you for giving me the tenacity and strength to never give up. My brother Gianpiero, thank you for being close to me during the university period and for making me aunt of Nicolò, my joy. All my gratitude goes to my brother Emanuele, I have no words to express everything you have always done for me: a friend, a brother, a father, you are everything to me a giant thank you for all the economic, psychological support and for being my first supporter, this success is also a bit yours.

I also want to acknowledge all my friends. Silvia, simply thank you for being a sincere friend, for rejoicing in my successes and for comforting me from my sorrows. People like you are rare, you deserve everything beautiful. Aurora, the person closest to me, thank you for your sweetness and above all, for all the laughs you managed to tear me away in difficult moments. You are special. Matteo, thank you for putting up with us and for all the outings together. Lorena, Martina, and Serena, thank you for all the carefree moments you gave me.

Last but not least, thanks to my friends and university colleagues. First, I would like to thank Claudia: thank you for sharing with me all the anxieties of our university career, of having been not only a colleague but a friend; your generosity was precious. I hope there will be many beautiful moments waiting for us. Carlo, thank you for being the shoulder to rely on and thank you for all the treats offered between breaks. Abhishek, thank you for sharing days of study with me, for never leaving me alone and for never having stopped asking me how I was doing. I carry each of you in my heart.

Finally, for the first time in my life, a big thank you goes to myself. There have been many difficult moments in recent years, but this has never made me lose the desire to fight and reach my goals. Thanks to me and my sensitivity that made me fragile but at the same time also strong.

*With love,
Luciana Silo*

Contents

1	Introduction	9
1.1	Background	9
1.1.1	Digital Twins	9
1.1.2	Service Composition	10
1.2	Thesis Motivations	10
1.3	Thesis Objectives	11
1.4	Thesis Results	11
1.5	Structure of the rest of the thesis	12
2	Industry 4.0	14
2.1	The evolution from Industry 1.0 to Industry 4.0	14
2.2	Technologies for Industry 4.0	15
2.2.1	Internet of Things and related technologies	15
2.2.2	Cloud computing	16
2.2.3	Cyber-physical systems	17
2.2.4	Industrial integration, enterprise architecture and enterprise application integration	18
2.3	Smart manufacturing systems for Industry 4.0	19
3	Digital Twins	22
3.1	DT literature	22
3.2	Enabling technologies	25
3.3	Digital Twins in Industry 4.0	26
3.4	Digital Twin platforms	28
3.4.1	Bosch IoT Things	29
3.4.2	Overview	29
3.4.3	Example	30
3.4.4	Implementation and Deployment	32

<i>CONTENTS</i>	7
3.4.5 Communication	32
4 Framework	36
4.1 Introduction	36
4.2 Architectural Model	37
4.3 Example	40
4.4 Use Case	41
4.4.1 Scenario: Ceramics production	41
4.4.2 Discussion	42
4.5 Technological Aspects	42
5 MDP and Stochastic Service Composition	44
5.1 Definition	44
5.2 Techniques for finding an optimal policy	46
5.3 Service Composition	48
5.4 Stochastic Service Composition	50
6 Service Composition with Stochastic Services	53
6.1 Introduction	53
6.2 Computing an Optimal Orchestrator	57
6.3 Examples	58
6.3.1 Ceramics Production	58
7 Solver	63
7.1 The <code>stochastic-service-composition</code> library	63
7.2 Implementation of Services and Targets	63
7.3 Build a System Service	71
7.3.1 Description	71
7.3.2 Algorithm	72
7.4 Build the Composition MDP	72
7.4.1 Description	72
7.4.2 Algorithm	73
7.5 Optimal Policy	75

<i>CONTENTS</i>	8
8 Technological Solution based on Digital Twins	80
8.1 High-level Architecture	80
8.1.1 Overview	80
8.1.2 MQTT	81
8.1.3 WebSocket	82
8.1.4 ThingAPI	82
8.2 Services as Digital Twins	83
8.3 Target as Digital Twin	86
8.4 Workflow and Communication Details	87
8.4.1 Target and Orchestrator	87
8.4.2 Orchestrator and Services	88
8.4.3 Target, Orchestrator and Services	90
8.4.4 Special case	91
9 Proof-of-Concept Implementation	92
9.1 Bosch IoT Things DT Implementation	92
9.2 Main	107
9.3 Launch Devices	110
9.4 Execution	110
10 Conclusion and Future Works	113
10.1 Overview	113
10.2 Remarks	113
10.3 Future works	114
Bibliography	116

Chapter 1

Introduction

This chapter presents the outline of this thesis and summarizes motivations, objectives, and results. The chapter is structured as follows:

- In **Section 1.1**: we provide a background in which the thesis developed, illustrated the main concept of Digital Twins and Service Composition;
- In **Section 1.2**: we describe all the motivation of the development of this thesis;
- In **Section 1.3**: we define all the objectives of the thesis;
- In **Section 1.4**: we clarify all the result achieved by the objectives set of the thesis;
- In **Section 1.5**: we list and explain the structure of the rest of the thesis.

1.1 Background

This chapter shows the background, i.e. the main concepts on which the development of the thesis is based.

1.1.1 Digital Twins

The continuous evolution of technologies in the fields of communication, networking, storage and computing, applied to the more traditional world of industrial automation, in order to increase productivity and quality, to ease workers' lives, and to define new business opportunities, has created the so-called smart manufacturing or Industry 4.0.

Digital Twins (DTs) are up-to-date digital descriptions of physical objects and their operating status. Modern information systems and industrial machines may natively come out with their digital twin; in other cases especially when the approach is applied to already established factories and production processes, digital twins are obtained by wrapping actors that are already in place.

DTs are commonly known as a key enabler for the digital transformation in manufacturing. Even though there is no common understanding concerning this term, different digital twin definitions agree on features such as:

- **connectivity**: i.e., the ability to communicate with other entities and DTs,
- **autonomy**: i.e., the possibility for the DT to live independently from other entities,
- **homogeneity**: i.e., the capability, strictly connected to the autonomy, that allows to use the same DT regardless of the specific production environment,

- **easiness of customization:** i.e., the possibility to modify the behavior of a physical entity by using the functionalities exposed by its DT,
- **traceability:** i.e., the fact that a DT leaves traces of the activity of the physical entity.

The availability of DTs could therefore have a huge impact on the design of manufacturing processes in digital factories, by allowing automatic recovery and optimization, and even automatic orchestration of the intermediate steps for achieving a production goal, thus achieving manufacturing resilience.

1.1.2 Service Composition

During the last years, many approaches have been proposed in order to address the issue of automated service composition. The composition of services, amounts to realize a (virtual) target service, by resorting only to (actual) available services. First, for sake of semplicity we can consider Roman Model, in which services are abstracted as transition systems and the objective is to obtain a composite service that preserves a desired interaction, expressed as a (virtual) target service.

Then, we discuss and elaborate upon a probabilistic model for the service composition problem, we find an optimal solution by solving an appropriate probabilistic planning problem (a Markov decision process – MDP) derived from the services and requirement specifications. Specifically, it is natural to make the requirement probabilistic, associating a probability with each action choice in each state (Brafman et al., 2017).

This probability captures how likely the user is to request the action in that state. Such information can be, initially, supplied by the designer, but can also be learned in the course of service operation in order to adapt the composition to user behavior. Next, a reward is associated with the requirement behavior. This reward can be defined in different ways depending on the designer's objectives.

1.2 Thesis Motivations

Recently, the concept of Digital Twin (DT), meant as a virtual replica of a physical asset, emerged. Due to the growth of the Internet-of-Things (IoT), which is the main enabling technology for DTs, an increasing number of software solutions that implement DTs appeared. The Digital Twin is an ideal tool to accomplish the purpose of Industry 4.0, since it enables massive exchange of data that can be interpreted by analytical tools, in order to improve decision making.

In fact, as showed in recent literature, Digital Twins can be potentially employed in every phase of manufacturing: from the very beginning, in the design phase, till the product is operative, in the maintenance phase. Nevertheless, the potential of DTs as a tool to fully realize the paradigm of smart manufacturing has been only superficially explored. In the traditional manufacturing process, the production plan is generated based on the new and historical orders. The preparation for production is carried out, such as equipment maintenance and material collection, and formal production is executed according to the plan. However, this approach is poorly maintainable and not resilient to changes of the execution environment.

The approach proposed in (Brafman et al., 2017), despite being an improvement from the classical Roman approach as it allows to find "good enough" solutions, does not have the expressivity to represent nondeterministic behaviours of the available services, and it does not take into account the rewards (or costs) of using a certain service. For example, in an Industry 4.0 scenario, different machines may have different performances or utilization costs, and reasoning about machines capabilities may be crucial to orchestrate them effectively. Moreover, there might be uncontrollable events, e.g. usury or breaking of the machines, that cannot be capture

by deterministic models. In other words, we need a way to represent *stochasticity* of services' state transitions and *rewards* associated to them, such that the orchestrator can opt for the best machine to take a job in a specific state of the target task to be accomplished.

1.3 Thesis Objectives

This thesis sets specific objectives about the orchestration of Digital Twins. Our first objective is to extend the theoretical work of service composition with stochastic settings introduced in (Brafman et al., 2017), proposing a model that permits not only to the target but also to the services to behave stochastically.

Then, what we want to construct is a library that captures the stochasticity of the available services and target, composes system service, composes MDP and calculates the optimal policy.

Another important objective, is to define the previous model with Digital Twins technologies capturing available services and target as Thing in Bosch IoT Things platform. What we want to propose is a software architecture for Smart Factories that, is able to orchestrate the devices: where given the target action (that represents user's choice) and available services (that represents machines) the orchestrator is able to choice which service can execute the action. We explain in detail how we managed the communication between Orchestrator, Bosch IoT Things and devices.

To validate the proposed architecture, we think about a hypothetical manufacturing process. Finally, we want to construct a proof-of-concept implementation of the mentioned frameworks about an Industry 4.0 use case.

1.4 Thesis Results

The thesis significantly contributes to the research areas of Artificial Intelligence and Computer Science. All the objectives stated above have been achieved. The first contribution of the thesis is extendend the work of (Brafman et al., 2017) about service composition, that assume that the target services behave stochastically. Extending this model allow us to capture stochastic services, where the service transitions are probabilistic too. The relevant changes are the definitions of transition functions associated to the available services. The MDP construction, too, need to be modified to take into account the stochastic transitions of both the system state and target state. We give motivations about this approach and analyzed the implications and the main advantages.

Then, we create a Python library ad-hoc, **stochastic-service-composition** where we implement all the components defined previously, this represent a crucial step for the calculation of MDP. In particular we define: the class of services, how to build its stochastically transitions and how build the system service; the class of target and how to build its stochastically transitions; the function of the composition MDP and the optimal policy calculation. This library is available at this github link <https://github.com/luusi/stochastic-service-composition> in the subfolder "stochastic_service_composition". Moreover, we implement also Jupyter notebooks which contain both code and automata of the available services, the system service, the target and the composition MDP. We provide this notebooks because they are very human-readable documents and also containing an analysis description and figures for better understand our case of study. This library is available at this github link <https://github.com/luusi/stochastic-service-composition> in the subfolder path "docs/notebooks"

We devise a way to represent the digital representation of available services and target, we doing this using Digital Twin created through Bosch IoT platform. Thanks to this representation we can simulate a sort of "real" simulation of available services and target be-

haviour that can happen in an Industry 4.0 scenario. For this purpose we define an orchestration between the user choice and the using of machines that can perform the action choose by the user. In this way when the target asks for an action, the orchestrator delegates to the right service this action. This part is available at this github link <https://github.com/luusi/stochastic-service-composition> in the subfolder “digital_twins”.

As proof of all that has been said we provide a simulation of process in an Industry 4.0 scenario that produces the ceramics. The target is linear and performs one action at time according to the optimal policy, the orchestrator delegate the action asked from target to the right machine that perfoms it and update the state in which it is, the orchestrator says to the target that all is going well and the target ask for to the next action.

1.5 Structure of the rest of the thesis

The rest of the thesis is structured as follows:

- In **Chapter 2**: we introduce the concept of Industry 4.0. We start illustrating the evolution from Industry 1.0 to Industry 4.0 and how over the years the industry has progressed; the technologies used for it as IoT, Cloud Computing, CPS and industrial integration, and finally we list and describe the smart manufacturing systems for Industry 4.0;
- In **Chapter 3**: we describe the notions of Digital Twins, providing the definitions that during the years have emerged. Then, we define enabling technologies as security, algorithms, software and platform etc., and we list all manufacturing processes that involves Digital Twins. Moreover, we give a detailed description of the platform used to create Digital Twins, Bosch IoT Things, and we illustrate all the benefits that the platform offers, the most important concepts and a short tutorial on its use;
- In **Chapter 4**: we illustrate the structure of the framework: the intermediate state between the concepts described till now and the software that we will use. We start with a small overview of the Digital Twins representation in Industry 4.0, we proceed with the architecture for a smart manufacturing process based on DTs and we provide an example of what described so far. Finally, we define the description of our use case, of how are managed the available services, the target and the orchestrator and we briefly underline the technological aspects of our use case;
- In **Chapter 5**: we present preliminary notions to understand technical content of the thesis. In particular, we describe the mathematical definition of MDP and all the main concepts; then, we report the techniques for finding an optimal policy, introducing policy iteration and value iteration, showing their algorithm in pseudocode. Moreover, we explain the problem of service composition starting from the simplest model, the Roman one, ending with stochastic service composition where the target is stochastic, introduced in (Brafman et al., 2017);
- In **Chapter 6**: we describe the first contribution of the thesis. We introduce the theoretical concepts of service composition with stochastic services, that is an extension of stochastic service composition. We illustrate all the mathematical concepts and definitions of this extension, the computation of optimal orchestrator and the relative proofs. Finally, we describe an example of how the definitions illustrated above can be applied;
- In **Chapter 7**: we present one of the practical contributions of the thesis, giving a small description about the library and tools used, and providing the GitHub link to the project developed. We continue by describing in detail the implementations of all the available services and the target, showing code and automata of both. We also define the construction of the system service algorithm and also of the composition MDP providing code,

automata and algorithm. Finally we show the optimal calculation with some interesting remarks;

- In **Chapter 8**: we describe the a technological solution of the model proposed in Chapter 6 based on DT in an Industry 4.0 scenario. First we present the architecture of the project, listing the main components between Bosch IoT Things and the system and how are managed the the connection between them, then we explain how the available services and the target are defined in the Bosch IoT Things platform. Moreover, we illustrate the workflow of our system and in particular how target, orchestrator and services communicate between them;
- In **Chapter 9**: we describe the proof-of-concept implementation of our thesis. In particular, we illustrate and explain how DT are implemented in Bosch IoT Things platform. Then, we describe the core structure of the project, presenting and explaining in detail the orchestrator's code and devices code, as well as the communication mechanisms. Finally, we describe the execution of the software and how to launch the different components;
- **Chapter 10**: summarizes the conclusions of the thesis, providing an overview, the most important remarks and the future works that can be done with this framework.

Chapter 2

Industry 4.0

In this chapter we explain the fundamental concepts of Industry 4.0: the ongoing automation of traditional manufacturing and industrial practices, using modern smart technology. The chapter is structured as follows:

- In **Section 2.1**: we talk about the evolution during the years of the industry from 1.0 to 4.0, highlighting progress;
- In **Section 2.2**: we propose the related technologies (starting from IoT and ending in Industrial Integration), describing them in detail and providing different examples; (Xu et al., 2018);
- In **Section 2.3**: we list the framework of Industry 4.0 in smart manufacturing systems (Zheng et al., 2018).

2.1 The evolution from Industry 1.0 to Industry 4.0

Industry 4.0 is a term coined to represent the fourth industrial revolution based on the latest technological advances. Over the years there has been a progression of industry as shown in Figure 2.1.

- **The First Industrial Revolution** began at the end of the eighteenth century and early nineteenth century, which was represented by the introduction of mechanical manufacturing systems utilising water and steam power.
- **The Second Industrial Revolution** started in the late nineteenth century, symbolised by mass production through the use of electrical energy.
- **The Third Industrial Revolution** began in the middle of twentieth century and introduced automation and microelectronic technology into manufacturing. In the Third Industrial Revolution, the advancement of Information and Communication Technologies (ICT) was at the core of every major shift of the manufacturing paradigm.

Industry 4.0 is mainly represented by: Cyber-Physical System (CPS), Internet of Things (IoT) and Cloud Computing (Jasperneite, 2012; Hermann et al., 2016) however, it will also rely on smart devices in addition to IoT, CPS, Cloud Computing and Business Process Management (BPM).

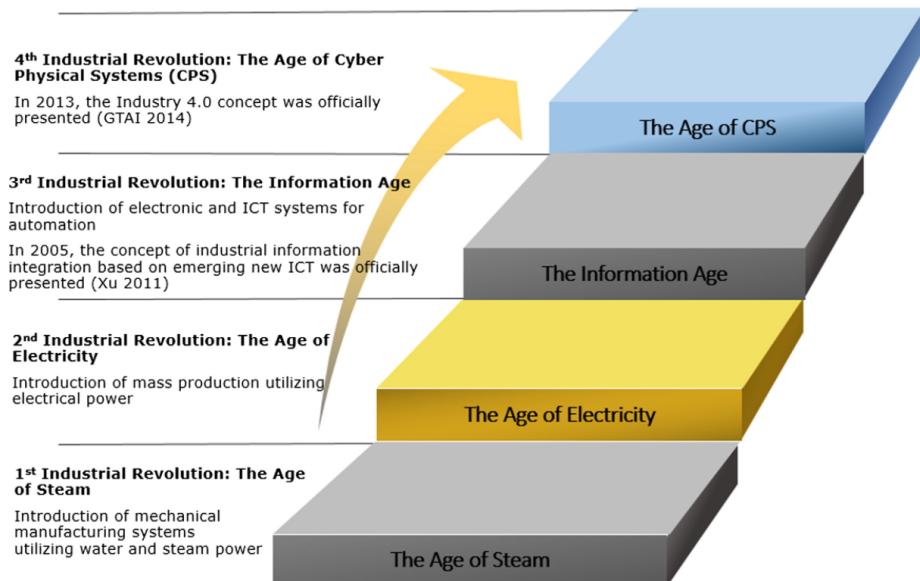


Figure 2.1. The evolution from Industry 1.0 to Industry 4.0

In a Industry 4.0 scenario, the manufacturing process is the main activity and, among several equipments, autonomous robots are extensively used toward manufacturing performance and revenue improvements. Combined with currently available techniques of data analysis and cognition, this creates new possibilities of interoperability, modularity, distributed processing, and integration in real time with other systems for industrial processes.

2.2 Technologies for Industry 4.0

Various technologies or techniques can be used for implementing Industry 4.0. These technologies include: CPS, IoT, cloud computing, blockchain, industrial information integration and other related technologies.

2.2.1 Internet of Things and related technologies

Internet of Things (IoT) is expected to offer promising transformational solutions for the operation and role of many existing industrial systems within the digital enterprises of tomorrow's complex industrial ecosystems. When the term, IoT first emerged, it was referred to uniquely identifiable interoperable connected objects using radio-frequency identification (RFID) technology (Ashton et al., 2009; Da Xu et al., 2014). Connecting RFID reader to the Internet, the readers can automatically and uniquely identify and track the objects attached with tags in real-time. Later on, the IoT technology was used with other technologies, such as sensors, actuators, the Global Positioning System (GPS) and mobile devices that are operated via Wi-Fi, Bluetooth ecc.

A recent definition of IoT is: a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual “Things” have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network.

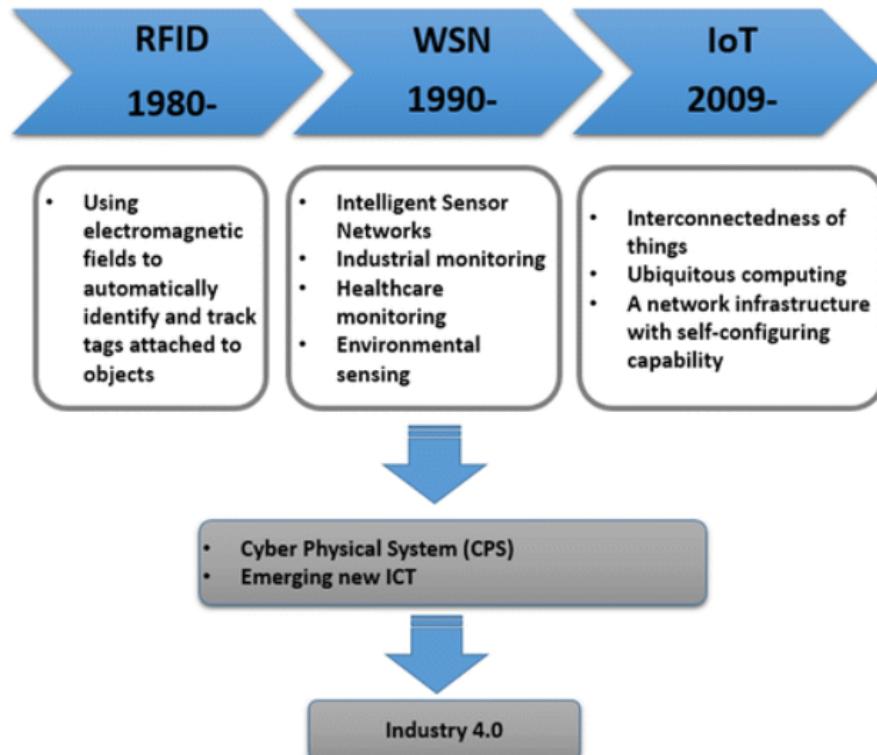


Figure 2.2. IoT related technologies made a significant impact on new ICT and paved the way for the realisation of Industry 4.0

The foundation of IoT can be considered as a global network infrastructure composed of numerous connected devices that rely on sensory, communication, networking and information processing technologies (Tan and Wang, 2010; Da Xu et al., 2014; Mao et al., 2016). Figure 2.2 presents advances in RFID, WSN and IoT. Today RFID, WSN and IoT are used to form a solid technological foundation for supporting CPS as well as the emerging new ICT.

Industry 4.0 combines intelligent sensors, artificial intelligence, and data analytics to optimise manufacturing in real time. With the advances in sensor network technologies, wireless communication, and other emerging technologies, more and more networked things, or smart objects, are being involved in IoT. Meanwhile, these IoT-related technologies have also made a significant impact on new ICT and CPS thus paved the way for the realisation of Industry 4.0.

2.2.2 Cloud computing

Cloud computing is a computing technology which offers high performance and low cost (Zheng et al., 2014; Mitra et al., 2017). Virtualization technology provides cloud computing with resource sharing, dynamic allocation, flexible extension, and numerous other advantages. Cloud-based manufacturing is a rising technology which can contribute significantly to the realisation of Industry 4.0 that enables modularization and service-orientation in the context of manufacturing, in which systems orchestration and sharing of services and components are important considerations (Wang et al., 2013; Thames and Schaefer, 2016; Moghaddam and Nof, 2018).

Cloud manufacturing, similar to cloud computing, uses a network of resources in a highly distributed way. Manufacturing-as-a-Service (MaaS) has been gaining attraction in the manufacturing industry. Cloud design allows anyone to upload and share designs with others. Local Motors, an American motor vehicle-manufacturing company, focused on low-volume production

of open-source motor vehicle designs using multiple micro-factories. Their designs are co-created by designers, engineers, fabricators and enthusiasts in its virtual community. This is an example of using a cloud design (Branger and Pang, 2015).

A modern enterprise's operation involves numerous decision-making activities, requiring a large amount of information and intensive computation. At one point, manufacturing enterprises required multiple computing resources such as servers for databases and decision-making units. This caused inefficient data exchange and sharing, low productivity and less optimal utilisation of manufacturing resources. Cloud computing provides an effective solution to such problems.

2.2.3 Cyber-physical systems

Cyber-physical systems (CPS) is the core foundation of Industry 4.0 (Varghese and Tandur, 2014; De Silva and De Silva, 2016; Kim, 2017). CPS are engineered systems that are built from, and depend upon the seamless integration of computational algorithms and physical components. Advances in CPS will enable capability, adaptability, scalability, resiliency, safety, security and usability that will far exceed the simple embedded systems of today as Figure 2.3 shows. According to the NSF, we have seen a convergence of CPS technologies and research thrusts that underpin the IoT and Smart & Connected Communities (SCC).

These domains offer new and exciting challenges for foundational research and provide opportunities for maturation at multiple time horizons. New smart CPS will drive innovations in sectors such as manufacturing, energy, transportation, agriculture, automation and healthcare. In Industry 4.0, CPS is expected to provide the basis for the creation of Industrial IoT, which combines with advanced ICT to make Industry 4.0 possible. CPS connects virtual space with physical reality by integrating computing, communication, and storage capabilities; furthermore, it can be real-time, efficient, reliable, and secure (Cheng et al., 2016).

CPS is considered to be an Industry 4.0 enabling technology that will merge the virtual and the physical worlds (Saldivar et al., 2015), making the boundaries between these two worlds disappear.

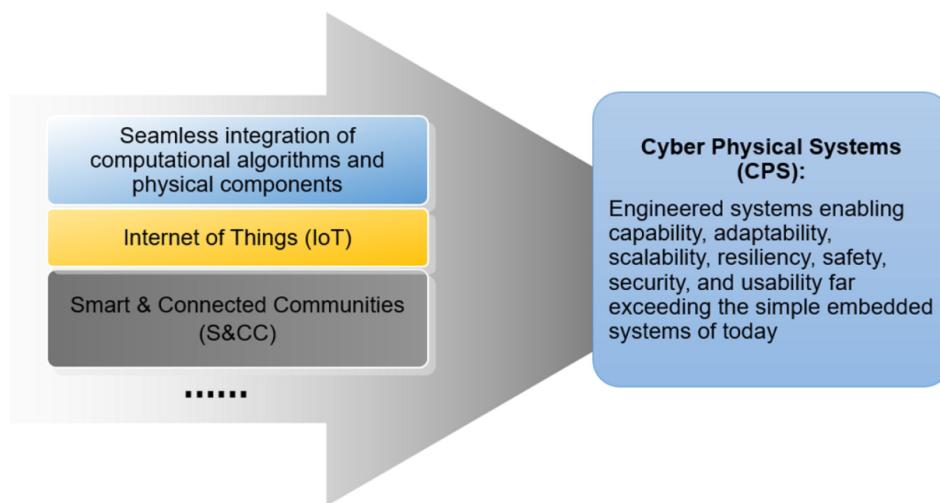


Figure 2.3. CPS

Industry 4.0 manufacturing systems will be collaborative systems involving various communicating agents including physical agents, software agents, and human agents. This will result in a fusion of both technical and business processes leading the way to a new industrial age, resulting in the smart factory. The essence of Industry 4.0 is applying CPS to realise smart factories (Kusiak, 2017).

In other words, the smart factory is made possible by CPS-based production systems. CPS can play a major role in smart factory manufacturing and production processes. This provides significant real-time, resource, and cost advantages in comparison with classic production system. In one application, for example, an automotive manufacturer employed cognitive technologies to optimise the configuration of its production line to balance the workload between stations, use labour more efficiently, and increase the rate of production while also adhering to its design for manufacturing (DFM) practices.

The application enabled the manufacturer to reduce operating costs and capital investments by about 10%. In another application, an automobile manufacturer used cognitive planning tools to optimise its use of available plant capacity to bring a new model of cars into production. The application enabled the manufacturer to reduce operating costs and capital investments by about 10%. In yet another application, a semiconductor manufacturer was able to reduce cycle time by 15% by reducing equipment idle wait times, thus increasing throughput and asset utilisation.

2.2.4 Industrial integration, enterprise architecture and enterprise application integration

In the first half of the first decade of 2000 it has become more and more clear that the emergence of Industrial Integration grew out from a new era of ICT, which occurred at the stage of the Third Industrial Revolution.

(Kaynak, 2007) wrote: “The area of industrial automation and control has had its share of the changes too”. It is easy to see how dominant IT has become in industrial electronics if one considers the changes in time spent by an engineer in designing a controlled drive system:

- **Before the 1960s:** 80% for designing a control system with mechanical switches.
- **After the 1960s:** 80% for designing power electronics converters.
- **After the 1980s:** 80% for designing digital hardware and software.
- **Currently:** 90% for software and IT.

In the current process of industrial integration, CPS represent a paradigm shift from existing business and market models, as revolutionary new applications, services and value chains will become available. Due to the arrival of Industry 4.0 and the profound changes to complex industrial ecosystems (Rennung et al., 2016), there is the need to embrace new architectures and new business processes that will help an industrial organisation with the adaptation of existing enterprise architecture, ICT infrastructures, processes and relationships to support the transformation. The Gartner Group describes the enterprise architecture (EA) change process as one that is creating, improving, and communicating the key requirements, principles and models that describe the enterprise’s future state and enable its evolution.

An enterprise architecture (EA) presents the structure of an enterprise and consists of the main enterprise components such as a company’s goals, organisational structures, information infrastructure and business process. Integration, consolidation and coordinated applications have been identified as a critical issue in the Industry 4.0 environment. The boundaries of individual factories will most likely fade away. Factories in different industrial sectors and different geographical regions will be interconnected or integrated. Most probably, an enterprise will have some existing legacy systems that it intends to continue to use, and meanwhile, it will add a new set of applications to the operation. To address the integration of new and existing applications, an ICT solution, which is referred to as Enterprise Application Integration (EAI) (Yu and Madiraju, 2014) can be applied.

In order to integrate the new CPS-based digital capabilities with existing architectures, systems and processes, the coordination of various systems and applications greatly depends on EA,

EI and EAI. Weber has pointed out that one of the important issues surrounding Industry 4.0 is the fact that existing equipment is not capable of communicating with newly deployed technology (Weber, 2016). This obstacle can be overcome by Enterprise Application Integration (EAI) system, which is created with different methods and on different platforms, and aims at connecting the current and new system processes, providing a flexible and convenient process integration mechanism. The integration of enterprise applications includes the integration of heterogeneous data sources, processes, applications, platforms and standards. By combining software, hardware, and standards, EAI makes sharing and exchanging data and information seamlessly possible (Da Xu, 2011), which is required by Industry 4.0.

2.3 Smart manufacturing systems for Industry 4.0

The Industry 4.0 concept in the manufacturing sector covers a wide range of applications from product design to logistics. The role of mechatronics, a basic concept in manufacturing system design, has been modified to suit CPS (Penas et al., 2017). Smart product design based on customized requirements that target individualized products has been proposed (Zawadzki and Żywicki, 2016). Predictive maintenance (Bokrantz et al., 2017) and its application in machine health prognosis are popular topics in Industry 4.0-based CPS (Xia and Xi, 2019).

Machine Tools 4.0 as the next generation of machine tools has been introduced in machining sites (Xu, 2017). Energy Management 4.0 has also been proposed for decision-based energy data and has transformed energy monitoring systems into autonomous systems with self-optimized energy use (Nienke et al., 2017). Moreover, the implication of Industry 4.0 technologies on logistic systems has been investigated (Hofmann and Rüsch, 2017). This section only presents design, monitoring, machining, control, and scheduling applications. Figure 2.4 presents a framework of Industry 4.0 smart manufacturing systems.

The horizontal axis shows typical issues in Industry 4.0, including smart design, smart machining, smart monitoring, smart control, smart scheduling, and industrial applications. The vertical axis shows issues in another dimension of Industry 4.0 ranging from sensor and actuator deployment to data collection, data analysis, and decision making. In Industry 4.0, data gathering and analysis are the main sources of the smartness of activities shown on the horizontal axis.

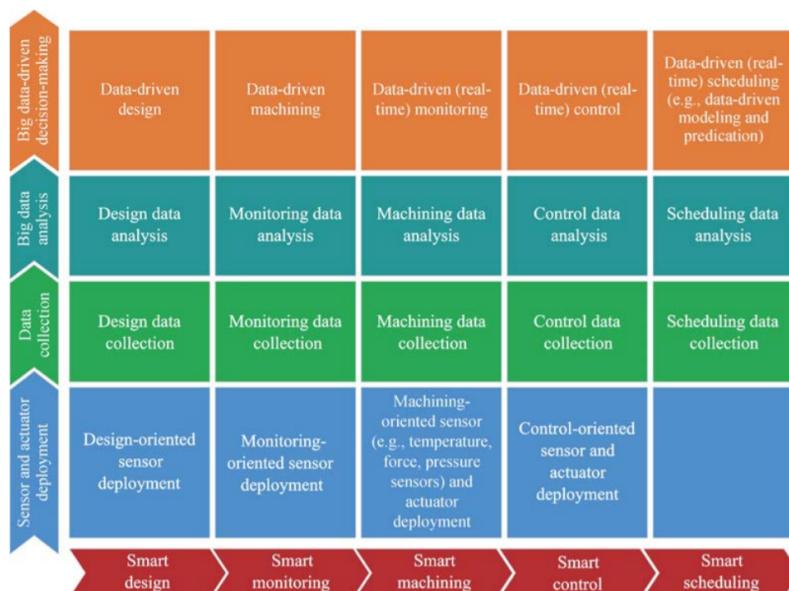


Figure 2.4. Conceptual framework of Industry 4.0 smart manufacturing systems

- **Smart design**

Traditional design has been upgraded and has become smart due to the rapid development of new technologies, such as virtual reality (VR) and augmented reality (AR). Hybrid prototyping using VR techniques has been introduced to additive manufacturing.

Design software, such as computer-aided design (CAD) and computer-aided manufacturing (CAM), can now interact with smart physical prototype systems in real time via 3D printing integrated with CPS and AR (Kolarevic, 2004). Thus, engineering changes and physical realizations could be combined to achieve a smart design paradigm.

- **Smart machining**

In Industry 4.0, smart machining can be achieved with the aid of smart robots and other types of smart objects that can sense and interact with one another in real time (Zhong et al., 2013a).

For example, CPS-enabled smart machine tools can capture real-time data and transfer them to a cloud-based central system so that machine tools and their twined services can be synchronized to provide smart manufacturing solutions. In addition, self-optimization control systems provide in-process quality control and eliminate the need for post-process quality inspection.

- **Smart monitoring**

Monitoring is an important aspect in the operation, maintenance, and optimal scheduling of Industry 4.0 manufacturing systems (Janak and Hadas, 2015). The widespread deployment of various sensors has made smart monitoring possible.

For example, data on various manufacturing objects, such as temperature, electricity consumption, vibrations, and speed, can be obtained in real time. Smart monitoring provides not only a graphical visualization of these data but also alerts when abnormality occurs in machines or tools. CPS and IoT are key technologies that enable smart monitoring in Industry 4.0 smart manufacturing systems.

- **Smart control**

In Industry 4.0, high-resolution, adaptive production control (i.e., smart control) can be achieved by developing cyber-physical production control systems. Smart control is mainly executed to manage various smart machines or tools physically through a cloud-enabled platform (Makarov et al., 2014).

End users can switch off a machine or robot via their smartphones. Decisions can then be timely reflected in frontline manufacturing sites, such as robot-based assembly lines or smart machines.

- **Smart scheduling**

Smart scheduling mainly utilizes advanced models and algorithms to draw information from data captured by sensors. Data-driven techniques and advanced decision architecture can be used to perform smart scheduling. For example, distributed smart models that utilize a hierarchical interactive architecture can be used for reliable real-time scheduling and execution (Marzband et al., 2015).

Production behavior and procedures can then be carried out automatically and effectively because of the well-established structures and services. With the aid of data input mechanisms, the output resolutions are fed back to the parties involved in different ways.

- **Industrial applications**

Industrial applications that target different industry implementations of various solutions are the ultimate goal of Industry 4.0 and may revolutionize manufacturing systems. The solutions provided by Industry 4.0 are sufficiently flexible to support customized configuration and development according to the uniqueness and specific requirements of several

industries, such as the food industry that includes a large number of perishable products. Thus, dynamic manufacturing networks are provided opportunities to manage their supply and business modes (Papakostas et al., 2013).

With the support of configurable facilities from layers of smart design and manufacturing and smart decision making, applications can achieve a holistic perspective by considering practical concerns, such as production efficiency, logistics availability, time constraints, and multiple criteria.

- **Smart design and manufacturing**

Research at this level encompasses smart design, smart prototyping, smart controllers, and smart sensors. Real-time control and monitoring support the realization of smart manufacturing (Zhong et al., 2015). Supporting technologies include IoT, STEPNC, 3D printing, industrial robotics, and wireless communication.

- **Smart decision-making**

Smart decision making is at the center of Industry 4.0. The ultimate goal of deploying widespread sensors is to achieve smart decision making through comprehensive data collection. The realization of smart decision making requires real-time information sharing and collaboration (Zhong et al., 2013b).

Big data and its analytics play an important role in smart decision-making tasks, such as data-driven modeling and data-enabled predictive maintenance. Many technologies, including CPS, big data analytics, cloud computing, modeling, and simulation, contribute to the realization of smart decision making.

- **Big data analytics**

CPS and IoT-based manufacturing systems involve the generation of vast amounts of data in Industry 4.0, and big data analytics is crucial for the design and operations of manufacturing systems.

For example, by using the big data analytics approach, a holistic framework for data-driven risk assessment for industrial manufacturing systems has been presented based on real-time data (Niesen et al., 2016). Such a topic has been widely reported to support production optimization and manufacturing CPS visualization.

- **Industrial implementations**

Industrial applications are the ultimate aim of Industry 4.0. Almost all industries including manufacturing, agriculture, information and media, service, logistics, and transportation, can benefit from the new industrial revolution.

Many new opportunities will be available for industrial parties (Lee et al., 2015). Companies may focus on their core business values or challenges, which could be upgraded or addressed with Industry 4.0-enabled solutions.

Chapter 3

Digital Twins

In this chapter we describe the meaning of Digital Twins, a virtual model of a process, product or service. In particular in:

- In **Section 3.1**: we talk about how this concept has evolved over the years;
- In **Section 3.2**: we define the enabling technologies required to support Digital Twins;
- In **Section 3.3**: we list their integration in Industry 4.0;
- In **Section 3.4**: we provide a description of platforms used to create a digital twin and in particular we focused on Bosch IoT Things, the one used in the development of the thesis. This last section in addition to describing the platform used, collects the definitions of the most important concepts used and a tutorial on its use.

3.1 DT literature

The concept of the Digital Twin dates back to a University of Michigan presentation to industry in 2002 for the formation of a Product Lifecycle Management (PLM) center. The presentation slide, as shown in Figure 3.1 and originated by Dr. Grieves, was simply called “Conceptual Ideal for PLM”.

However, it did have all the elements of the Digital Twin: real space, virtual space, the link for data flow from real space to virtual space, the link for information flow from virtual space to real space and virtual sub-spaces.

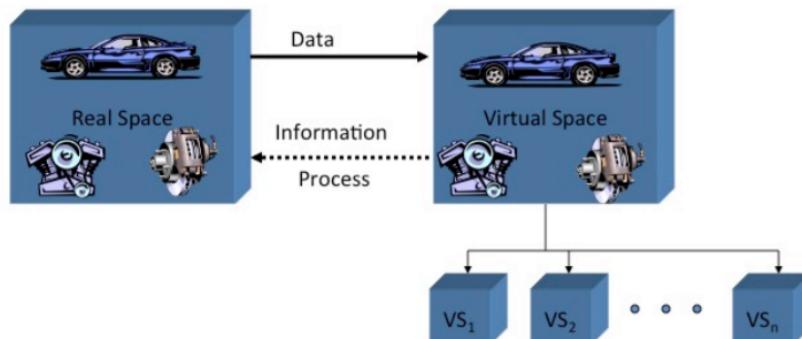


Figure 3.1. Conceptual ideal for PLM

This concept was officially expanded and named as Digital Twin in (Grieves, 2011). Today are among the most promising technologies for smart manufacturing and Industry 4.0. Several definitions emerged in the literature:

- “A Digital Twin is an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin.” (Glaessgen and Stargel, 2012)
- “A digital twin is a computerized model of a physical device or system that represents all functional features and links with the working elements.” (Chen, 2017)
- “The digital twin is actually a living model of the physical asset or system, which continually adapts to operational changes based on the collected online data and information, and can forecast the future of the corresponding physical counterpart.” (Liu et al., 2018)
- “A Digital Twin is a set of virtual information that fully describes a potential or actual physical production from the micro atomic level to the macro geometrical level.” (Zheng et al., 2018)
- “A Digital Twin is a real mapping of all components in the product life cycle using physical data, virtual data and interaction data between them.” (Tao et al., 2018)
- “A Digital Twin is a digital replica of a living or non-living physical entity. By bridging the physical and the virtual world, data is transmitted seamlessly allowing the virtual entity to exist simultaneously with the physical entity.” (El Saddik, 2018)
- “A Digital Twin is a dynamic virtual representation of a physical object or system across its lifecycle, using real-time data to enable understanding, learning and reasoning.” (Bolton et al., 2018)

Based on the given definitions of a Digital Twin in any context, one might identify a common understanding of Digital Twins, as digital counterparts of physical objects. Within these definitions, the terms Digital Model, Digital Shadow and Digital Twin are often used synonymously. However, the given definitions differ in the level of data integration between the physical and digital counterpart.

Some digital representations are modelled manually and are not connected with any physical object in existence, while others are fully integrated with real-time data exchange.

Therefore is proposed a classification of Digital Twins into three subcategories, according to their level of data integration (Kritzinger et al., 2018):

- **Digital Model**

A Digital Model is a digital representation of an existing or planned physical object that does not use any form of automated data exchange between the physical object and the digital object, as shown in Figure 3.2. The digital representation might include a more or less comprehensive description of the physical object.

These models might include, but are not limited to simulation models of planned factories, mathematical models of new products, or any other models of a physical object, which do not use any form of automatic data integration. Digital data of existing physical systems might still be in use for the development of such models, but all data exchange is done in a manual way.

A change in state of the physical object has no direct effect on the digital object and vice versa.

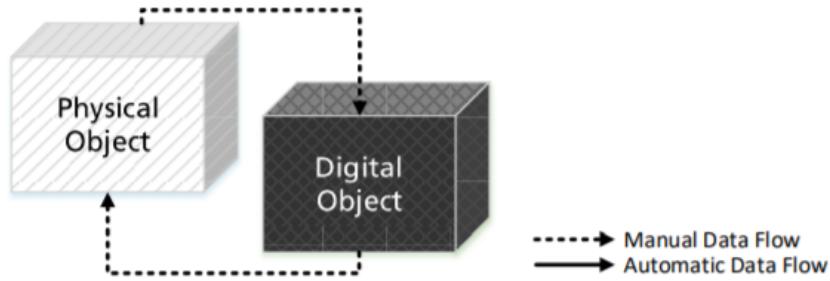


Figure 3.2. Data Flow in a Digital Model

- **Digital Shadow**

Based on the definition of a Digital Model, if there exists an automated one-way data flow between the state of an existing physical object and a digital object, one might refer to such a combination as Digital Shadow, as shown in Figure 3.3.

A change in state of the physical object leads to a change of state in the digital object, but not vice versa.

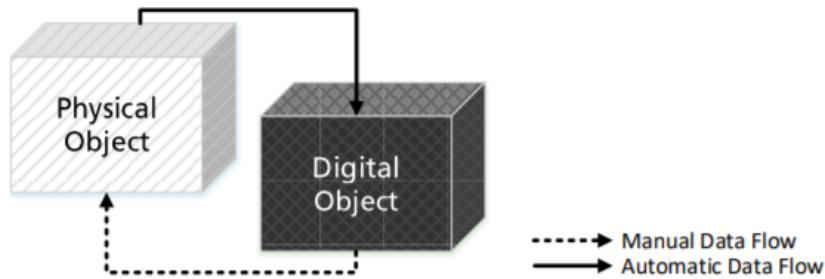


Figure 3.3. Data Flow in a Digital Shadow

- **Digital Twin**

If the data flows between an existing physical object and a digital object are fully integrated in both directions, one might refer to it as Digital Twin as shown in Figure 3.4. In such a combination, the digital object might also act as controlling instance of the physical object. There might also be other objects, physical or digital, which induce changes of state in the digital object.

A change in state of the physical object directly leads to a change in state of the digital object and vice versa.

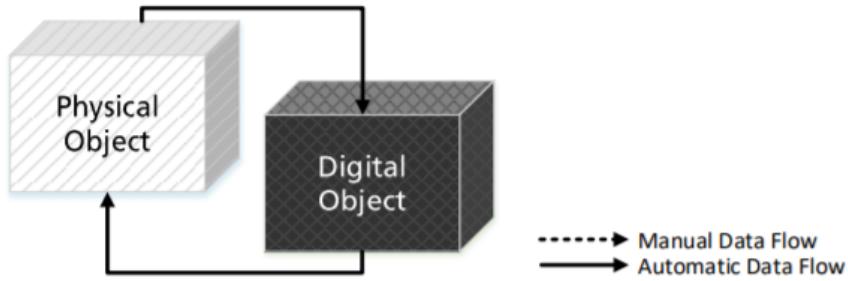


Figure 3.4. Data Flow in a Digital Twin

3.2 Enabling technologies

According to the 5-dimension model, as shown in Figure 3.5 a variety of enabling technologies are required to support different modules of DT:

- **Physical Entity:** full understanding for the physical world is a prerequisite for DT
- **Virtual Model:** represents a faithful replica of the physical entity. It can be a three-dimension geometric model and/or a behaviour model
- **Services:** provided by DT
- **DT Data** provided by:
 - *Physical Model:* both static and dynamic data
 - *Virtual Model:* data resulting from simulations
 - *Service*
- **Connections:** all those that act as a bridge between the physical entity, the virtual entities, services and data

DT involves multidisciplinary knowledge, including dynamics, structural mechanics, acoustics, thermals, electromagnetism, materials science, hydromechatronics, control theory, and more. Combined with the knowledge, sensing, and measurement technologies, the physical entities and processes are mapped to the virtual space to make the models more accurate and closer to the reality. For the virtual model, various modeling technologies are essential. Visualization technologies are of the essence for real-time monitoring of physical assets and processes. The accuracy of virtual models directly affects the effectiveness of DT.

Therefore, the models must be validated by verification, validation & accreditation (VVA) technologies and optimized by optimization algorithms. Besides, simulation and retrospective technologies can enable rapid diagnosis of quality defects and feasibility verification. Since the virtual models must co-evolve with constant changes in the physical world, model evolution technologies are needed to drive the model update. During the operation of DT, a huge volume of data is generated. To extract useful information from raw data, advanced data analytics and fusion technologies are necessary.

The process involves data collection, transmission, storage, processing, fusion, and visualization. DT-related services include application service, resource service, knowledge service, and platform service. To deliver these services, it requires application software, platform architecture technology, service oriented architecture (SoA) technologies, and knowledge technologies. Finally the physical entity, virtual model, data and service of DT are interconnected to enable interactions and exchange information. The connection involves Internet technologies, interac-

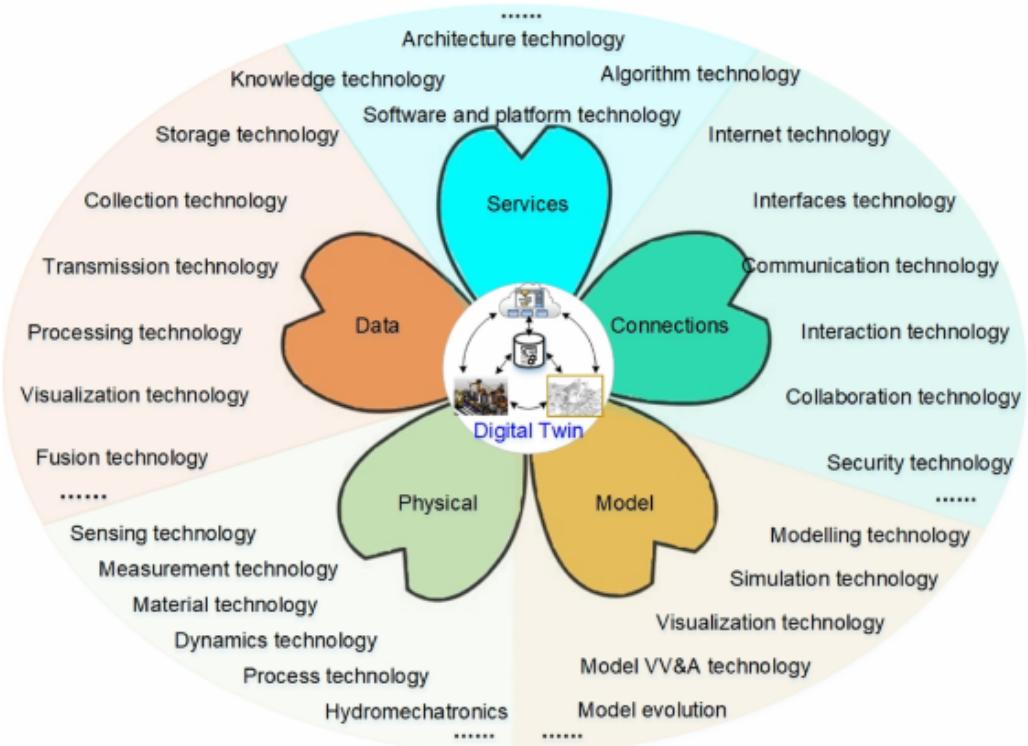


Figure 3.5. Framework of enabling technologies for Digital Twin

tion technologies, cyber-security technologies, interface technologies, communication protocols, etc.

3.3 Digital Twins in Industry 4.0

A digital twin integrates all manufacturing processes (Qi and Tao, 2018), as shown in Figure 3.6:

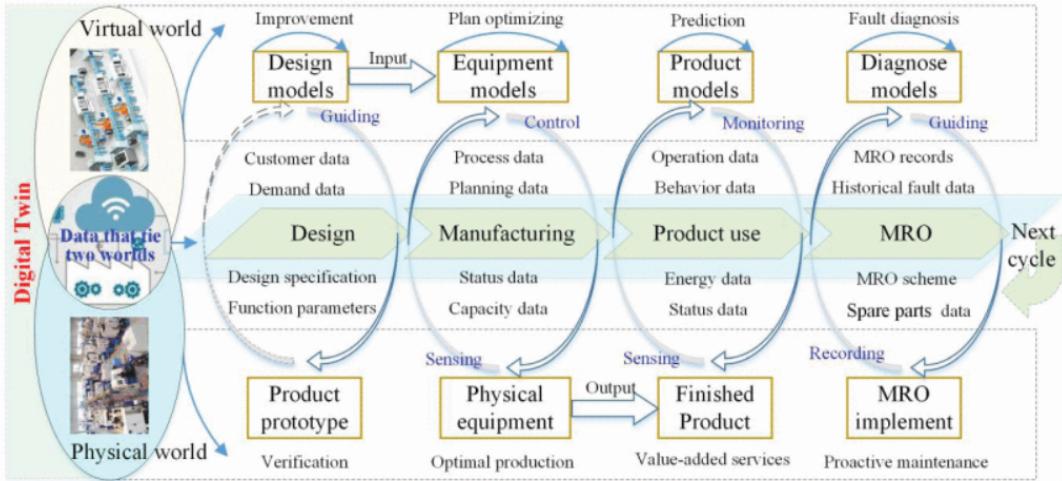


Figure 3.6. Digital twin in manufacturing

1. Digital Twin Based Product Design

In the design phase, it involves back-and-forth interactions between the expected, interpreted, and physical worlds. Based on digital twin, the digital representation (i.e., virtual models) of the physical product is created in the interpreted world (i.e., virtual world). The virtual models reflect both the expectations in the designer's mind, and the practical constraints in physical world. Digital twin enables the iterative optimization of design scheme to guide the designers to iteratively adjust their expectations and improve the design models, achieving personalized product design.

In addition, digital twin driven virtual verification can quickly and easily forecast and verify product functions, behavior, structures and manufacturability, etc.. Taking advantage of digital twin, it can accurately find the defect of design in virtual world and take rapid changes, which make the improvement of the design, avoiding tedious verification and testing.

2. Smart Manufacturing in Digital Twin Workshop/Factory

Next, the proven product design is input into the smart workshop or factory to be manufactured. From the input of raw material to the output of finished products, the whole manufacturing process is managed and optimized through digital twin. The virtual workshop or factory include the geometrical and physical models of operators, material, equipment, tools, environment, etc., as well as the behaviors, rules, dynamics models and others. Before they commit to manufacturing the products, the manufacturing resources and capacities are allocated, and production plan is devised to predefine the manufacturing process. The virtual workshop or factory simulate and evaluate the different manufacturing strategies and planning until a satisfactory planning is confirmed.

In the actual manufacturing execution stage, the real-time monitoring and adjustment of manufacturing process are realized through virtual-physical interaction and iteration. The virtual models update themselves based on the data from the physical world, to keep abreast of the changes. And the problems are rapidly found out and the optimal solution is developed, through simulation in virtual world. According to simulation in virtual workshop or factory, the manufacturing process is adjusted to achieve optimal manufacturing (e.g., accuracy, stability, high efficiency and product quality).

3. Product Digital Twin for Usage Monitoring

The virtual model of product is created to establish the product digital twin. The product digital twin would always keep in company with the product to provide the value-added

services. Firstly, the product in use is monitored in real time, as the product digital twin continually records the product usage status data, use environment data, operating parameters, etc. Consequently, users can keep abreast of the latest state of the product. Secondly, the virtual model can simulate the operation conditions of product in different environments.

As a result, it can confirm what effects the different environmental parameters and operation behaviors would have on the health, lifetime, and performance, etc., so as to control the status and behaviors of physical product (e.g., change the operating parameters). Thirdly, based on the real-time data from physical product and historical data, the product digital twin is able to accurately predict the product remaining life, faults, etc..

4. Digital Twin as Enabler for Smart MRO

Based on the prediction for health condition, remaining life, and faults, the proactive maintenance is carried out to avoid the sudden downtime. Furthermore, when a fault occurs, with the ultra-high-fidelity virtual model of the product, the fault would be visually diagnosed and analyzed, so that the position of faulty part and the root cause of fault are displayed to users and servicemen. Thereby, the MRO strategies (e.g., disassembly sequence, spare parts, and required tools) are developed to recovery the product.

However, before starting the actual MRO (both proactive and passive), the walkthrough about MRO strategies would be executed in the virtual world based on virtual reality and augmented reality. As the mechanical structure of the parts and the coupling between each other are faithfully reflected by the virtual models, it can identify whether the MRO strategies are effective, executable and optimal. Once the MRO strategies are determined, they will be executed to recovery the product. Last, the data from the different stage of product lifecycle are accumulated and inherited to contribute to the innovation of the next generation product.

3.4 Digital Twin platforms

To create Digital Twin there are several platforms: Eclipse Ditto, Bosch IoT Things, AWS IoT (Device Shadow Service), Azure Digital Twins. Between these, for example, Eclipse Ditto displays key advantages compared to the other platforms: it is an open-source project, offers a wide range of connectivity services, highly customizable, embraces the concept of “Device-as-a-Service” (especially relevant in a Digital Twin scenario), extensible with Eclipse Vorto seamlessly.

However, the price to pay in exchange for the listed benefits is a steep learning curve and, therefore, a high barrier to entry for developers. For this reason in the thesis is used Bosch IoT Things platform that alleviates the problem, by bringing together Eclipse Hono, Eclipse Ditto and Eclipse hawkBit: due to its pre-configured format, developers do not need to integrate the different open source projects. The trade-off for a more user-friendly environment is the subscription cost: Bosch IoT Things sets a limit to the number of API calls and managed data volume to the free version, besides which it is necessary to own a paid subscription.

It also provides the following advantages:

- A wide range of connectivity services that will be useful in the execution engine implementation
- A built-in search index, helpful to retrieve things that satisfy given conditions
- A selective change notification system, which allow to select only a set of desired events
- A generic yet extensible description language using JSON, upon which a new ad-hoc description language will be based

- The benefits provided by a cloud platform like: security, flexibility, mobility insight, quality control, disaster recovery, loss prevention, automatic software updates, competitive edge, sustainability

3.4.1 Bosch IoT Things

The following subsection contains parts of the Bosch IoT Things documentation, which can be found at the following link: <https://docs.bosch-iot-suite.com/things/>.

Bosch IoT Things enables applications to manage digital twins of IoT device assets in a simple, convenient, robust and secure way. Based on the digital twin approach, applications can manage asset data, get notified automatically on all relevant changes of their IoT devices, and share device data and functionality across the layers of their application or with 3rd-party applications.

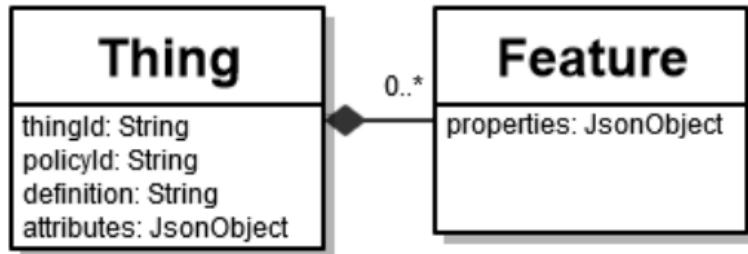
The digital twins of your tools, cars, sensors, and other web-enabled things are subsequently able to interact with one another. You can also enrich your digital twins with further capabilities based on the information or functionality provided by additional systems. General aspects:

- **Ready-to-use cloud offering**
 - Bosch IoT Things is part of pre-configured packages with connections between all its microservices
 - Can also be booked stand-alone
- **Fully managed, shared cloud service**
 - Various service plans: free (for evaluation), starter and standard
 - Pay-as-you-grow price model
 - Online calculator to estimate the consumption for your scenario
- **High availability & reliability**
 - Always-on (availability 99,5%)
 - High resilience of the system
- **Scalability**
 - Growth on demand
 - Quick response to increased data volume and transactions
- **Open & flexible**
 - Based on open source project: Eclipse Ditto
 - Flexibility to integrate with 3rd-party applications and other infrastructures

3.4.2 Overview

Each Thing consists of:

- **A thing ID**
- **A policy ID** which links to a Policy containing the authorization information
- **A definition** that documents how a feature's state is structured, and which behavior/capabilities can be expected from such a feature
- **Attributes:** intended for managing static meta data of a Thing which does not change frequently

**Figure 3.7.** Domain model of a Thing

- **Features:** a Thing can be designed with as many Features as desired. Usually, features are intended for managing dynamic meta data

To represent a Thing operatively in Eclipse Ditto, a JSON format is used. Here an example of one Thing with one Attribute and one Feature:

```
{
  "thingId": "my.namespace:myFridge",
  "policyId": "my.namespace:myFridgePolicy",
  "definition": "digitaltwin:DigitaltwinExample:1.0.0",
  "attributes": {
    "location": "Kitchen"
  },
  "features": {
    "temperature": {
      "properties": {
        "cur_temp": 5
      }
    }
  }
}
```

3.4.3 Example

Figure 3.8 shows an example of an e-bike deployment of the multiple actors which can access the Digital Twin and shows that the location and battery features could rely on Eclipse Vorto function block definitions, while the guarantee feature is free form, to emphasize that using such definitions is optional.

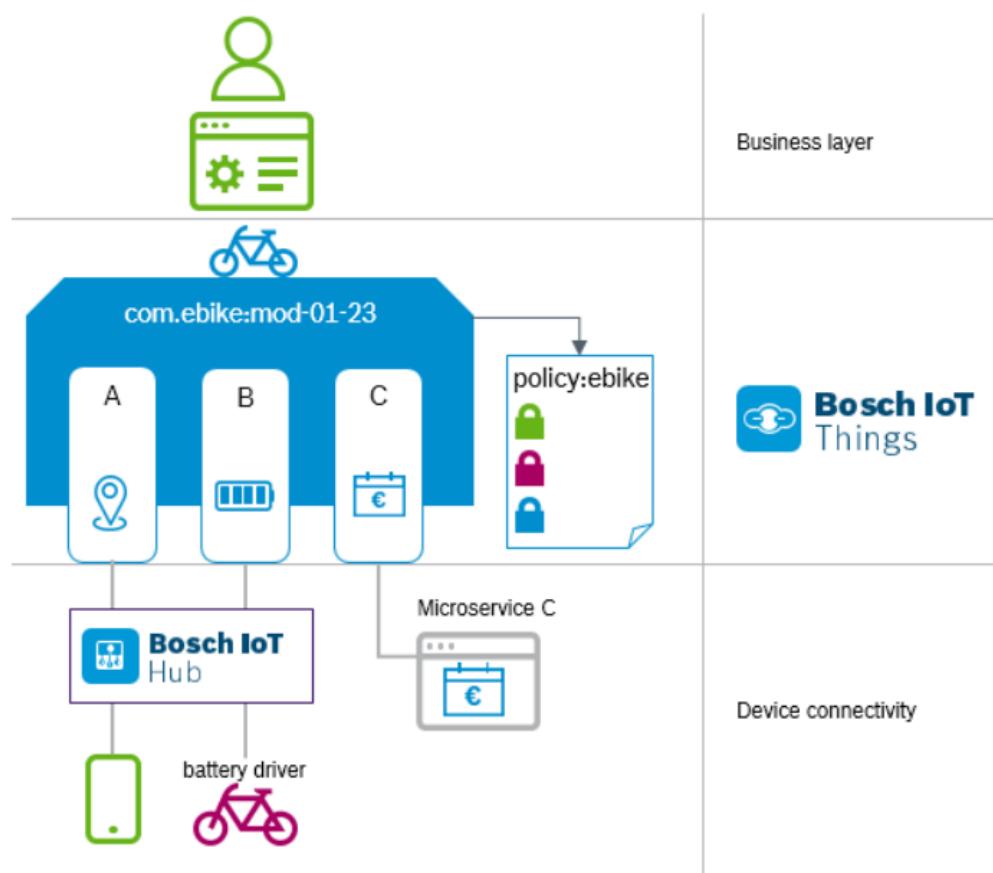


Figure 3.8. Bosch IoT Things example architecture

When implementing your digital twin you can re-use already existing function blocks, or create some new ones, specific to your devices. The data, where the status information for your digital twin come from, can be diverse. You can imagine for example that:

- the data representing the battery level comes from the ebike directly,
- the location coordinates are sent by a navigation system which is coupled to the physical ebike,
- the guarantee information is stored in a CRM system.

3.4.4 Implementation and Deployment

Digital twins are typically composed of multiple implementations for their different features.

- **For features that mainly represent a state with properties** (e.g. device state), the state management within Bosch IoT Things can be used. This state is then updated e.g. by the devices using a device connectivity layer like Bosch IoT Hub, or by a business application that updates configuration properties within this state.
- **The Things service** then cares about applying the state changes and notifying all relevant components about the changes. Functionality that is not reflected as state (e.g. events, operations, triggers), is managed by Bosch IoT Things as messages. Messages can be sent to features and/or received from features. These messages are not processed within the Things service but are routed and dispatched to the respective implementation, which is provided separately.

Functionality of IoT devices is handled this way using messages and the Things service routes the messages from/to the device connectivity layer, e.g. using Bosch IoT Hub. Depending on the type of operation, these messages are either one-way messages or are correlated with response messages in order to represent the result of an operation.

- **Features with other types of functionality or features representing state that is managed externally**, must be implemented by integrating separate microservices/-components. These microservices are responsible to listen to signals/notifications coming from the Things service, process the respective signal and optionally send responses. These microservices can use one of the provided bindings to integrate with the Things service (WebSocket, AMQP 1.0, or AMQP 0.91).

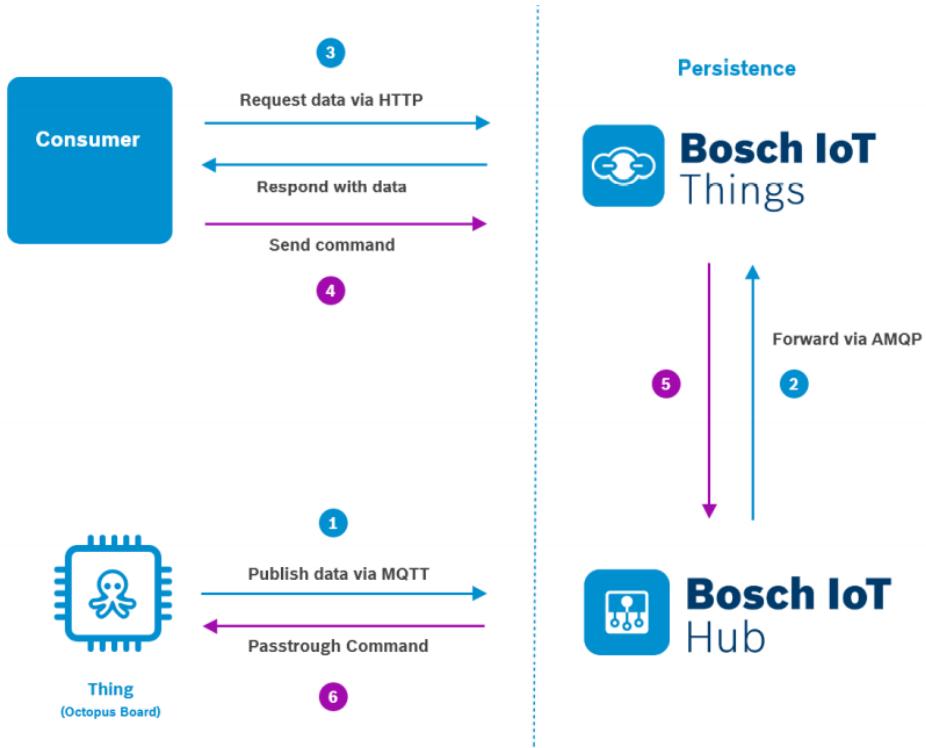
Summarized, this means that for “simple” digital twins, mainly consisting of device state and device functionality, the management of Bosch IoT Things integrated with a device connectivity layer like Bosch IoT Hub is sufficient. For more complex digital twins you can extend this pattern, by integrating separate microservices/components which implement specific features.

3.4.5 Communication

In Figure 3.9, there is a bidirectional communication between a board (equipped with a temperature sensor and a LED) and the Bosch IoT Hub:

- The board transmits the detected temperature and the LED status to the Bosch IoT Hub periodically (according to a time interval), via MQTT. A client is able to query the detected temperature and the LED status to Bosch IoT Hub via the HTTP REST API
- A client is able to set the value of the LED remotely, sending a command to Bosch IoT Hub (via HTTP REST API) that then is forwarded to the board (via MQTT)

For the following example, it is necessary to own a Bosch IoT Suite subscription.

**Figure 3.9.** Communication example

The first step is to create a namespace (all things and policies are required to be created with a namespace):

1. Go to “dashboard”
2. Navigate to the “namespace” tab
3. Input the desired namespace name following the reserve domain name notation (in the example “my.test”)

Then, it is necessary to create a token to access the HTTPS API:

1. Click the user icon
2. Go to “oAuth2 Clients”
3. Follow the instructions to generate the token
4. Copy the token to a temporary document

In order to execute the device registration:

1. Go to the Bosch IoT Suite Device Provisioning API page
2. Authorize API request via the authorization token, by clicking on the “Authorize” button on the upper right corner and paste the token into the dedicated input field
3. Provide the “service-instance-id” on the required input-field (it can be found in the “Credential” section of the dashboard)
4. Edit the body request. In the example:

```
{
  "id": "my.test:octopus",
```

```

"hub": {
    "device": {
        "enabled": true
    },
    "credentials": {
        "type": "hashed-password",
        "secrets": [
            {
                "password": "<any-password>"
            }
        ]
    }
},
"things": {
    "thing": {
        "attributes": {
            "name": "octopus",
            "type": "octopus board"
        },
        "features": {
            "temp_sensor": {
                "properties": {
                    "value": 0
                }
            },
            "LED": {
                "properties": {
                    "value": 0
                }
            }
        }
    }
}
}

```

5. Send the request and save the response containing the policy ID of the just created thing (the device provisioning request generated a default policy for the thing) and authorization ID for the device (it will be necessary for the device to access the IoT Hub)

In order to add a subject to the policy and in the example, to give read access to the registered thing on the thing dashboard:

1. Go to the Bosch IoT Things HTTP API page
2. Select the

```
"PUT /policies/policyId/entries/label"
```

API call

3. Authorize the API request again, if necessary (each token lasts for 30 minutes)
4. Provide the previously received policy ID and set the desired label for the new subject
5. Edit the body request (with the correct technical user id). In this example:

```
{
```

```

"subjects": {
    "bosch:<your-technical-user-id>": {
        "type": "bosch-id"
    }
},
"resources": {
    "thing:/": {
        "grant": [
            "READ"
        ],
        "revoke": [
        ]
    }
}
}

```

Upon success, the thing should be visible in the dashboard, under the “Things” section.

A Python script simulates the board. Due to the fact that the MQTT communication with Bosch IoT Hub is encrypted, it is necessary to download the server certificate for MQTT TLS via the following command:

```
curl -o iothub.crt https://docs.bosch-iot-suite.com/hub/cert/
iothub.crt
```

In order to change the state of the LED, another HTTP API call will be issued with the Bosch IoT Things HTTP API page:

1. Go to the Bosch IoT Things HTTP API page
2. Select the

```
"POST /things/thingId/features/featureId/inbox/
messages/messageSubject"
```

API call

3. Authorize the API request again, if necessary (each token lasts for 30 minutes)
4. Edit the “thingId”, “featureId” and the “messageSubject” fields, and the body to the desired value of the LED
5. Click “Execute”

In order to retrieve all the data associated to the board:

1. Go to the Bosch IoT Things HTTP API page
2. Select the

```
"GET /things/thingId"
```

API call

3. Authorize the API request again, if necessary (each token lasts for 30 minutes)
4. Edit the thingId
5. Click “Execute”

Chapter 4

Framework

In this chapter, after providing an overview of the most important concepts of Industry 4.0 and digital twins, we describe the structure of our framework, the intermediate state between the concepts previously explained and the software that we will use. The chapter is structured as follows:

- In **Section 4.1**: we give an overview on how Digital Twins can be represented in an Industry 4.0 scenario;
- In **Section 4.2**: we explain an architecture representation of the previous overview;
- In **Section 4.3**: we provide an example of everything described so far;
- In **Section 4.4**: we define a general description of our use case, in particular of: the available services, the target and the orchestrator;
- In **Section 4.5**: we underline technological aspects of our use case, that will be presented in detail in the next chapters.

4.1 Introduction

The evolution of technologies in the fields of communication, networking, storage and computing that found its way in the traditional world of industrial automation, increase productivity and quality to ease workers' lives and define new business opportunities, goes under the name of *smart manufacturing or Industry 4.0*, as fully explained in the chapter 2 . The technological foundation of smart manufacturing consists of cyber-physical systems and the Internet-of-Things (IoT). Each IoT device in a smart factory can be coupled with a digital twin, that is, a dynamic virtual representation of the physical system across its life-cycle using real-time sensor data.

Currently, the manufacturing process itself, the involved devices, and how they interact, is designed by human experts in a traditional way. *Digital factory* is a key concept. It aims at using digital technologies to promote the integration of product design processes, manufacturing processes, and general collaborative business processes across factories. An important aspect of this integration is to ensure interoperability between machines, products, processes, and services. A digital factory consists of a multi-layered integration of the information related to various activities along the factory and related resources. Actors can fall in different categories, being humans (i.e., final users or participants in the production process), information systems or industrial machines. These physical entities must have a faithful representation in the digital world, usually referred to as digital twins.

A *digital twin* (DT) exposes a set of services allowing to execute certain operations and produce data describing its activity, as widely described in chapter 3. We can imagine these data stored

in a factory data space together with other information, e.g., data available from the company and production history, worker suggestions and preferences. Such services are typically used to query or manipulate the state of the system, and associated data are leveraged for diagnostics and prognostics. The availability of DT services and data can have a huge impact on the design of manufacturing processes, by allowing automatic recovery and optimization, and even automatic composition of the intermediate steps for achieving a production goal.

Inspired by the research about automatic orchestration and composition of software artefacts, such as Web services, we argue that:

- an important step towards the development of new automation techniques in smart manufacturing is the modeling of DT services and data as software artifacts;
- the principles and techniques for composition of artifacts in the digital world can be leveraged to improve automation in the physical one.

A crucial difference between traditional software artifacts used in composition techniques and DTs is that DTs may not share the same view of the world. Modern information systems and industrial machines may natively come out indeed with their digital twin. In other cases, especially when the approach is applied to already established factories and production processes, digital twins are obtained by wrapping actors that are already in place. In such scenarios, data management techniques (including integration and exchange) are a key ingredient for DTs interoperability

4.2 Architectural Model

At the foundation of the contribution of the thesis there is the architectural model proposed in (Catarci et al., 2019), which we will briefly review in this section. This model, inspired by the Roman model for service composition (formally described in the next chapter 5), considers smart manufacturing scenarios where DTs of physical systems – or, simply, twins – provide stateful services wrapping the functionalities of machines and tasks of human operators. In such contexts, data are usually available through several sources not sharing a common schema and vocabulary, as DTs come from different vendors.

It is reasonable to consider an heterogeneous data space where a mediator is present and it also takes the role of translator between different formalisms and access methods. We consider learning as a fundamental feature of DT. Learned functions include the automatic generation of alarms, but also automatic triggering of actions and status changes. Additionally, twins can be queried on learned functions and, as a result, the data space is far more dynamic than in more traditional scenarios. As in modern micro-services architectures, notifications based on *publish&subscribe* is a common architectural pattern, and therefore we provide for subscriptions to events generated by other DTs.

We consider an architecture for a smart manufacturing process based on DTs as depicted in Figure 4.1, where the main components are the DTs, the data space, human supervisors and a mediator.

DTs wrap physical entities involved in the process. These physical entities can be manufacturing machines or human operators. A DT exposes a Web API consisting, in general, of three parts:

- **the synchronous interface:** allows to give instructions to the physical entity. These instructions may, for example, produce a state change in a manufacturing machine (in case the twin is over a machine) or ask a human operator to perform a manual task (in case the twin is over a manufacturing worker);
- **the query interface:** allows for asking information to the physical entity about its state and related information; noteworthy, these latter can be obtained by applying diagnostic

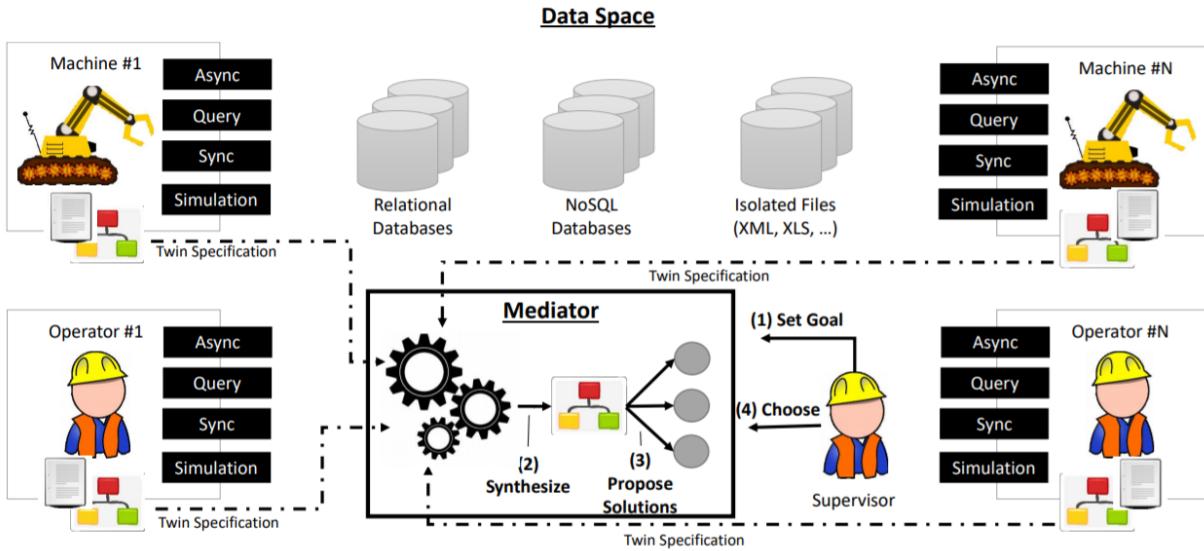


Figure 4.1. The proposed architecture

and prognostic functions results of machine learning;

- **the asynchronous interface:** generates events available to subscribers. It is important to note how manufacturing machines and human actors can be considered identical from the point of view of the offered API, e.g., human actors produce asynchronous events as well, for example generating alarms.

The data space contains all the data available to the process. These data are heterogeneous in their nature from the access technology point of view, the employed schema (or its absence) and the employed vocabulary. It is important to note how the DTs contribute to the data space with both the query API and the asynchronous one. Other sources for the data space may include relational and no-SQL databases or unstructured sources such as spurious files, which constitute the factory information system.

The human supervisor is the one defining the goals of the process in terms of both final outcomes and key performance indicators to be obtained. In the Roman model lingo, this is also called the *target* service. In order to reach the goal defined by the human supervisor, available twins and data must be integrated. This task is fulfilled by the mediator. The mediator acts in two phases:

- **the synthesis phase:** during this the specifications of the APIs exposed by digital twins and the meta-data (e.g. data source schemas) available in the data space, are composed in order to construct a mediator process;
- **the execution phase:** the mediator runs its program by preparing the input messages for the single twins involved in the proper sequencing/interleaving.

Indeed, as each twin may potentially adopt a different language and vocabulary, in order to compose required input/output messages, the mediator translates and integrates the data available in the data space to comply with the format requested by the specific called service. An important aspect of the proposed architecture is that multiple companies can participate in the process (typically those ones involved in the value chain). Again, it is not reasonable to have twins directly communicating with one another.

Once again, the role of the mediator is fundamental, being the component that can access the

services offered by the twins available in the different companies.

1. The data space as a polystore

In digital factories, it is of strategic importance to provide effective mechanisms for searching information along diverse and distributed data sources. Indeed, when dealing with data management, organizations are becoming polyglot (Maccioni and Torlone, 2018): they tend to adopt the data management systems that are most suitable to the kind of data, that can significantly vary. Polystores, together with its first reference implementation BigDAWG (Duggan et al., 2015), have been proposed recently as a valuable solution for this scenario. A polystore system provides a loosely coupled integration over multiple, disparate data models and query languages.

In this system, queries are posed over islands of information, i.e. collections of data sources, each accessed with a single query language, and the same data can be accessed from multiple islands. Data transformation and migration in polystores have been considered in (Dziedzic et al., 2016). In our approach, the data space can be modelled as a polystore. We inherit the data modelling approach proposed in (Maccioni and Torlone, 2018) where a polystore is made of a set of databases stored in a variety of data management systems, each one potentially offered by a twin through the query interface. A collection of operators act on the polystore with the aim of supporting the data access needs due to digital twin composition.

2. Modelling the DTs

We model the behaviour of DTs as guarded automaton (by extending (Berardi et al., 2005)): a DT wraps a physical entity which, in a production process, follows specific stages/steps. The synchronous API of the twin corresponds then to input messages of the guarded automaton. The main difference here consists in what the local storage contains and how this influence automaton transitions and atomic processes. In (Berardi et al., 2005), the local storage of a Web service contains variables instantiated and modified by the automaton during the execution, and by messages sent by the mediator. Here we extend to have autonomous threads enriching the local storage with measurements coming from sensors and outcome of machine learning predictions. In this sense, data provided by sensors and machine learning are somewhat similar. We add a further UStore (where the U stands for uncertain) containing triples (s, m, c) where s is a variable name corresponding to a sensor or a prediction task, m is the information measured or predicted, and c is the confidence in this value.

As a consequence, we can define the transition conditions of the guarded automaton in terms of UStore as well. The result is the possibility to include in the model automatic transitions that are not the result of explicit message exchange. The data contained in the local storage is part of the polystore defining the process. In order to query this data, each digital twin provides the query API. The DT can additionally provide asynchronous data to other twins and to the mediator itself through its asynchronous API. In particular, asynchronous events can be generated in response to a change of the UStore or to the transition of the twin automaton. Interested recipients subscribe to this events following a publish/subscribe architectural pattern.

3. Actual and simulation perspectives

Machine learning represents a fundamental feature of DTs, especially for simulation purposes. Most real systems that are confronted with multiple data streams benefit from machine learning and analysis to make sense of the data. For example, machine learning can automate complex analytical tasks, evaluate data in real time, regulate behavior with the minimum need for supervision, and increase the likelihood of desired results (Madni et al., 2019). The uses of machine learning within a DT include: supervised learning (for example, using neural networks) of the preferences and priorities of the user in an experimental test bed based on simulation (Schluse et al., 2017); learning without super-

vision of objects and models using, for example, clustering techniques in virtual and real environments (Madni and Sievers, 2018); and strengthening the learning of system and environment states in uncertain and partially observable operating environments. Another fundamental feature of a digital twin is simulation.

We have already introduced the role of machine learning in the population of the UStore, but it has a fundamental role in simulation too. In particular, when proposing possible solutions, the mediator may require to simulate the result of operations on the twins following the human-in-the-loop philosophy. When the supervisor takes a decision, the actions in the proposed plan are executed in the real world and the actual results can be used to improve the simulation feature in a reinforcement learning pattern. In order to do this, both twins and data space have two perspectives. The actual perspective reflects the current state of the physical world, whereas the simulation perspective allows the mediator to perform simulations useful to produce alternatives for the supervisor. As a result, each DT has two instances of the corresponding automaton, in different states and with different values in the local storage. The same holds for the other data sources composing the polystore. To access the functionalities of the simulation perspective, the DT provides a further simulation API.

4. Semantics

The principal differences between the mediator proposed in (Berardi et al., 2005) and the evolution we propose to meet the requirements of smart manufacturing, are (i) the way the goal is defined by the supervisor, and (ii) the need for it to operate a translation over the data available in the data space in order to perform its coordination task between the digital twins required to reach the goal. In (Berardi et al., 2005), the goal of the mediator was a guarded automaton to be synthesized combining the automata of the single services. This is not appropriate for our scenario, as it requires an expensive human work and the vocabulary of the automaton to be compliant with that of twins.

We propose instead to formulate goals in terms of key performance indicators in a declarative manner. In order to do that, the goal declaration must be translated in terms of the input/output messages of twins' automata and the content of the data space. Additionally, the mediator must be able to discover twins and their capabilities in terms of offered services.

4.3 Example

In the following, we will give an intuition of the proposed approach by considering a cardboard boxes real manufacturing scenario involving three companies:

- the cardboard manufacturer,
- the die cutter manufacturer,
- a delivery service.

For the sake of simplicity, we will consider three twins:

- the twin corresponding to the die cutter manufacturer,
- the twins corresponding to the delivery services (potentially they might be many),
- a twin corresponding to a single die cutter installed at the cardboard manufacturer factory

The polystore will contain the APIs of the twins and the historical production and shipping data from the die cutter manufacturer and the delivery services.

The digital twin corresponding to the die cutter contains at least two states: mounted and unmounted. At any time, the twin provides information about the number of rotations performed

and the residual life expectancy. The twin corresponding to the die cutter manufacturer may provide information about the time needed to produce a new die cutter, whereas the twin of a delivery service may provide information about the expected shipping time from the die cutter manufacturer factory to the cardboard manufacturer factory.

The twin corresponding to the die cutter may expose a simulation API that can help to simulate what happens if the die cutter undergoes a given setting of the rotation speed throughout a 24h period.

The production goal may be to avoid interruptions in the production process. Currently, for economic reasons, manufacturers prefer not to store in the warehouse replacement production machines and tools. As a consequence, the supervisor may instruct the mediator to order a new die cutter by predicting when the current one is going to break, taking into account production time on the die cutter manufacturing site and shipping time. In order to satisfy the goal, the mediator will discover the different services and automatically understand how to combine them.

4.4 Use Case

The example described above refers to the classical composition of services of the Roman model that assumes both target and available services to be deterministic. Previous works have generalized the technique in the case the target requests action according to a probability distribution, and collects rewards for being able to execute actions (Brafman et al., 2017) (see also Section 5.4). An interesting feature of that approach is that, if a composition exists, the computed orchestrator coincides with the exact solution; otherwise it provides an approximate solution that maximizes the expected sum of values of user requests that can be serviced. Nevertheless, this models does not capture certain aspects of the available services, such as stochastic behaviour or action execution costs. For this reason we define a new extension with stochastic service composition, as we describe in detail later in Section 6.

In this section, we propose an use case in the context of Industry 4.0 whose aim is to motivate the extension to Service Composition with Stochastic Services, and the application of such novel composition technique to the orchestration of Digital Twins.

4.4.1 Scenario: Ceramics production

The scenario is the following: there is an industrial process of ceramics production in which a product must be processed sequentially in different ways.

Available Services. Each sub-task can be completed by a set of *available services*. The tasks to be carried out in order to complete the industrial process are: provisioning, moulding, drying, first baking, enamelling, painting, second baking and shipping. Such tasks can be accomplished by different types of machines or human workers. Each available service that can perform the task can be seen as finite state machines with a probability and a reward associated to each action. There could be multiple services for the same task, e.g. multiple version of a machine (new one and old one) and a human that can perform the task required, and so on.

When an available service is being assigned a task, this has a *task cost* in terms of time taken and resources needed for the completion of the operation on that specific service. Each available service is simulated by its own process. Usually, in terms of task cost, machines are cheaper than human workers, because they can perform their task much faster. However, the machines have a certain probability to *break* when they perform their job. In such a case, the machine must be repaired as soon as the operation has been carried out, that incurs in a *repair cost* for that specific machine.

These parameters could change: although the machine is more efficient, if it begins to have significant wear (so higher probability to break and higher cost to be repair), it will be more

convenient to use the human to perform the task. Services like provisioning and shipping have a single state, with no possibility to broken.

Target. The product to be assembled will be represented by the *target* of the model. The target represents what the user want to do. Given the available services the user choose to do an action that is performed by the dedicated service. The action performed has a certain probability and a reward associated. Since we are in an Industry 4.0 scenario, aimed at the ceramics production, we imagine a linear process where is performed one action at a time by an available service.

Orchestrator. The goal of the orchestrator is to orchestrate all the services according to the target requests in such a way that the overall expected sum of rewards is maximized (or, equivalently, that the expected sum of costs are minimized), even if the orchestration is not guaranteed to succeeds in all the cases, as in (Brafman et al., 2017). However, differently from (Brafman et al., 2017), in the optimization problem, the orchestrator should also take into account the costs and probabilities associated to the services' actions. We remark that this is a limitation of the previous work (Brafman et al., 2017).

4.4.2 Discussion

From the above description of the use case scenario, it is clear that the composition technique must be able to handle the stochasticity of the available services' transitions, as well as their reward/cost. Indeed, an optimal orchestration depends on several parameters, like the task costs, the breaking probabilities and the repair costs, one for each candidate service for accomplish a certain task. Therefore, it is not straightforward to determine a priori which service a certain task must be assigned to. For example, it might be the case that despite the task cost of a machine is low, its breaking probability might be high, and considering the repair cost it might let us to prefer a human worker for that task. We argue that our model can fit very well our use case. Indeed, we can reduce the problem to an instance of stochastic service composition suggested above in which a service can capture the task cost, the breaking probability, and the repair cost.

In Chapter 6 we will formally describe the new framework, that also work with stochastic services, and we will provide a solution technique for such problem.

4.5 Technological Aspects

We implemented a proof-of-concept by representing each finite state machines (available services and the target) as a Digital Twin, with all of them connected to the Bosch IoT Things platform. We call each Digital Twin one at a time, where we give them the action to perform and wait for the feedback (that the action has been performed) and the update status (after performing the action).

All the available services and the target service are simulated in separated processes. They are connected to the Bosch IoT platform. They communicate with the orchestrator through the platform. The orchestrator process is connected to the Bosch IoT platform. Before the start of the orchestration, the orchestrator downloads the formal specifications of all the available services and the target service, i.e. the state machines with the transition probabilities, the costs and the rewards. Then, it computes an optimal orchestration policy according to the underlying optimization problem, i.e. the maximization of the expected observed rewards. Finally, it starts waiting for requests from the target process. The target chooses the first action according to the probability distribution, and sends it to the orchestrator process. The orchestrator dispatches them to the chosen service according to the computed optimal policy, and then it waits until the execution of the action is completed. The service executes the action (in the proof-of-concept,

it is a no-op) and notifies the orchestrator of the completion of the task. The orchestrator, in turn, notifies the target process that it can continue its execution. The target chooses a new action according to the current action distribution, and the cycle continues.

The following chapters describe in greater detail each component of our system, as well as the theoretical formalization of the new framework.

Chapter 5

MDP and Stochastic Service Composition

In this chapter, we report some preliminary notions to understand the technical content of the next chapter. In particular, we will report: the definition of Markov Decision Processes (MDP) (Puterman, 1994), a discrete-time stochastic control process very popular in the literature, and the definition of the problem of service composition, with its stochastic variant, and how to solve it using the so called Roman model (Berardi et al., 2003).

The chapter is structured as follows:

- In **Section 5.1**: we start with the mathematical definition of MDP and its main concepts;
- In **Section 5.2**: we continue with the various techniques for finding an optimal policy like policy iteration and value iteration, providing their algorithms;
- In **Section 5.3**: we describe the problem of service composition, with reference to the Roman Model;
- In **Section 5.4**: we explain the stochastic service composition taken from (Brafman et al., 2017).

5.1 Definition

A Markov Decision Process (MDP) is a tuple $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$ where:

- S is the set of *states*,
- A is the set of *actions*,
- $T : S \times A \rightarrow \text{Prob}(S)$ is *the transition function* that returns for every state $s \in S$ and action $a \in A$ a distribution over the states,
- $R : S \times A \rightarrow \mathbb{R}$ is *the reward function* that specifies the average reward (i.e. a real value) received by the decision-maker when performing action a in state s ,
- γ is *the discount factor*, with $0 \leq \gamma \leq 1$, that indicates the present value of future rewards.

With $T(s, a, s')$ we denote the probability to end in state s' given the action a from s .

The discount factor γ deserves some attention. Its value highly influences the MDP, its solution, and how the agent interprets rewards:

- if $\gamma = 0$, we are in the pure *greedy setting*, i.e. the agent is shortsighted and looks only at the reward that it might obtain in the next step, by doing a single action;

- the higher γ , the longer the sight horizon, or the foresight, of the agent: the far rewards are taken into account for the current action choice;
- if $\gamma < 1$ we are in the *finite horizon* setting: namely, the agent is intrinsically motivated to obtain rewards as fast as possible, depending on how γ is far from 1;
- when $\gamma = 1$ we are in the *infinite horizon* setting, which means that the agent considers far rewards as they can be obtained in the next step. In other words, we may think the agent as immortal, since the time the agent spend to reach rewards does not matter anymore.

A solution to an MDP \mathcal{M} , called *policy*, is a mapping from states to a probability distribution over a set of actions:

$$\rho : S \rightarrow \text{Prob}(A) \quad (5.1)$$

A policy is said *deterministic* if the probability distributions take only a single value, i.e. ρ is a function of the form: $\rho : S \rightarrow A$.

Given a sequence of rewards $R_{t+1}, R_{t+2}, \dots, R_T$, the *expected discounted* return G_t at time step t is defined as:

$$G_t := \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (5.2)$$

where T can be $T = \infty$ and $\gamma = 1$ (but not both).

The *value function* of a state s , the *state-value function* $v_\rho(s)$ is defined as the expected return when starting in s and following policy ρ , i.e.:

$$v_\rho(s) := \mathbb{E}_\rho[G_t \mid S_t = s], \forall s \in S \quad (5.3)$$

Similarly, we define q_ρ , the *action-value function for policy* ρ , as:

$$q_\rho(s, a) := \mathbb{E}_\rho[G_t \mid S_t = s, A_t = a], \forall s \in S, \forall a \in A \quad (5.4)$$

Notice that we can rewrite (5.3) and (5.4) recursively, yielding the *Bellman equation*:

$$v_\rho(s) = \sum_a \rho(a \mid s) \left(R(s, a) + \gamma \sum_{s'} P(s' \mid s, a) v(s') \right) \quad (5.5)$$

where we used the definition of the transition function:

$$T(s, a, s') = P(s' \mid s, a) \quad (5.6)$$

We define the *optimal state-value function* and the *optimal action-value function* as follows:

$$v^*(s) := \max_\rho v_\rho(s), \forall s \in S \quad (5.7)$$

$$q^*(s, a) := \max_\rho q_\rho(s, a), \forall s \in S, \forall a \in A \quad (5.8)$$

Notice that with (5.7) and (5.8) we can show the correlation between $v_\rho^*(s)$ and $q_\rho^*(s, a)$:

$$q^*(s, a) = \mathbb{E}_\rho[R_{t+1} + \gamma v_\rho^*(S_{t+1}) \mid S_t = s, A_t = a] \quad (5.9)$$

We can define a partial order over policies using value functions, i.e. $\forall s \in S. \rho \geq \rho' \Leftrightarrow v_\rho(s) \geq v_{\rho'}(s)$.

An optimal policy ρ^* is a policy such that:

$$\rho^* \geq \rho, \forall \rho \quad (5.10)$$

Definition 5.1. The definition of optimal policy is: the value of a policy ρ at state s , denoted by $V^\rho(s)$, is the expected sum of (possibly discounted by a discount factor γ , with $0 \leq \gamma \leq 1$) rewards when starting at state s and selecting actions based on ρ . Formally:

$$V^\rho(s) = \mathbb{E}_{s \sim s_0, \rho, T} \left[\sum_{i=0}^{\infty} \gamma^i r_{i+1} \right] \quad (5.11)$$

Typically, the MDP is assumed to start in an initial state s_0 , so policy optimality is evaluated w.r.t. $V^\rho(s_0)$. Every MDP has an optimal policy ρ^* . In discounted cumulative settings, there exists an optimal policy that is *Markovian*, i.e. ρ depends only on the current state, and it is deterministic (Puterman, 1994).

5.2 Techniques for finding an optimal policy

The algorithms used for solving the optimal control in MDPs are based on estimating the value function at all states. The two classical algorithms used to determine the optimal policy for a MDP are:

- **Policy iteration algorithm**

Once a policy ρ , has been improved using v_ρ to yield a better policy ρ^* , we can then compute $v_{\rho'}$ and improve it again to yield an even better ρ^{**} . We can thus obtain a sequence of monotonically improving policies and value functions:

$$\rho_0 \xrightarrow{E} v_{\rho_0} \xrightarrow{I} \rho_1 \xrightarrow{E} v_{\rho_1} \xrightarrow{I} \rho_2 \xrightarrow{E} \dots \xrightarrow{I} \rho_* \xrightarrow{E} v_*,$$

where \xrightarrow{E} denotes a policy *evaluation* and \xrightarrow{I} denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

- **Value iteration algorithm**

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_ρ occurs only in the limit. In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.

One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] = \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')], \end{aligned} \quad (5.12)$$

for all $s \in \mathcal{S}$.

Algorithm 5.1. Policy Iteration (using iterative policy evaluation)

1. Initialization
 $V(s) \in \mathbb{R}$ and $\rho(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
Loop:
 $\Delta \leftarrow 0$
Loop for each $s \in \mathcal{S}$:
 $v \leftarrow V(s)$
 $V(s) \leftarrow \sum_{s',r} p(s',r|s,\rho(s)) [r + \gamma V(s')]$
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
3. Policy Improvement
 $policy-stable \leftarrow true$
For each $s \in \mathcal{S}$:
 $old-action \leftarrow \rho(s)$
 $\rho(s) \leftarrow \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
If $old-action \neq \rho(s)$, then $policy-stable \leftarrow false$
If $policy-stable$, then **stop and return** $V \approx v_*$ and $\rho \approx \rho_*$: **else go to 2**

For arbitrary v_0 , the sequence v_k can be shown to converge to v_* under the same conditions that guarantee the existence of v_* . Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to ρ_* .

In practice, we stop once the value function changes by only a small amount in a sweep. The algorithm below shows this kind of termination condition.

Algorithm 5.2. Value Iteration

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation.
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
 $\Delta \leftarrow 0$
Loop for each $s \in \mathcal{S}$:
 $v \leftarrow V(s)$
 $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\rho \approx \rho_*$, such that:

$$\rho(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep.

In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (5.12) is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation.

5.3 Service Composition

The problem of service composition has been considered in the literature for over a decade, starting from seminal manual approaches, e.g., (Medjahed et al., 2003; Yang and Papazoglou, 2004; Cardoso and Sheth, 2004), which mainly focused on modeling issues as well as on automated discovery of services described making use of rich ontologies, to automatic ones based on planning, e.g., (Wu et al., 2003; Pistore et al., 2005) or on KR techniques, e.g., (McIlraith and Son, 2002), or on automated synthesis (Berardi et al., 2003; Hu and De Giacomo, 2013; De Giacomo et al., 2013). The reader interested in a survey of approaches can refer to (Hull, 2008; Su, 2008; De Giacomo et al., 2014).

Here we concentrate on the approach known in literature as the “Roman model” (Berardi et al., 2003): each available (i.e., to be used in the composition, therefore referred to as *component*) service is modeled as a finite-state machines (FSM), in which at each state, the service offers a certain set of actions, where each action changes the state of the service in some way. The designer is interested in generating a new service (referred to as *target*) from the set of existing services. The required service (the *requirement*) is specified using a FSM, too.

The computational problem is to see whether the requirement can be satisfied by properly orchestrating the work of the component services. That is, by building a scheduler (called the *orchestrator*) that will use actions provided by existing services to implement action request of the requirement. Thus, a new service is synthesized using existing services.

The Roman Model Now we describe more formally the Roman model and its components. A *service* is a tuple $\mathcal{S} = \langle \Sigma, \sigma_0, F, A, \delta \rangle$ where:

- Σ is the finite set of service states,
- $\sigma_0 \in \Sigma$ is the initial state,
- $F \subseteq \Sigma$ is the set of the service’s final state,
- A is the finite set of services’ actions,
- $\delta \subseteq \Sigma \times A \rightarrow \Sigma$ is the service’s deterministic and partial transition function.

We use the notation $\sigma \xrightarrow{a} \sigma'$ and $\sigma' \in \delta(\sigma, a)$ interchangeably when δ is clear from the context. We also write $A(\sigma)$ to denote $\{a \in A : \delta(\sigma, a) \text{ is defined}\}$.

A service can be seen as a software artifact, distributed and built on top of different technologies, that export a description of themselves, accessible to external clients and communicate through a commonly standard interface which enables interoperability. *Actions* in A denote interactions between service and clients. The behaviour of each available service is described in terms of a finite transition system that uses only actions from A .

A *history* h of a service \mathcal{S} is a sequence of alternating states and actions (necessarily ending with a state):

$$h = \sigma_0, a_1, \sigma_1, a_2, \dots, a_n, \sigma_n.$$

The available services are grouped into *community*, where they share a common set of actions Σ (the actions of the community). The *system service* of a community $\mathcal{C} = \{\mathcal{S}_\infty, \dots, \mathcal{S}_n\}$ is the service $\mathcal{Z} = (\Sigma_z, \delta_{z0}, F_z, A_z, \delta_z)$ such that:

- $\Sigma_z = \Sigma_1 \times \dots \times \Sigma_n$,
- $\sigma_{z0} = (\sigma_{10}, \dots, \sigma_{n0})$,
- $F_z = \{(\sigma_1, \dots, \sigma_n) \mid \sigma_i \in F_i, 1 \leq i \leq n\}$

- $A_z = A \times \{1, \dots, n\}$ is the set of pairs (a, i) formed by a shared action a and the index i of the service that executes it.
- $\sigma \xrightarrow{(a,i)} \sigma'$ iff, for $\sigma = (\sigma_1, \dots, \sigma_n)$ and $\sigma' = (\sigma'_1, \dots, \sigma'_n)$, it is the case that $\sigma_i \xrightarrow{a} \sigma'_i$ in σ_i and $\sigma_j = \sigma'_j$ for $j \neq i$.

Intuitively, \mathcal{Z} is the service stemming from the product of the asynchronous execution of the services in S . This is a virtual entity, i.e., without any actual counterpart, that offers a formal account of the evolution of the available services, when the community is seen as a whole. Note that in the transitions of Z , the service executing the corresponding action, is explicitly mentioned. Also $(a, i) \in A(\sigma)$ indicates that a can be executed by service i in the current state. We denote the set of system service histories by H_z .

Given a service community C , a *target service* \mathcal{T} is the desired service, described as a finite, deterministic transition system that uses action from A . The target service provides a formal characterization of a desired service that may not be available in the community. We denote the set of possible target service histories by H_t . Informally, the target represents a business process that one would like to offer to clients, where each state represents a decision point. At each state, the client is provided with a set of options to choose among, each corresponding to an action available in the state. Notice that typically the target service is not available. Further, the only entities able to execute actions, i.e., activities, are the available services.

Thus, one cannot build the target service by simply combining the actions of the target service, but has to resort to the available services, which impose constraints on the execution of actions, depending on the conversations they can actually carry out. An *orchestrator* is an entity that has the ability of scheduling services on a step-by-step basis and coordinate the community services so as to mimic the behavior of the target service. Differently put, the behavior obtained by coordinating the services should present no differences, from the client perspective, with the target service. Formally, an orchestrator for a community is a partial function γ :

$$\gamma : \Sigma_z \times A \rightarrow \{1, \dots, n\}$$

Intuitively, γ is a decision maker able to keep track of the way the services in the community have evolved up to a certain point, and that, in response to an incoming action request, returns the index of a service. The dynamics of the system is deterministic given the actions selected by the user. Hence, together with the orchestrator choice, it determines a system history. That is, an orchestrator defines a partial function from target-service histories to system histories, based on the (partial) mapping from system state and action to a service and the (partial) mapping from system state, action, and service, to the next system state.

We denote this mapping by τ . More formally, $\tau : H_t \rightarrow H_z$ is defined inductively as follows:

- $\tau(\sigma_{t0}) = \sigma_{z0}$.
- Let $\tau(h_t) = h_z$, and let s_t, s_z denote the last states, respectively, in h_t, h_z . Then, τ is also defined on $h_t \cdot a \cdot s'_t$ provided: $a \in A(s_t)$ and $s'_t = \delta_t(s_t, a)$, and that γ is defined on (s_z, a) , and $(a, \gamma(s_z, a)) \in A(s_z)$. That is, provided the orchestrator function is defined on s_z and a , assigning some value i , and δ_z is well defined on (s_z, i) , we have $\tau(h_t \cdot a \cdot s'_t) = h_z \cdot a \cdot \delta_z(s_z, (a, i))$.
- Otherwise, $\tau(h_t \cdot a \cdot s'_t)$ is undefined.

If $\tau(h_t)$ is well-defined, we say that target history h_t is *realizable* by the orchestrator. The orchestrator γ is said to *realize* a target service \mathcal{T} if it realizes all histories of \mathcal{T} . In this case, γ is also called a *composition* of \mathcal{T} . A target \mathcal{T} is *realizable* if there exists an orchestrator that realizes it.

The problem of service composition is known to be EXPTIME-complete, in fact exponential on the number of the available services (Berardi et al., 2003; Muscholl and Walukiewicz, 2008) and techniques based on model checking, simulation, and LTL synthesis are available (De Giacomo et al., 2014). Also, several variants have been studied, including the case of nondeterministic (i.e., partially controllable but fully observable) available services (De Giacomo et al., 2013).

Example 5.1. *To describe this setting in terms of the framework previously discussed, we identify the following correspondences:*

- *the community ontology is simply Σ ;*
- *the available services are the actual services in the community;*
- *the mapping from the available services to the community ontology is represented by the transition systems that describe the available services (built from community actions);*
- *the client request is the target service (again, built from community actions).*

5.4 Stochastic Service Composition

Unfortunately, it is not always possible to synthesize a service that fully conforms with the requirement specification. This zero-one situation, where we can either synthesize a perfect solution or fail, often is not very applicable. Rather than returning no answer, we may want notion of the “best-possible” solution. A model with this notion has been developed in (Brafman et al., 2017), where the authors discuss and elaborate upon a probabilistic model for the service composition problem, first presented in (Yadav and Sardiña, 2011). In this model, an optimal solution can be found by solving an appropriate probabilistic planning problem (e.g. an MDP) derived from the services and requirement specifications.

Specifically, it is natural to make the requirement probabilistic, associating a probability with each action choice in each state. This probability captures how likely the user is to request the action in that state. Such information can be, initially, supplied by the designer, but can also be learned in the course of service operation in order to adapt the composition to user behavior. To model the value and likelihood of requests, we augment the target service model with two additional elements. P_t will be a distribution over the actions given the state. $P_t(s, a)$ is the likelihood that a user will request a in target state s .

Technically, $P_t(s)$ returns a distribution over the actions, or the empty set, when s is a terminal state on which no actions are possible. R_t is the reward function, associating a non-negative reward with the ability to provide the action requested by a user. $R_t(s, a)$ is the value we associate with being able to provide action a in state s .

Definition 5.2. *Formally, a target service is $\mathcal{T} = \langle \Sigma_t, \sigma_{t0}, F_t, A, \delta_t, P_t, R_t \rangle$, where:*

- $\Sigma_t, \delta_{t0}, F_t, A, \delta_t$ are defined as before,
- $P_t : \Sigma_t \rightarrow \pi(A) \cup \emptyset$ is the action distribution function,
- $R_t : \Sigma_t \times A \rightarrow \mathbb{R}$ is the reward function.

P_t induces a probability density function over the set of all infinite target histories, which we will denote by P_∞ . (This follows by the Ionescu-Tulcea extension theorem). R_t can be used to associate a value with every infinite history.

Definition 5.3. *The standard definition of the value of a history h_t , which we adopt here, is that of the sum of discounted rewards:*

$$v(\sigma_0, a_1, \sigma_1, \dots) = \sum_{i=0}^{\infty} \lambda^i R_t(\sigma_i, a_{i+1}), \quad (5.13)$$

where $0 < \lambda < 1$ is the discount factor.

The discount factor can be viewed as measuring the factor by which the value of rewards is reduced as time progresses, capturing the intuition that the same reward now is better than in the future.

Definition 5.4. Given the above, we can define the expected value of an orchestrator γ to be:

$$v(\gamma) = \mathbb{E}_{h_t \sim P_\infty}(v(h_t), \text{real}(\gamma, h_t)) \quad (5.14)$$

where $\text{real}(\gamma, h_t)$ is 1 if h_t is realizable in γ , and 0 otherwise. That is, $v(\gamma)$ is the expected value of histories realizable in γ .

Definition 5.5. Finally, we define an optimal orchestrator to be:

$$\gamma^* = \arg \max_{\gamma'} v(\gamma') \quad (5.15)$$

We have the following theorem:

Theorem 5.1 (Brafman, De Giacomo, Mecella, and Sardina (2017)). If the target is realizable and every target history has strictly positive value then γ realizes the target iff it is an optimal orchestrator.

Proof. We prove the two directions of the equivalence separately.

- Necessity condition: \Rightarrow . We want to prove that if the orchestrator γ realizes the target is optimal. The thesis stems from the fact that if the set of histories realizable using orchestrator γ contains the set realizable using orchestrator γ' , then $v(\gamma) \geq v(\gamma')$.
- Sufficiency condition: \Leftarrow . We want to prove that if the orchestrator is optimal, then it realizes the target. We prove the statement by proving its contrapositive, i.e. any orchestrator that does not realize some history, is non-optimal. Such thesis stems from the fact that if, in addition, the set of histories realizable by γ but not by γ' has positive probability, then $v(\gamma) > v(\gamma')$. Now, if h_t is not realizable by γ' , there exists a point in h_t where γ' does not assign the required action to a service that can supply it. Thus, any history that extends the corresponding prefix of h_t is not realisable, and the set of such histories has non-zero probability. Since we assume all histories have positive value, we obtain the desired result.

□

In other words, if it is possible to realize the target requirement, then any orchestrator realizing it is optimal, and any orchestrator that does not realize some history, is non-optimal. The importance of this new model is that we now have a clear notion of an optimal orchestrator that works even when the target service is not fully realizable, and this notion is clearly an extension of the standard notion, coinciding with it when the service is realizable by some orchestrator. An optimal controller is simply one that is able to handle more (in expectation) valued histories.

We now explain how to solve the above model by formulating an appropriate MDP.

Definition 5.6. The composition MDP is a function of the system service and the target service as follows $\mathcal{M}(\mathcal{Z}, \mathcal{T})$ where:

- $S_{\mathcal{M}} = \Sigma_{\mathcal{Z}} \times \Sigma_{\mathcal{T}} \times A \cup \{s_{\mathcal{M}0}\}$
- $A_{\mathcal{M}} = \{a_{\mathcal{M}0}, 1, \dots, n\}$
- $T_{\mathcal{M}}(s_{\mathcal{M}0}, a_{\mathcal{M}0}, (\sigma_{z0}, \sigma_{t0}, a)) = P_t(\sigma_{t0}, a)$

- $T_{\mathcal{M}}((\sigma_z, \sigma_t, a), i, (\sigma'_z, \sigma'_t, a')) = P_t(\sigma'_t, a')$, if $\sigma_z \xrightarrow{(a,i)} \sigma'_z$ and $\sigma_t \xrightarrow{a} \sigma'_t$
- $R((\sigma_z, \sigma_t, a), i) = R_t(\sigma_t, a)$ if $(a, i) \in A(\sigma_z)$ and 0 otherwise.

It can be shown that:

Theorem 5.2 (Brafman, De Giacomo, Mecella, and Sardina (2017)). *Let π be an optimal policy for $\mathcal{M}(\mathcal{Z}, \mathcal{T})$. Then, the orchestrator γ such that $\gamma((\sigma_z, \sigma_t), a) = \pi(\sigma_z, \sigma_t, a)$ is an optimal orchestrator.*

Proof. The result follows from the fact that there is a one-to-one correspondence between orchestrators and policies for $\mathcal{M}(\mathcal{Z}, \mathcal{T})$, via the relationship: $\gamma((\sigma_z, \sigma_t), a) = \rho(\sigma_z, \sigma_t, a)$, and the fact that the value of policy ρ so defined equals $v(\gamma)$. \square

Chapter 6

Service Composition with Stochastic Services

In this chapter we extend the stochastic model proposed in (Brafman et al., 2017) where not only the target service but also the services behave stochastically. The chapter is structured as follows:

- In **Section 6.1**: we provide an introduction about all mathematical concepts used in service composition with stochastic services;
- In **Section 6.2**: we explain how computing an optimal orchestrator with this extension;
- In **Section 6.3**: we define an example of how the definitions defined above can be applied.

6.1 Introduction

Each stochastic transition over the services is associated to a reward signal; hence, the orchestrator can take into account the reward associated to delegations to services. Such a model has greater expressivity as it allows to model services with richer features, e.g. a service action might be associated to a cost of execution (i.e. a negative reward), that might induce the orchestrator to prefer another service. Obviously, this is a more general definition in which the previous formalization is a special case.

Definition 6.1. A stochastic service is a tuple $\tilde{\mathcal{S}} = \langle \Sigma_s, \sigma_{s0}, F_s, A, P_s, R_s \rangle$, where $\Sigma_s, \sigma_{s0}, F_s, A$ are defined as before, $P_s : \Sigma_s \times A \rightarrow \text{Prob}(\Sigma_s)$ is the transition function, and $R_s : \Sigma_s \times A \rightarrow \mathbb{R}$ is the reward function. In short words, the stochastic service is the stochastic variant of the service we defined in the preliminaries. We also define $\delta_s \subseteq \Sigma_s \times A \times \Sigma_s$ to be the transition relation of $\tilde{\mathcal{S}}$, defined as follows:

$$\delta_s = \{(\sigma, a, \sigma') \mid P(\sigma' \mid \sigma, a) > 0\}$$

The definition of a community is the same in the non-stochastic case, i.e. a set of (stochastic) services.

Definition 6.2. The stochastic system service $\tilde{\mathcal{Z}}$ of a community $\tilde{\mathcal{C}}$ of stochastic services $\{\tilde{\mathcal{S}}_1, \dots, \tilde{\mathcal{S}}_n\}$ is a stochastic service where $\tilde{\mathcal{Z}} = \langle \Sigma_z, \sigma_{z0}, F_z, A, P_z, R_z \rangle$ are defined as follows:

- $\Sigma_z = \Sigma_1 \times \dots \times \Sigma_n$,
- $\sigma_{z0} = (\sigma_{10}, \dots, \sigma_{n0})$,
- $F_z = \{(\sigma_1, \dots, \sigma_n) \mid \sigma_i \in F_i, 1 \leq i \leq n\}$

- $A_z = A \times \{1, \dots, n\}$ is the set of pairs (a, i) formed by a shared action a and the index i of the service that executes it.
- $P_z(\sigma' | \sigma, (a, i)) = P(\sigma'_i | \sigma_i, a)$, for $\sigma = (\sigma_1 \dots \sigma_n)$, $\sigma' = (\sigma'_1 \dots \sigma'_n)$ and $a \in A_i(\sigma_i)$, with $\sigma_i \in \Sigma_i$ and $\sigma_j = \sigma'_j$ for $j \neq i$.
- $R_z(\sigma, (a, i)) = R_i(\sigma_i, a)$ for $\sigma \in \Sigma_z$, $a \in A_i(\sigma_i)$.

We define the set of joint histories of the target and the system service as $H_{t,z} = \Sigma_t \times \Sigma_z \times (A \times \Sigma_t \times \Sigma_z)^*$. A joint history $h_{t,z} = \sigma_{t,0}\sigma_{z,0}a_1\sigma_{t,1}\sigma_{z,1}a_2\dots$ is an element of $H_{t,z}$. The projection of $h_{t,z}$ over the target (system) actions is $\pi_t(h_{t,z}) = h_t$ ($\pi_z(h_{t,z}) = h_z$). When it is clear from the context, we drop the subscript t, z to lighten the notation.

In order to proceed, we need a notion of value over $H_{t,z}$. Crucially, since the stochasticity comes also from the services, the orchestrator *does* affect the probability of an history $h_{t,z}$. Note how this is in contrast with the previous model, in which the value of a history was defined only on target histories, whose probability was not influenced by the orchestrator. In this context, we define the orchestrator γ :

$$\gamma : \Sigma_t \times \Sigma_z \times A \rightarrow \{1, \dots, n\}$$

as a mapping from a state of the target-system service and user action $(\sigma_t, \sigma_z, a) \in \Sigma_t \times \Sigma_z \times A$ to the index $j \in \{1, \dots, n\}$ of the service that must handle it.

Differently from the previous case, the dynamics of the system is *not* deterministic even when the actions selected by the user are given, because given an user action a and an assignment by the orchestrator to service j , σ_j has many possible successors σ'_j , where σ'_j is such that $P_j(\sigma'_j | \sigma_j, a) > 0$. Therefore, in general, there are *several* system histories associated to a given target history.

An orchestrator defines a partial function from target-service histories to sets of system histories, based on the (partial) mapping from system state and action to a service and the (partial) mapping from system state, action, and service, to the next system states. We denote this mapping by $\tilde{\tau}_\gamma$. More formally, given an orchestrator γ , $\tilde{\tau}_\gamma : H_t \rightarrow 2^{H_z}$ is defined inductively as follows:

- $\tilde{\tau}_\gamma(\sigma_{t0}) = \{\sigma_{z0}\}$.
- Let $\tilde{\tau}_\gamma(h_t) = \{h_{z1}, \dots, h_{zn}\}$. Let σ_t and $\sigma_{z1}, \dots, \sigma_{zn}$ denote the last states of the target history and the system histories, respectively. Then, $\tilde{\tau}$ is also defined on $h_t a \sigma'_t$ as follows:

$$\tilde{\tau}_\gamma(h_t a \sigma'_t) = \{h_{zk} a \sigma'_{zk} \mid h_{zk} \in \tilde{\tau}_\gamma(h_t), j = \gamma(\sigma_t, \sigma_{zk}, a), (\sigma_{zk}, \langle a, j \rangle, \sigma'_{zk}) \in \delta_z\}$$

provided: $a \in A(\sigma_t)$, $\sigma'_t \in \delta_t(\sigma_t, a)$, $j = \gamma(\sigma_t, \sigma_{zk}, a)$, and $\langle a, j \rangle \in A(\sigma_{zk})$.

At this point, the definitions of realizability are analogous to the previous case:

- If $\tilde{\tau}_\gamma(h_t)$ is well defined and $\tilde{\tau}_\gamma(h_t) \neq \emptyset$, we say that target history h_t is *realizable* by the orchestrator γ .
- The orchestrator γ is said to *realize* a target service $\tilde{\mathcal{Z}}$ if it realizes all histories of $\tilde{\mathcal{Z}}$.

The probability of a (joint) history $h = \sigma_{t0}\sigma_{z0}\langle a_1, j_1 \rangle\sigma_{t1}\sigma_{z1}\langle a_2, j_2 \rangle\dots$ under orchestrator γ is given by:

$$P_\gamma(h) = \prod_{i=0}^{|h|} P_t(\sigma_{t,i}, a_{i+1}) P_z(\sigma_{z,i+1} | \sigma_{z,i}, \langle a_{i+1}, \gamma(\sigma_{t,i}, \sigma_{z,i}, a_{i+1}) \rangle) \quad (6.1)$$

Intuitively, at every step, we take into account the probability, determined by P_t , that the user does action a_{i+1} in the target state $\sigma_{t,i}$, in conjunction with the probability, determined by

P_z , that the system service does the transition $\sigma_{z,i} \xrightarrow{(a_{i+1},j)} \sigma'_{z,i+1}$, where j is the choice of the orchestrator at step i under orchestrator γ , i.e. $j = \gamma(\sigma_{t,i}, \sigma_{z,i}, a_{i+1})$.

The value of a joint history under orchestrator γ is the sum of discounted rewards, both from the target and the system services:

$$v_\gamma(h) = \sum_{i=0}^{|h|} \lambda^i \left(R_t(\sigma_{t,i}, a_{i+1}) + R_z(\sigma_{z,i}, \langle a_{i+1}, \gamma(\sigma_{t,i}, \sigma_{z,i}, a_{i+1}) \rangle) \right) \quad (6.2)$$

Intuitively, we take into account both the reward that comes from the execution of action a_{i+1} in the target service, but also the reward associated to the execution of that action in service j chosen by orchestrator γ .

We can refine Equation 6.2 by considering the two components separately:

$$v_\gamma(h) = \left(\sum_{i=0}^{|h|} \lambda^i R_t(\sigma_{t,i}, a_{i+1}) \right) + \left(\sum_{i=0}^{|h|} \lambda^i R_z(\sigma_{z,i}, \langle a_{i+1}, \gamma(\sigma_{t,i}, \sigma_{z,i}, a_{i+1}) \rangle) \right) \quad (6.3)$$

$$= v_{\gamma,\mathcal{T}}(h) + v_{\gamma,\mathcal{Z}}(h) \quad (6.4)$$

Note that $v_{\gamma,\mathcal{T}}$ depends only on the projected target history, whereas $v_{\gamma,\mathcal{Z}}$ depends on the joint target-system history.

Now we can define the expected value of an orchestrator to be:

$$v(\gamma) = \mathbb{E}_{h_{t,z} \sim P_\gamma} [v_\gamma(h_{t,z}) \cdot \text{realizable}(\gamma, \pi_t(h_{t,z}))] \quad (6.5)$$

where $\text{realizable}(\gamma, \pi_t(h_{t,z}))$ is 1 if $h_t = \pi_t(h_{t,z})$ is realizable in γ , and 0 otherwise. That is, $v(\gamma)$ is the expected value of histories realizable in γ .

Similarly to Equation 6.3, thanks to the linearity of the expectation, we can separate the two contributions to $v(\gamma)$: one coming from the target's rewards, and the one coming from the services' rewards:

$$v(\gamma) = \mathbb{E}_{h_{t,z} \sim P_\gamma} [(v_{\gamma,\mathcal{T}}(h_t) + v_{\gamma,\mathcal{Z}}(h_{t,z})) \cdot \text{realizable}(\gamma, h_t)] \quad (6.6)$$

$$= \mathbb{E}_{h_{t,z} \sim P_\gamma} [v_{\gamma,\mathcal{T}}(h_t) \cdot \text{realizable}(\gamma, h_t)] + \mathbb{E}_{h_{t,z} \sim P_\gamma} [v_{\gamma,\mathcal{Z}}(h_{t,z}) \cdot \text{realizable}(\gamma, h_t)] \quad (6.7)$$

$$= v_{\mathcal{T}}(\gamma) + v_{\mathcal{Z}}(\gamma) \quad (6.8)$$

Where step 6.6 is by definition of $v(\gamma)$ and step 6.7 is by linearity of the expectation. Finally, we define an *optimal orchestrator* to be $\gamma = \arg \max_{\gamma'} v(\gamma')$.

Our goal now is to find a relationship between the concept of realizability and the one of optimality. Before proceeding, we make the following assumption, without loss of generality:

Assumption 6.1 (Non-negative rewards). *Rewards are non-negative.*

Assumption 6.1 can be relaxed by making sure all the rewards are non-negative by shifting positively all the rewards by $\max(-R_{min}, 0)$, where R_{min} is the minimum reward across both the target and the system service.

Intuitively, such transformation is policy-invariant, as the $\arg \max_a$ operation, the operation that computes the optimal action a to take in any state s from the action-value function $Q(s, a)$, is invariant with respect to a constant value. However, Assumption 6.1 will make the proofs easier. This observation is formally proven by Lemma 6.5 (see below).

We now state other assumptions to show the relationship between realizability and optimality.

Assumption 6.2 (Target Realizability). *The target is realizable.*

Assumption 6.3 (Strict positive target history values). *Every target history has strictly positive value, i.e. $v_{\gamma, \mathcal{T}}(h_t) > 0$ for all h_t .*

Assumption 6.4 (Target's rewards dominance). *For any target reward $r_{\mathcal{T}}$ and any service reward $r_{\mathcal{Z}}$, we have that $r_{\mathcal{T}} > r_{\mathcal{Z}}$.*

Intuitively, we need Assumption 6.4 because the orchestrator must never be "deceived" by the services' rewards; the real objective is always to maximize the expected discounted sum of target's rewards, and secondarily the services' ones.

Now we prove Lemma 6.5:

Lemma 6.5. *Let $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$ be an MDP, and let $\mathcal{M}' = \langle S, A, T, R', \gamma \rangle$ another MDP where R' is defined as:*

$$R'(s, a) = R(s, a) + c$$

where $c \in \mathbb{R}$. Then, for all policies ρ on \mathcal{M} , ρ is optimal on \mathcal{M} iff ρ is optimal on \mathcal{M}'

Proof. The relationship between $v_{\rho}(s)$ and $v'_{\rho}(s)$ is determined by the following derivation:

$$v_{\rho}(s) = \mathbb{E}_{\rho}[G_t \mid S_t = s] \quad (6.9)$$

$$= \mathbb{E}_{\rho}\left[\sum_{k=0}^{\infty} \gamma^k R_{k+1} \mid S_t = s\right] \quad (6.10)$$

$$= \mathbb{E}_{\rho}\left[\sum_{k=0}^{\infty} \gamma^k (R'_{k+1} - c) \mid S_t = s\right] \quad (6.11)$$

$$= \mathbb{E}_{\rho}\left[\sum_{k=0}^{\infty} \gamma^k R'_{k+1} \mid S_t = s\right] - \mathbb{E}_{\rho}\left[c \sum_{k=0}^{\infty} \gamma^k \mid S_t = s\right] \quad (6.12)$$

$$= v'_{\rho}(s) - \frac{c}{1-\gamma} \quad (6.13)$$

where step 6.9 is by definition of $v^*(s)$; step 6.10 is by definition of G_t ; step 6.11 is by definition of $R'(s, a)$; step 6.12 by the linearity of the expectation; and step 6.13 by the convergence of the geometric sum ad infinitum.

Therefore, the partial order $\rho \geq \rho'$ between two policies ρ, ρ' is the same both on \mathcal{M} and \mathcal{M}' , i.e.

$$\begin{aligned} v_{\rho}(s) &\geq v_{\rho'}(s) \\ \iff \\ v'_{\rho}(s) - \frac{c}{1-\gamma} &\geq v'_{\rho'}(s) - \frac{c}{1-\gamma} \\ \iff \\ v'_{\rho}(s) &\geq v'_{\rho'}(s) \end{aligned}$$

The thesis follows by definition of optimal policy, because ρ^* is an optimal policy for \mathcal{M} iff $\forall \rho'. \rho^* \geq \rho'$ on \mathcal{M} , but from the above we also have that it is true iff $\forall \rho'. \rho^* \geq \rho'$ on \mathcal{M}' . \square

Therefore, we can choose $c = \max(-R_{\min}, 0)$ whilst preserving the same set of optimal policies from \mathcal{M} with (possibly) negative rewards to \mathcal{M}' with non-negative rewards.

Theorem 6.6. *If the target is realizable (Assumption 6.2), every history has strictly positive value (Assumption 6.3), and the target's rewards are always greater than services' rewards (Assumption 6.4), then if γ is an optimal orchestrator, γ realizes the target.*

Proof. The proof is very similar to Theorem 5.1. It is crucial to observe that all the histories have positive value thanks to Assumption 6.1, which as stated above is without loss of generality thanks to Lemma 6.5.

We prove the thesis by proving its contrapositive: if γ does not realize the target, then γ is not an optimal orchestrator. Indeed, assume the orchestrator γ does not realize a target history h_t , there exists a point in h_t where γ does not assign the required action to a service that can supply it. Thus, any history that extends the corresponding prefix of h_t is not realisable, and the set of such histories has non-zero probability. Since each realized target history has a strictly positive value, $v_{\gamma,\mathcal{T}}(h_t)$, and therefore $v_{\mathcal{T}}(\gamma)$, would be better if γ had chosen a better action at that point that would make h_t and its extensions realizable. Note that the value contribution from the system, i.e. $v_{\gamma,\mathcal{Z}}(h_t)$, can either increase and remain the same, in which case we are done, or can decrease, in which case the new overall value might be lower than before. However, to handle this case, assume we first maximize the target's value component, regardless of the system's value contribution. Then the orchestrator can switch the services with the most rewarding ones without neglecting the target's rewards, and since we assumed that any target's reward is grater than any service's reward, by assumption 6.4, this change will always be optimal, whilst keeping γ realizing the target.

□

Note that the other direction of the implication does not hold, i.e. if γ realizes the target then it is not necessarily the case that γ is an optimal orchestrator. The reason is that the orchestrator may still realize the target but choosing among two equivalent services the less rewarding one, whereas the optimal orchestrator would have chosen the other service.

6.2 Computing an Optimal Orchestrator

We now explain how to solve the above model by formulating an appropriate MDP.

Definition 6.3. *The composition MDP is a function of the system service and the target service as follows $\tilde{\mathcal{M}}(\tilde{\mathcal{Z}}, \tilde{\mathcal{T}}) = \langle S_{\tilde{\mathcal{M}}}, A_{\tilde{\mathcal{M}}}, T_{\tilde{\mathcal{M}}}, R_{\tilde{\mathcal{M}}} \rangle$:*

- $S_{\tilde{\mathcal{M}}} = \Sigma_{\tilde{\mathcal{Z}}} \times \Sigma_{\tilde{\mathcal{T}}} \times A \cup \{s_{\mathcal{M}0}\}$
- $A_{\tilde{\mathcal{M}}} = \{a_{\mathcal{M}0}, 1, \dots, n\}$
- $T_{\tilde{\mathcal{M}}}(s_{\mathcal{M}0}, a_{\mathcal{M}0}, (\sigma_z, \sigma_t, a)) = P_t(\sigma_t, a)$
- $T_{\tilde{\mathcal{M}}}((\sigma_z, \sigma_t, a), i, (\sigma'_z, \sigma'_t, a')) = P_t(\sigma'_t, a') \cdot P_z(\sigma'_z \mid \sigma_z, \langle a, i \rangle)$, if $P_z(\sigma'_z \mid \sigma_z, \langle a, i \rangle) > 0$ and $\sigma_t \xrightarrow{a} \sigma'_t$
- $R_{\tilde{\mathcal{M}}}((\sigma_z, \sigma_t, a), i) = R_t(\sigma_t, a) + R_z(\sigma_z, \langle a, i \rangle)$, if $(a, i) \in A(\sigma_z)$ and 0 otherwise.

This definition is pretty similar to the construction proposed in (Brafman et al., 2017), with the difference that now, in the transition function, we need to take into account also the probability of transitioning to the system successor state σ'_z from σ_z doing the system action $\langle a, i \rangle$, i.e. $P_z(\sigma'_z \mid \sigma_z, \langle a, i \rangle)$. Moreover, in the reward function, we need to take into account also the reward observed from doing system action $\langle a, i \rangle$ in σ_z , and sum it to the reward signal coming from the target.

Theorem 6.7. *Let ρ be an optimal policy for $\tilde{\mathcal{M}}(\tilde{\mathcal{Z}}, \tilde{\mathcal{T}})$. Then, the orchestrator γ such that $\gamma(\sigma_z, \sigma_t, a) = \rho(\langle \sigma_z, \sigma_t, a \rangle)$ is an optimal orchestrator.*

Proof. To see this, consider $v_\rho(s)$ for any policy ρ of $\tilde{\mathcal{M}}$. By definition, $v_\rho(s)$ is the expected total return starting from s and acting by following ρ . Consider the orchestrator γ associated to π , i.e. $\gamma(\sigma_z, \sigma_t, a) = \pi(\langle \sigma_z, \sigma_t, a \rangle)$. We use $\tilde{\sigma}$ as a shorthand for (σ_z, σ_t, a) . We have:

$$V_\rho(\tilde{\sigma}) = \mathbb{E}_{h \sim \tilde{\mathcal{M}}, \pi} \left[\sum_i \lambda^i R_{\tilde{\mathcal{M}}}(\tilde{\sigma}_i, \pi(\tilde{\sigma}_i)) \right] \quad (6.14)$$

$$= \mathbb{E}_{h \sim \tilde{\mathcal{M}}, \pi} \left[\sum_i \lambda^i \left(R_t(\sigma_t, a) + R_z(\sigma_z, \langle a, \pi(\tilde{\sigma}_i) \rangle) \right) \right] \quad (6.15)$$

$$= \mathbb{E}_{h_{t,z} \sim P_\gamma} \left[v_\gamma(h_{t,z}) \cdot \text{realizable}(\gamma, h_t) \right] \quad (6.16)$$

Where step 6.14 is by definition of V , step 6.15 is by definition of $R_{\tilde{\mathcal{M}}}$, and step 6.16 is because from h we can recover $h_{t,z}$, $P_\gamma = T_{\tilde{\mathcal{M}}}^\rho$ by construction, and target histories that are not realizable are rewarded with 0 by construction. Since we proved that $v_\rho = v(\gamma)$ where γ is computed from ρ , for all ρ , we have the thesis. \square

6.3 Examples

According to the model describing above, we provide two different examples: the first is a very simple one, while the second is the use cases of the thesis.

6.3.1 Ceramics Production

The example that we provide is about the process of producing ceramics, as we explain in detail later in section 4.4. In particular we have the following services:

- (a) `provisioning`
- (b) `moulding`
- (c) `drying`
- (d) `first_baking`
- (e) `enamelling`
- (f) `painting_by_robot`
- (g) `painting_by_human`
- (h) `second_baking`
- (i) `shipping`

We can imagine that at the end of each step we have a quality control check. The depicted process is a linear process, while the target service have for each state a single action (with action probability 1.0) and a unitary (or zero) reward. The complexity is then moved to the component services. A stochastic service (for example provisioning service), remanding the definition 6.1 can be defined as follow:

- $\Sigma_s = \{\text{available}\}$
- $\sigma_{s0} = \{\text{available}\}$
- $F_s = \{\text{available}\}$
- $A = \{\text{provisioning}\}$
- $P_s(\text{available} \mid \text{available}, \text{provisioning}) = 1.0$

- $R_s(\text{available} \mid \text{provisioning}) = -c_t$

Where c_t is the cost of the task execution.

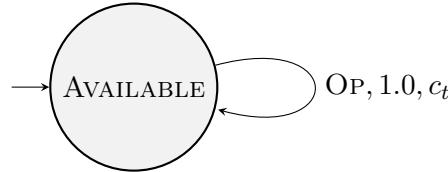


Figure 6.1. A prototype of the service's MDP we consider to model simple services. Such services have a single state and a self-loop transition with the OP action. The transition is associated to a cost $c_{i,t}$ of performing the task.

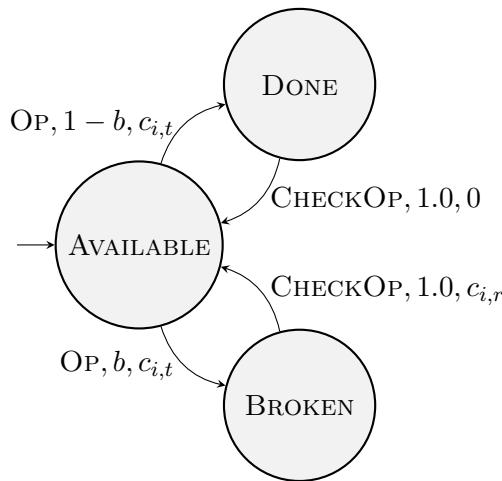


Figure 6.2. A prototype of the service's MDP we consider to model complex services (i.e. services that can break). It starts from the AVAILABLE state, and waits for the operation OP. b is the breaking probability, $c_{i,t}, c_{i,t} < 0$, is the cost of completing the task t on service i , and $c_{i,r}, c_{i,r} < 0$, is the repair cost for service i . When the action OP is executed, the system, with probability b , goes to the BROKEN state, which models the case when the machine gets broken after the action execution, and with probability $1 - b$ goes to the DONE state, which models the case when the task succeeded. The action CHECKOP is assumed to be executed by the target right after OP in order to make the service available again, and in particular, to force the repairing in case the service is the BROKEN state.

Another stochastic service (for example the moulding service) can be defined as (b is the broken probability) :

- $\Sigma_s = \{\text{available}, \text{broken}, \text{done}\}$
- $\sigma_{s0} = \{\text{available}\}$
- $F_s = \{\text{available}\}$
- $A = \{\text{moulding}, \text{check_moulding}\}$
- P_s :
 - $P_s(\text{done} \mid \text{available}, \text{moulding}) = 1 - b$
 - $P_s(\text{broken} \mid \text{available}, \text{moulding}) = b$
 - $P_s(\text{available} \mid \text{done}, \text{check_moulding}) = 1.0$

- $P_s(\text{available} \mid \text{broken}, \text{check_moulding}) = 1.0$
- R_s :
 - $R_s(\text{available}, \text{moulding}) = -c_t$
 - $R_s(\text{done}, \text{check_moulding}) = 0.0$
 - $R_s(\text{broken}, \text{check_moulding}) = -c_r$

Where c_t is the cost of the task execution, and c_r is the cost of repair.

The stochastic system service of a community of stochastic services remanding the definition 6.2 is:

- $\Sigma_z = \Sigma_a \times \Sigma_b \times \dots \times \Sigma_i$
- $\sigma_{z0} = \langle \text{available}_a, \text{available}_b, \dots, \text{available}_i \rangle$
- $F_z = \{\sigma_{z0}\}$
- $A = \{\langle \text{provisioning}, a \rangle, \langle \text{moulding}, b \rangle, \langle \text{check_moulding}, b \rangle, \dots, \langle \text{shipping}, i \rangle\}$
- P_z and R_z :
 - For all simple services actions a_i , $i = 1, \dots, m$, $P_z(\sigma \mid \sigma, a_i) = 1.0$ and $R_z(\sigma, a_i) = -c_{i,t}$, i.e., the execution remains in the same state with cost set to the cost of the task according to the service being executed.
 - For all complex services $i = m, \dots, n$ ($m < n$):
 - * If action a_i is of type $\langle op, i \rangle$, then $P_z(\sigma' \mid \sigma, a_i) = 1 - b$ with $\sigma = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_n \rangle$, $\sigma_i = \text{available}_i$, if $\sigma' = \langle \sigma_0, \dots, \text{done}_i, \dots, \sigma_n \rangle$, else $P_z(\sigma' \mid \sigma, a_i) = b$ and $\sigma' = \langle \sigma_0, \dots, \text{broken}_i, \dots, \sigma_n \rangle$; in both cases, $R(\sigma, a_i) = c_{i,t}$;
 - * If action a_i is of type $\langle check_op, i \rangle$, then $P_z(\sigma' \mid \sigma, a_i) = 1.0$ with $\sigma = \langle \sigma_0, \dots, \sigma_i, \dots, \sigma_n \rangle$, $\sigma_i \in \{\text{done}, \text{broken}\}$, and $\sigma' = \langle \sigma_0, \dots, \text{available}_i, \dots, \sigma_n \rangle$. Moreover, $R(\sigma, a_i) = 0$ if $\sigma_i = \text{done}$, otherwise $R(\sigma, a_i) = -c_{i,r}$ if $\sigma_i = \text{broken}$.

The target service, remanding the definition 5.2:

- $\Sigma_t = \{s_0, s_1, \dots, s_m\}$, one state for each action to be done ($m = 13$);
- $\sigma_{t0} = \{s_0\}$, the initial state;
- $F_t = \{s_0\}$ (only the initial state is accepting)
- $A = \{a \mid a \in \text{actions}\}$
- $P_t(s_{(i+1 \bmod m+1)} \mid s_i, a_i) = 1.0$ for $i = 0, \dots, m$
- $R_t(s_i, a_i) = 0.0$ if $i \neq m$, else $R_t(s_i, a_i) = 1.0$.

In Figure 6.3 you can see a picture of the target service described above.

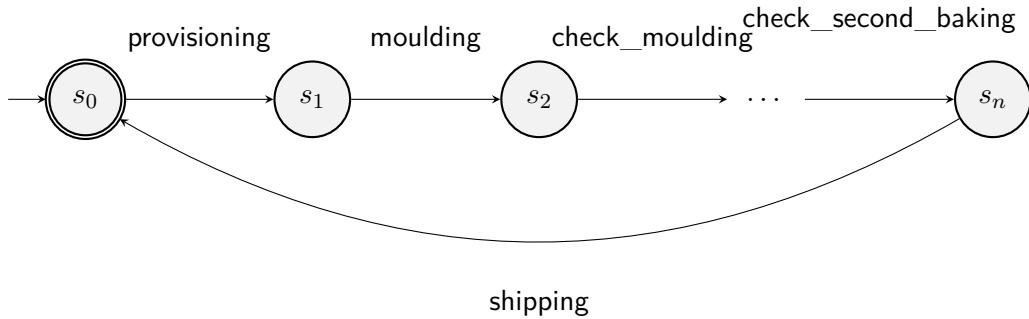


Figure 6.3. The state machine of the target. Some steps are omitted.

The composition MDP is built according to Definition 6.3. An execution of the MDP looks like as follows:

- For simple services, the orchestrator selects one among the available services according to the cost of the task execution;
- For complex services, on the execution of the task, there is a nondeterminism due to the breaking probability; the execution can lead either to the done state of the service with probability $1-b$ or to the broken state of the service with probability b . By construction of the target, the next action the orchestrator must dispatch is the *check* of the operation just executed; this will make again available the previously chosen service, eventually paying a cost due to repair in case the service was broken.

Figure 6.4 depicts the initial portion of the composition MDP.

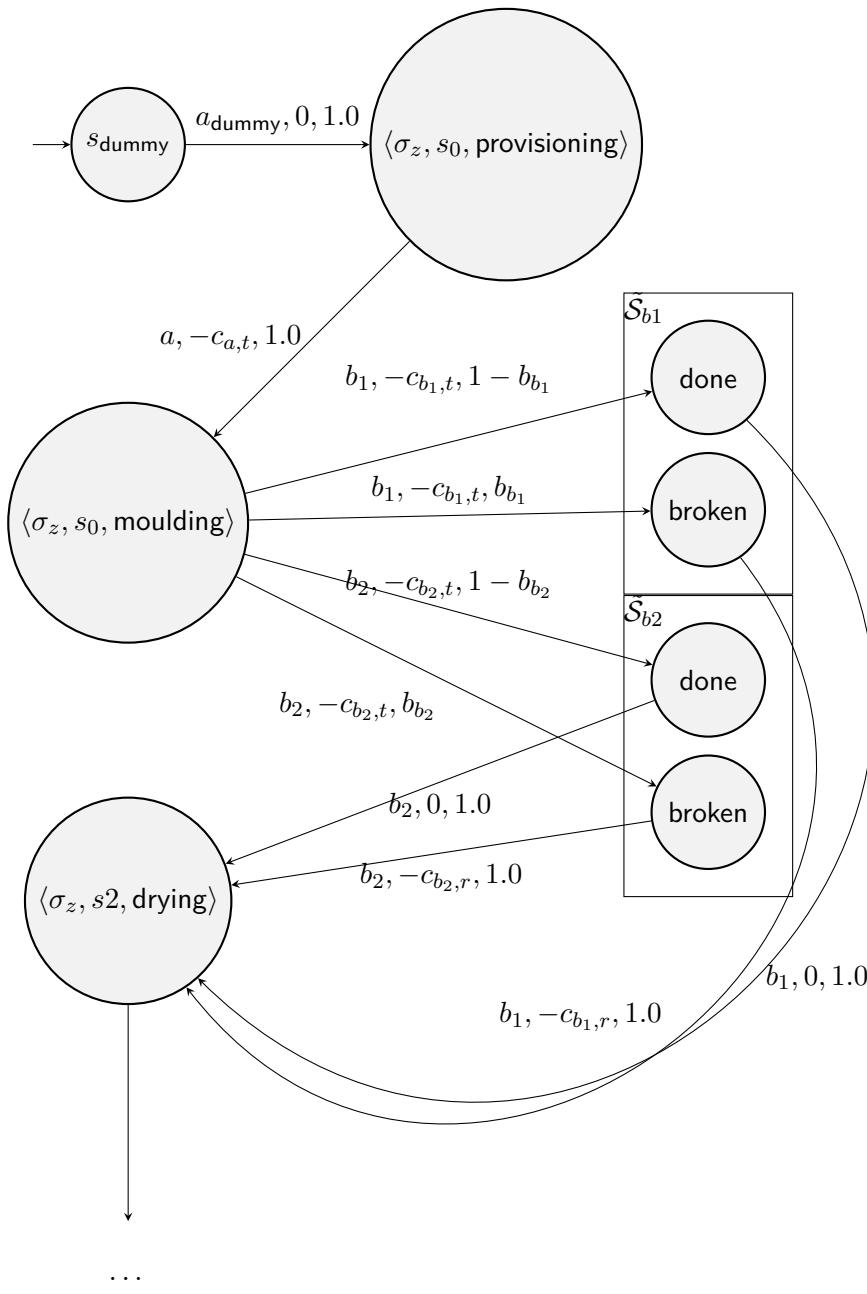


Figure 6.4. The initial part of an example of composition MDP for the use case. The system starts with dispatching the provisioning request to a simple service (forced choice); then, to process the request moulding, the orchestrator can choose between services $\tilde{\mathcal{S}}_{b1}$ and $\tilde{\mathcal{S}}_{b2}$, taking into account the costs c_{b_1} , c_{b_2} and the probability of breaking b_{b_1} , b_{b_2} . The execution continues after checking the operation on the service previously chosen (a forced orchestration choice, by construction).

Chapter 7

Solver

In this chapter we analyze the solver of stochastic service composition library and how every component is defined. The chapter is structured as follows:

- In **Section 7.1**: we give a small description about the library and the tools used, providing the github link of the project;
- In **Section 7.2**: we describe in detail the implementation of all services and target listing the code and automata of both;
- In **Section 7.3**: we define the construction of the system service providing the algorithm;
- In **Section 7.4**: we define the construction of the composition MDP providing the code, automata and the algorithm;
- In **Section 7.5**: we show the optimal policy calculation, listing the code and the output;

7.1 The `stochastic-service-composition` library

One of the core contribution of the thesis is the implementation of stochastic service composition approach described in Chapter 6. The library, called `stochastic-service-composition`, is implemented in *Python 3.8*, but can be used also for other Python's version and it is available on the following github link: <https://github.com/luusi/stochastic-service-composition>

We use *Graphviz*, an open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. In our case we imagine the services and the target as automata, we use it for that representation, as we show in the following.

7.2 Implementation of Services and Targets

In this section we describe stochastic service as defined in 6.1 and the target service as defined in 5.2

- **Every service is composed of:**

a set of states, a set of actions, an initial state, a set of final states and a transition function. The transition function is represented by nested dictionaries, where each pair state-action corresponds to a pair in which one is the dictionary from a state to probability (tells the probability to end up in a certain state) and the other is the reward.

The main function implemented in the service's construction (`build_service_from_transitions`) is the one that initialize a service from tran-

sitions, initial state and final states. The set of states and the set of actions are parsed from the transition function. This will guarantee that all the states are reachable. Consider the use case described in 4.4 we define the various services as we show in the following.

Provisioning and *shipping* services are formed by one state, with no possibility for machine to broken, with a probability of doing the action equal to 1.0 and reward (cost) equal to -1.0. Every other service is modelled with 3 states: *available*, *done* and *broken*. In the *available* state it is possible to execute the operation and goes to the *done* state (with a certain probability equal to “*good prob*”) or goes to the *broken* state (with prob 1 - “*good prob*”).

From both *done* state and *broken* state it is possible to execute the “check operation” and return to the *available state*: the difference is that the reward (or the cost) doing from *done* state is zero, while the reward (or the cost) doing from *broken* state is much higher (-10.0) because we have to fix the machine. So, the structure of remaining services is the same, for this reason we create a preliminary function that built device service in almost same way.

Also we can have multiple services that can do the same thing. For example, we can have various versions of a machine (new or old) or the same action can be done by a human. In our the service *painting* can be do by the machine or the human. Generally we can note that a machine is more efficient than a human, but it has a certain probability of wearing out and therefore ending up in the broken state. The human is less efficient than a machine but he has no probability of broken.

```
[1]: # Python imports, put at the top for simplicity
from docs.notebooks.utils import render_service
from stochastic_service_composition.services import
    build_service_from_transitions, Service

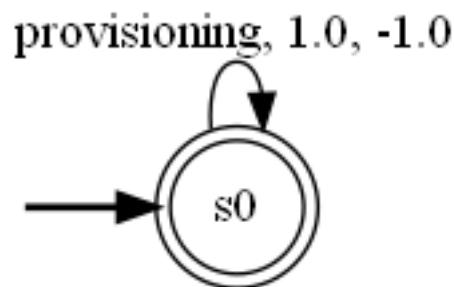
[2]: DEFAULT_REWARD = -1.0
DEFAULT_BROKEN_REWARD = -10.0
DEFAULT_BROKEN_PROB = 0.05

#This is a preliminary function
def _build_device_service(action_name: str, broken_prob: float,
    broken_reward: float, action_reward: float):
    assert 0.0 <= broken_prob <= 1.0
    success_prob = 1.0 - broken_prob
    transitions = {
        "available": {
            action_name: ({"done": success_prob, "broken": broken_prob},
                action_reward),
            },
        "broken": {
            f"check_{action_name}": ({"available": 1.0}, broken_reward),
            },
        "done": {
            f"check_{action_name}": ({"available": 1.0}, 0.0),
            }
    }
    final_states = {"available"}
    initial_state = "available"
```

```
    return build_service_from_transitions(transitions, initial_state, final_states) # type: ignore
```

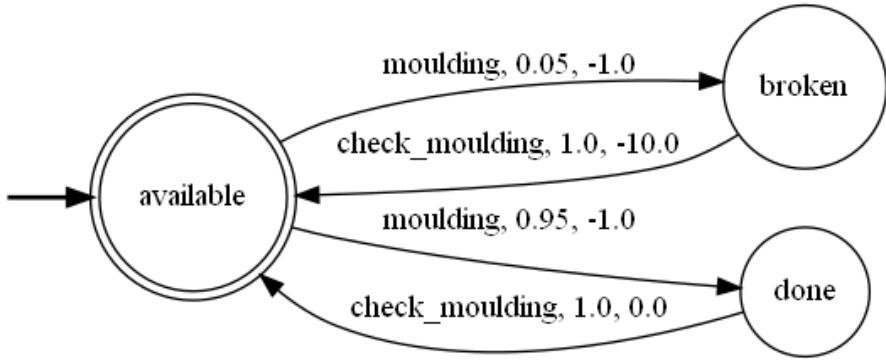
Provisioning service:

```
[3]: def provisioning_service(action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the provisioning device."""
    transitions = {
        "s0": {
            "provisioning": ({"s0": 1.0}, action_reward),
        },
    }
    final_states = {"s0"}
    initial_state = "s0"
    return build_service_from_transitions(transitions, initial_state, final_states) # type: ignore
service_provisioning=provisioning_service()
render_service(service_provisioning)
```



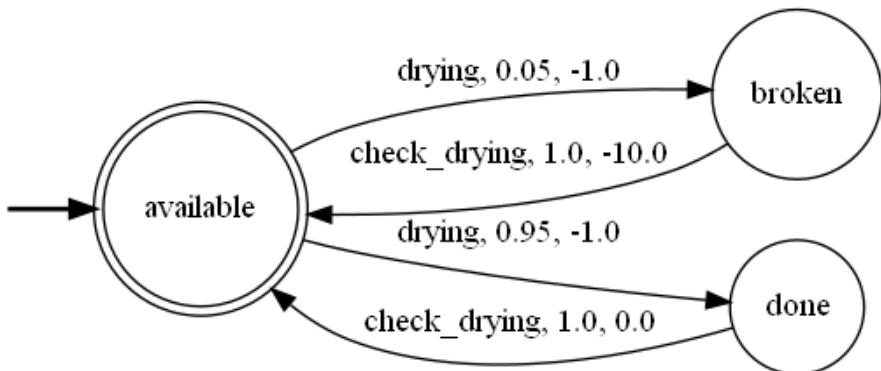
Moulding service:

```
[4]: def moulding_service(broken_prob: float = DEFAULT_BROKEN_PROB, broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the moulding device."""
    return _build_device_service("moulding", broken_prob=broken_prob, broken_reward=broken_reward, action_reward=action_reward)
service_moulding=moulding_service()
render_service(service_moulding)
```



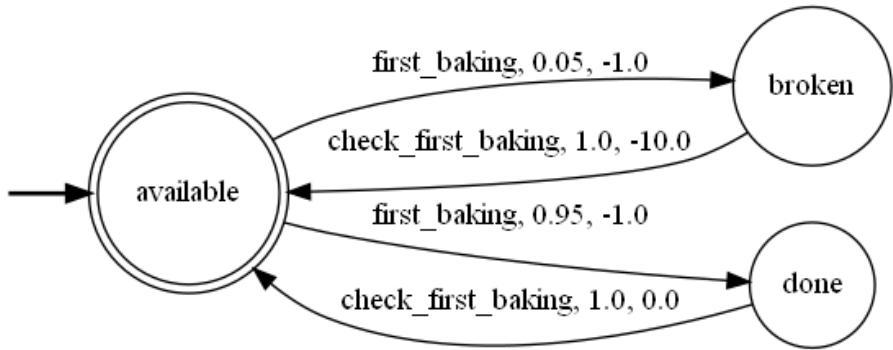
Drying service:

```
[5]: def drying_service(broken_prob: float = DEFAULT_BROKEN_PROB,
                     broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the drying device."""
    return _build_device_service("drying", broken_prob=broken_prob,
                                 broken_reward=broken_reward, action_reward=action_reward)
service_drying=drying_service()
render_service(service_drying)
```



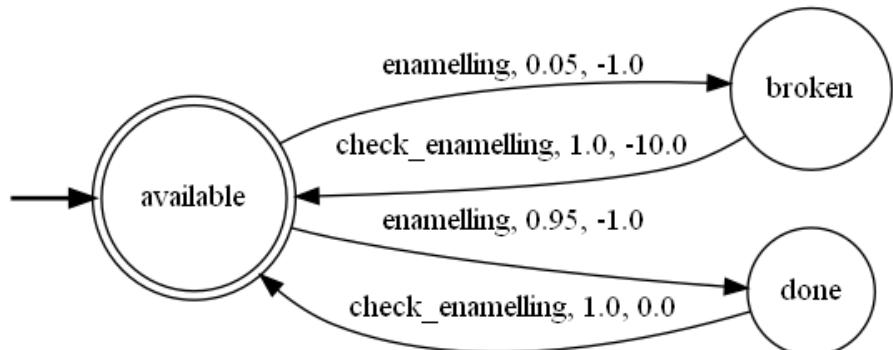
First baking service:

```
[6]: def first_baking_service(broken_prob: float = DEFAULT_BROKEN_PROB,
                            broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the first baking device."""
    return _build_device_service("first_baking",
                                 broken_prob=broken_prob, broken_reward=broken_reward,
                                 action_reward=action_reward)
service_first_baking=first_baking_service()
render_service(service_first_baking)
```



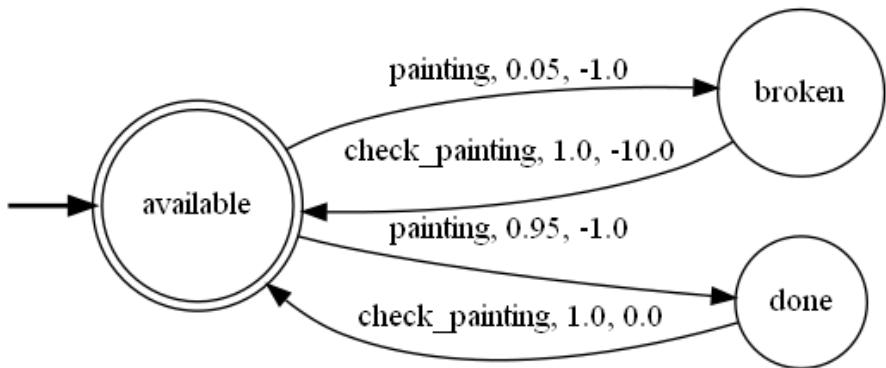
Enamelling service:

```
[7]: def enamelling(broken_prob: float = DEFAULT_BROKEN_PROB, broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the enamelling device."""
    return _build_device_service("enamelling", broken_prob=broken_prob, broken_reward=broken_reward, action_reward=action_reward)
service_enamelling=enamelling()
render_service(service_enamelling)
```



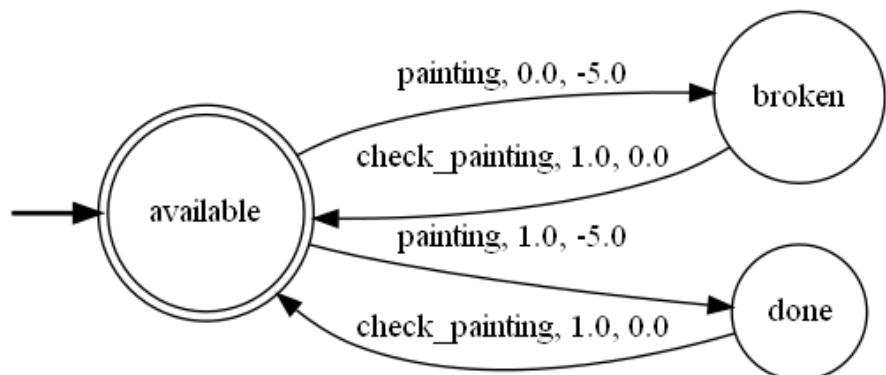
Painting service:

```
[8]: def painting_service(broken_prob: float = DEFAULT_BROKEN_PROB, broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the painting device."""
    return _build_device_service("painting", broken_prob=broken_prob, broken_reward=broken_reward, action_reward=action_reward)
service_painting=painting_service()
render_service(service_painting)
```



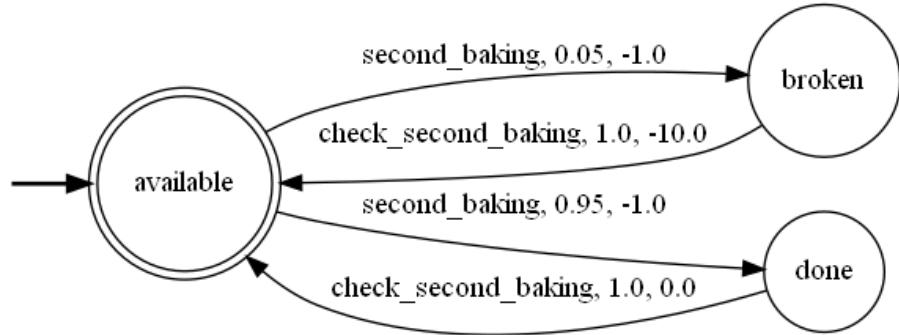
Painting by human service:

```
[9]: def painting_human_service(action_reward: float = -5.0) -> Service:
    """Build the painting device."""
    return _build_device_service("painting", broken_prob=0.0,
        ↵broken_reward=0.0, action_reward=action_reward)
service_painting_human=painting_human_service()
render_service(service_painting_human)
```



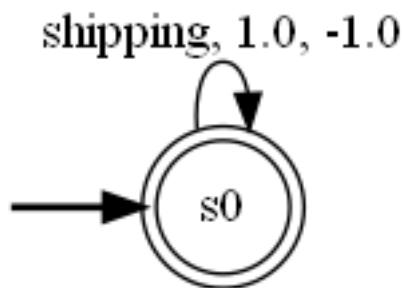
Second baking service:

```
[10]: def second_baking_service(broken_prob: float = DEFAULT_BROKEN_PROB,
    ↵broken_reward: float = DEFAULT_BROKEN_REWARD, action_reward: float = ↵
    ↵DEFAULT_REWARD) -> Service:
    """Build the second_baking device."""
    return _build_device_service("second_baking",
        ↵broken_prob=broken_prob, broken_reward=broken_reward,
        ↵action_reward=action_reward)
service_second_baking=second_baking_service()
render_service(service_second_baking)
```



Shipping service:

```
[11]: def shipping_service(action_reward: float = DEFAULT_REWARD) -> Service:
    """Build the shipping device."""
    transitions = {
        "s0": {
            "shipping": ({"s0": 1.0}, action_reward),
        },
    }
    final_states = {"s0"}
    initial_state = "s0"
    return build_service_from_transitions(transitions, initial_state, final_states) # type: ignore
service_shipping=shipping_service()
render_service(service_shipping)
```



- **The target is composed of:**

a set of states, a set of actions, an initial state, a set of final states, a transition function, a policy and a reward. The transition function is represented by nested dictionaries, where each pair state-action corresponds to next state. The policy is represented by nested dictionaries, where each pair state-action corresponds to the probability associated to the action from the state. Finally, the reward is represented by nested dictionaries, where each pair state-action corresponds to reward associated to the action from the state.

The main function implemented in the target's construction (`build_target_from_transitions`) is the one that initialize a service from transitions, initial state and final states. Also in this case, the set of states and the set of

actions are parsed from the transition function. This will guarantee that all the states are reachable.

Since the process of ceramics production in an industry is linear, the target service have for each state only one action with probability equal to 1.0 and unitary reward (or cost) equal to 0.0.

```
[9]: # Python imports, put at the top for simplicity
from docs.notebooks.utils import render_target
from stochastic_service_composition.target import
    build_target_from_transitions
```

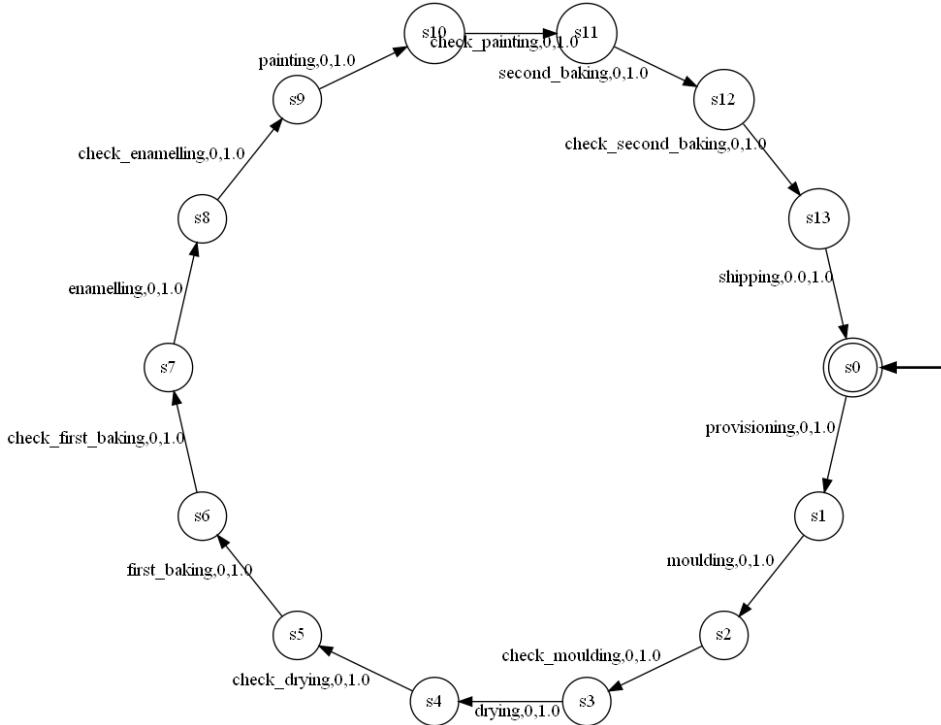
Target service:

```
[10]: def target_service():
    """Build the target service."""
    transition_function = {
        "s0": {
            "provisioning": ("s1", 1.0, 0),
        },
        "s1": {"moulding": ("s2", 1.0, 0),
        },
        "s2": {"check_moulding": ("s3", 1.0, 0),
        },
        "s3": {"drying": ("s4", 1.0, 0),
        },
        "s4": {"check_drying": ("s5", 1.0, 0),
        },
        "s5": {"first_baking": ("s6", 1.0, 0),
        },
        "s6": {"check_first_baking": ("s7", 1.0, 0),
        },
        "s7": {"enamelling": ("s8", 1.0, 0),
        },
        "s8": {"check_enamelling": ("s9", 1.0, 0),
        },
        "s9": {"painting": ("s10", 1.0, 0),
        },
        "s10": {"check_painting": ("s11", 1.0, 0),
        },
        "s11": {"second_baking": ("s12", 1.0, 0),
        },
        "s12": {"check_second_baking": ("s13", 1.0, 0),
        },
        "s13": {"shipping": ("s0", 1.0, 0.0),
        },
    }

    initial_state = "s0"
    final_states = {"s0"}

    return build_target_from_transitions(
        transition_function, initial_state, final_states
    )

target = target_service()
render_target(target, engine="circo")
```



To represent in terms of code the stochastic composition we proceed as follows: given a list of service instances we build the service's system and then given the target, the services and the discount factor, we compose the MDP.

7.3 Build a System Service

7.3.1 Description

The system service, according to the Definition 6.2, is the combination of the set of service's states, the set of target's states and the set of actions. We don't build the explicit cartesian product between the state spaces, as this would lead to large use of memory, due to the creation of many unreachable states. Instead, we build it in a forward fashion, incrementally, using a breadth-first approach which makes sure that all states are reachable. We avoid visiting the same node twice by using a hash-table that stores the visited nodes. When we visit a node for the first time, we proceed as follows: we check if every component of the system state (i.e. the node being visited in this iteration), is in the final state's set of the correspondent service; if so, we add the system state to the set of final states.

Then, we go ahead and we do an iteration for every service to find all transitions for which the i_{th} service can proceed. For every outgoing transaction from the service we add all the possible transitions to the system transitions. Since we have an asynchronous product, we enrich the system transition with other components of the system that are stopped, while it proceeds only the service component that we are analyzing in the "for" loop. If there is no transition out of the current state we set the set to empty. We add other components: possible successor states and the probability of reaching them, the other components must stand still. The algorithm terminates until all the reachable states have been visited.

7.3.2 Algorithm

Algorithm 7.1 contains the pseudocode of our algorithm. We build the System Service in an incremental fashion.

Algorithm 7.1. Build system service.

```

1: function BUILDSYSTEMSERVICE(services)
2:   NewStates  $\leftarrow \{\}$ 
3:   NewFinalStates  $\leftarrow \{\}$ 
4:   Actions  $\leftarrow \{\}$ 
5:   NewInitialState  $\leftarrow \text{tuple}(\text{ServiceInitialState})$ 
6:   NewTransitionFunction  $\leftarrow \{\}$ 
7:   queue.append(NewInitialState)
8:   ToBeVisited  $\leftarrow \{\text{NewInitialState}\}$ 
9:   Visited  $\leftarrow \{\}$ 
10:  while queue not empty do
11:    CurrentState  $\leftarrow \text{queue.pop}()$ 
12:    ToBeVisited.remove(CurrentState)
13:    Visited.add(CurrentState)
14:    NewStates.add(CurrentState)
15:    NextStateTemplate  $\leftarrow \text{CurrentState}$ 
16:    for i in ENUMERATE(Services) do
17:      CurrentService  $\leftarrow \text{Services}[i]$ 
18:      CurrentServiceState  $\leftarrow \text{list}(\text{NextStateTemplate})[i]$ 
19:      for a, (NextServStates, Rew) in CURSERVTRANFUN[CURSERVST].ITEMS() do
20:        Symbol  $\leftarrow (a, i)$ 
21:        Actions.add(symbol)
22:        NewTransFunct.setdefault(CurrentState,  $\{\}\text{[symbol]}$ )  $\leftarrow (\{\}, \text{rew})$ 
23:        for NextServiceState, prob in NEXTSERVICESTATES.ITEMS() do
24:          NextStateList  $\leftarrow \text{list}(\text{NextStateTemplate})$ 
25:          NextStateList[i]  $\leftarrow \text{NextServiceState}$ 
26:          NextState  $\leftarrow \text{tuple}(\text{NextStateList})$ 
27:          NewTransitionFunction[CurrentState][Symbol][0][NextState]  $\leftarrow \text{prob}$ 
28:          if NextState not in Visited and NextState not in ToBeVisited then
29:            ToBeVisited.add(NextState)
30:            queue.append(NextState)
31:  NewService  $\leftarrow (\text{NewStates}, \text{Actions}, \text{NewFinalStates}, \text{NewInitialState}, \text{NewTransFunction})$ 
32:  return NewService

```

7.4 Build the Composition MDP

The object “MDP” to which we refer is defined in a library of Standford that we use in the following link <https://github.com/coverdrive/MDP-DP-RL>, we did a fork of this project available at the following link: <https://github.com/luusi/MDP-DP-RL>.

7.4.1 Description

We start to define an MDP initial state, an MDP initial action and an MDP undefined action, used when the orchestrator cannot perform the target request. For example: if we end up in a state in which no service in the current states can consume the action of the target, then for consistency of the MDP there is an undefined action that makes the execution remain in the same state with probability 1.0.

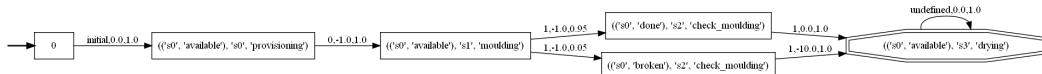
The construction of the composition MDP is performed in a forward fashion, for similar reasons of the system service construction. To begin with, we initialize the initial transitions, i.e. the ones that start from the dummy state s_0 and with the dummy action a_0 nondeterministically lead to a set of states, according to the first action choice of the target. These new states will be the next state to be visited. For all actions that the target can perform in the initial state we have the state formed by system state, target state and the request action (in this way we populate also queue). At each iteration, a state is visited and we add an orchestrator action for each action that the system service can do. Then, the construction proceeds straightforwardly according to Definition 6.3 and the data structures defined above.

The state without outgoing transitions are sink states, so if the number of the transition is 0 we add an auxiliary transition with action “undefined” which leads to the same state with probability 1. The algorithm terminates until all the reachable states have been visited.

```
[3]: # Python imports, put at the top for simplicity
from docs.notebooks.utils import render_composition_mdp, u
→service_provisioning, service_moulding, target
from stochastic_service_composition.composition import composition_mdp
```

```
[4]: all_services = [
    service_provisioning,
    service_moulding
]

mdp = composition_mdp(target, *all_services, gamma=0.9)
render_composition_mdp(mdp)
```



The figure above shows a small portion of the MDP composition with two services, due to lack of space.

7.4.2 Algorithm

Algorithm 7.2 contains the pseudocode of our algorithm. We build the System Service in an incremental fashion.

Algorithm 7.2. Build composition MDP.

```

1: function COMPOSITIONMDP(target, services, gamma)
2:   SystemService  $\leftarrow$  BuildSystemService(services)
3:   InitialState  $\leftarrow$  0
4:   Actions  $\leftarrow \{ \text{ENUMERATE}(\text{services}) \}$ 
5:   InitialAction  $\leftarrow$  "initial"
6:   Actions.add(InitialAction)
7:   ToBeVisited  $\leftarrow \{ \}$ 
8:   Visited  $\leftarrow \{ \}$ 
9:   TransitionFunction[InitialState]  $\leftarrow \{ \}$ 
10:  InitialTransitionDist  $\leftarrow \{ \}$ 
11:  SymbolsFromInitialState  $\leftarrow \text{target.policy}[\text{target.InitialState}].\text{keys}()$ 
12:  for Symbol in SYMBOLSFROMINITIALSTATE do
13:    NextState  $\leftarrow (\text{SystemService.InitialState}, \text{target.InitialState}, \text{Symbol})$ 
14:    NextProb  $\leftarrow \text{target.policy}[\text{target.InitialState}][\text{Symbol}]$ 
15:    InitialTransitionDist[NextState]  $\leftarrow \text{NextProb}$ 
16:    queue.append(NextState)
17:  TransitionFunction[InitialState][InitialAction]  $\leftarrow (\text{InitialTransitionDist}, 0.0)$ 
18:  while queue not empty do
19:    CurrentState  $\leftarrow \text{queue.pop}()$ 
20:    ToBeVisited.remove(.currentState)
21:    Visited.add(currentState)
22:    CurrentSystemState, CurrentTargetState, CurrentSymbol  $\leftarrow \text{CurrentState}$ 
23:    TransitionFunction[CurrentState]  $\leftarrow \{ \}$ 
24:    SystemSymbols  $\leftarrow \text{list}(\text{SystemService.TransitionFunction}[\text{CurrentSystemState}].\text{keys}())$ 
25:    SystemSymbolsBySymbols  $\leftarrow \{ \}$ 
26:    for Action, ServiceId in SYSTEMSYMBOLS do
27:      SystemSymbolsBySymbols.setdefault(Action, set()).add(ServiceId)
28:    for i in SYSTEMSYMBOLSBYSYMBOLS.GET(CURRENTSYMBOL, SET()) do
29:      NextTransitions  $\leftarrow \{ \}$ 
30:      NextReward  $\leftarrow \text{target.reward}[\text{CurrentTargetState}][\text{CurrentSymbol}]$ 
31:      if (CurrentSymbol, i) in SysService.TransFunction[CurrentSysState] then
32:        else0
33:        NextTargetState  $\leftarrow \text{target.TransitionFunction}[\text{CurrentTargetState}][\text{CurrentSymbol}]$ 
34:        NextSysState, NextSysRew  $\leftarrow \text{SysService.TransFun}[\text{CurrSysState}][\text{(CurrSymbol, i)}]$ 
35:        for NextSym, NextProb in TARGET.POLICY[NEXTTARGSTATE].ITEMS() do
36:          for NextSysState, NextSysProb in NEXTSYSSTATE.ITEMS() do
37:            NextState  $\leftarrow (\text{NextSystemState}, \text{NextTargetState}, \text{NextSymbol})$ 
38:            if (NextProb * NextSysProb) == 0 then
39:              continue
40:            NextTransitions[NextState]  $\leftarrow \text{NextProb * NextSystemProb}$ 
41:            if (NextState) not in Visited and NextState not in ToBeVisited then
42:              ToBeVisited.add(NextState)
43:              queue.append(NextState)
44:            TransitionFunction[CurrentState][i]  $\leftarrow (\text{NextTrans}, \text{NextRew} + \text{NextSysRew})$ 
45:            if len(TransFunc[CurrentState]) == 0.0 then
46:              TransitionFunction.setDefault(.currentState, {})
47:              TransFunction[CurrentState], ["undefined"]  $\leftarrow \{ (\text{CurrentState} : 1.0), 0.0 \}$ 
48:  return MDP(TransitionFunction, gamma)

```

7.5 Optimal Policy

In this section we show every step of optimal policy calculation, also for implement this part we refer to the library MDP-RL-DP as defined before in 7.4.

```
[11]: # Python imports, put at the top for simplicity
from mdp_dp_rl.algorithms.dp.dp_analytic import DPAnalytic
from docs.notebooks.utils import print_policy_data, print_value_function, \
    print_q_value_function, target, \
    service_provisioning, service_moulding, service_drying, \
    service_first_baking, service_enamelling, service_painting, \
    service_painting_human, service_second_baking, service_shipping
from stochastic_service_composition.composition import composition_mdp
```

```
[12]: all_services = [
    service_provisioning,
    service_moulding,
    service_drying,
    service_first_baking,
    service_enamelling,
    service_painting,
    service_painting_human,
    service_second_baking,
    service_shipping,
]

mdp = composition_mdp(target, *all_services, gamma=0.9)

opn = DPAnalytic(mdp, 1e-4)
opt_policy = opn.get_optimal_policy_vi()
value_function = opn.get_value_func_dict(opt_policy)
q_value_function = opn.get_act_value_func_dict(opt_policy)

# remove '0' state to sort output
opt_policy.policy_data.pop(0, None)
value_function.pop(0, None)
_ = q_value_function.pop(0, None)
```

```
[13]: print_policy_data(opt_policy)
```

```
Policy:
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's0', 'provisioning'), Action=0
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's1', 'moulding'), Action=1
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's11', 'second_baking'), Action=7
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's13', 'shipping'), Action=8
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's3', 'drying'), Action=2
State=([('s0', 'available', 'available', 'available', 'available', 'available', 'available',
         'available', 'available', 's0'), 's5', 'first_baking'), Action=3
```

```

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's7', 'enamelling'), Action=4
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's9', 'painting'), Action=5
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'broken', 's0'), 's12', 'check_second_baking'), Action=7
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'done', 's0'), 's12', 'check_second_baking'), Action=7
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'done', 'available', 's0'), 's10', 'check_painting'), Action=6
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's10', 'check_painting'), Action=5
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's10', 'check_painting'), Action=5
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'), Action=4
State=([('s0', 'available', 'available', 'available', 'done', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'), Action=4
State=([('s0', 'available', 'available', 'broken', 'available', 'available',
'available', 'available', 's0'), 's6', 'check_first_baking'), Action=3
State=([('s0', 'available', 'available', 'done', 'available', 'available',
'available', 'available', 's0'), 's6', 'check_first_baking'), Action=3
State=([('s0', 'available', 'broken', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'), Action=2
State=([('s0', 'available', 'done', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'), Action=2
State=([('s0', 'broken', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'), Action=1
State=([('s0', 'done', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'), Action=1

```

[14]: `print_value_function(value_function)`

```

Value function:
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 'provisioning'), value=-8.016734337200308
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's1', 'moulding'), value=-7.796371485778121
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's11', 'second_baking'), value=-8.104199331819025
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's13', 'shipping'), value=-8.215060903480278
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's3', 'drying'), value=-7.835026525652
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's5', 'first_baking'), value=-7.882748797101234
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's7', 'enamelling'), value=-7.941665181606463

```

```

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 'available', 's0'), 's9', 'painting'),
value=-8.014401458773412
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'broken', 's0'), 's12', 'check_second_baking'),
value=-17.39355481313225
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'done', 's0'), 's12', 'check_second_baking'),
value=-7.39355481313225
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'done', 'available', 's0'), 's10', 'check_painting'),
value=-7.293779398637123
State=([('s0', 'available', 'available', 'available', 'available', 'broken',
'available', 'available', 's0'), 's10', 'check_painting'),
value=-17.29377939863712
State=([('s0', 'available', 'available', 'available', 'available', 'done',
'available', 'available', 's0'), 's10', 'check_painting'),
value=-7.293779398637123
State=([('s0', 'available', 'available', 'available', 'broken', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'),
value=-17.21296131289607
State=([('s0', 'available', 'available', 'available', 'done', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'),
value=-7.21296131289607
State=([('s0', 'available', 'available', 'broken', 'available', 'available',
'available', 'available', 's0'), 's6', 'check_first_baking'),
value=-17.147498663445816
State=([('s0', 'available', 'available', 'done', 'available', 'available',
'available', 'available', 's0'), 's6', 'check_first_baking'),
value=-7.147498663445816
State=([('s0', 'available', 'broken', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'),
value=-17.09447391739111
State=([('s0', 'available', 'done', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'),
value=-7.09447391739111
State=([('s0', 'broken', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'),
value=-17.0515238730868
State=([('s0', 'done', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'),
value=-7.051523873086801

```

[15]: `print_q_value_function(q_value_function)`

```

Q-value function:
State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 'available', 's0'), 's0', 'provisioning'):
    Action=0,           Value=-8.016734337200308

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 'available', 's0'), 's1', 'moulding'):
    Action=1,           Value=-7.796371485778121

```

```

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's11', 'second_baking'):
        Action=7,           Value=-8.104199331819025

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's13', 'shipping'):
        Action=8,           Value=-8.215060903480278

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's3', 'drying'):
        Action=2,           Value=-7.835026525651999

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's5', 'first_baking'):
        Action=3,           Value=-7.882748797101234

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's7', 'enamelling'):
        Action=4,           Value=-7.941665181606463

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's9', 'painting'):
        Action=5,           Value=-8.01440145877341
        Action=6,           Value=-11.56440145877341

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's12', 'check_second_baking'):
        Action=7,           Value=-17.39355481313225

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'done', 's0'), 's12', 'check_second_baking'):
        Action=7,           Value=-7.39355481313225

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'done', 'available', 's0'), 's10', 'check_painting'):
        Action=6,           Value=-7.293779398637123

State=([('s0', 'available', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's10', 'check_painting'):
        Action=5,           Value=-17.293779398637124

State=([('s0', 'available', 'available', 'available', 'available', 'done',
'available', 'available', 's0'), 's10', 'check_painting'):
        Action=5,           Value=-7.293779398637123

State=([('s0', 'available', 'available', 'available', 'broken', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'):
        Action=4,           Value=-17.21296131289607

State=([('s0', 'available', 'available', 'available', 'done', 'available',
'available', 'available', 's0'), 's8', 'check_enamelling'):
        Action=4,           Value=-7.212961312896071

State=([('s0', 'available', 'available', 'broken', 'available', 'available',
'available', 'available', 's0'), 's11', 'second_baking'):
        Action=7,           Value=-8.104199331819025

```

```
'available', 'available', 's0'), 's6', 'check_first_baking'):
    Action=3,           Value=-17.147498663445816

State=((('s0', 'available', 'available', 'done', 'available', 'available',
'available', 'available', 's0'), 's6', 'check_first_baking'):
    Action=3,           Value=-7.147498663445816

State=((('s0', 'available', 'broken', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'):
    Action=2,           Value=-17.094473917391113

State=((('s0', 'available', 'done', 'available', 'available', 'available',
'available', 'available', 's0'), 's4', 'check_drying'):
    Action=2,           Value=-7.094473917391111

State=((('s0', 'broken', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'):
    Action=1,           Value=-17.0515238730868

State=((('s0', 'done', 'available', 'available', 'available', 'available',
'available', 'available', 's0'), 's2', 'check_moulding'):
    Action=1,           Value=-7.051523873086801
```

In the calculation of the q-value function it is interesting to note, when the painting service is called (which can be performed by a robot or a human) that the value when the robot performs the action is less than the value when the human performs the action. This perfectly reflects the fact that the robot is more efficient than the human. Obviously if the machine begins to have significant wear and is more likely to break (and also it has an high cost to be repaired), these values will change and it will be more convenient for the human to perform the action, who in this case, even being slower, has no risk of breaking and no cost to be repaired.

Chapter 8

Technological Solution based on Digital Twins

In this chapter, we describe an application of the model introduced in Chapter 6 in an Industry 4.0 scenario. In particular, the application will involve the orchestration of several Digital Twins to accomplish the creation of a product through the assembly line. The chapter is structured as follows:

- In **Section 8.1**: we describe the architecture of the project, listing the most important components between Bosch IoT Things and the system and how are managed the connection between them;
- In **Section 8.2**: we explain how we define services in Bosch IoT Things platform;
- In **Section 8.3**: we explain how we define the target in Bosch IoT Things platform;
- In **Section 8.4**: we talk about the workflow of the project and how target, services and orchestrator communicates between them.

8.1 High-level Architecture

8.1.1 Overview

The main architecture components are:

- **Bosch IoT Things**, which provides the API for interacting with the Digital Twins and the actual devices;
- **Bosch IoT Hub**, which allows the devices to securely communicate in a wide range of protocols:
 - It receives the messages from the devices and it forwards them to Bosch IoT Things;
 - It receives the messages from Bosch IoT Things and it forwards them to the appropriate devices.
- The devices (both the services and the target) are connected with Bosch IoT Hub via **MQTT** only. The connection between Bosch IoT Hub and Bosch IoT Thing uses **AMQP 1.0** (Advanced Message Queuing Protocol) that is a frame and transfer protocol that allows messages to be transferred between two parties asynchronously, securely and reliably. It is the main protocol of Hub event messaging.
- The **Orchestrator** process, that:

- Collects the Digital Twin descriptions and their current values;
- Invokes the composition of MDP and the optimal policy calculation;
- Depending on the target's actions, dispatches the request to the service that has to perform the action according to the optimal policy; it then updates the current state and goes on.

The Orchestrator communicates with Bosch IoT Things through WebSocket communication.

Figure 8.1 depicts a diagram of the architecture.

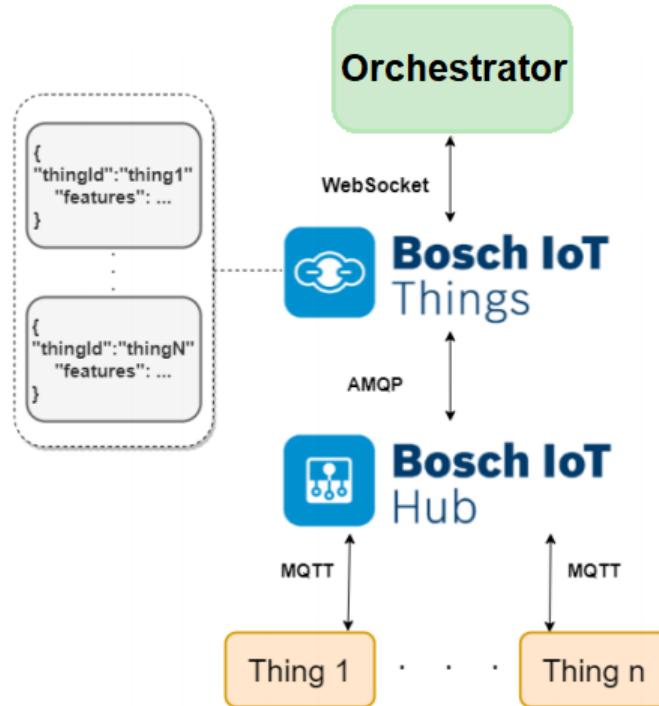


Figure 8.1. Software Architecture

8.1.2 MQTT

The connection uses between services, target and Bosch IoT Hub is **MQTT** only, because provides:

- publish/subscribe at the protocol level,
- quality of service: it is for small band communication channels and a restricted number of devices,
- a limited overhead.

For managing this protocol we create an abstract class where first, we define all parameters for representing a Thing:

- `device_id`: the ID of tenant,
- `tenant_id`: the ID of tenant,

- `hub_adapter_host`: MQTT IoT Hub,
- `certificate_path`: path to IoT Hub certificate,
- `device_password`: the secret that registered with the credential,
- `client_id`: MQTT client,
- `authentication_id`: the ID of the credential,
- `ditto_topics`

Then, we establish a connection with MQTT client that is subscribed to the Bosch IoT Hub and it is ready to receive commands. After this services receive the message and the command from the Orchestrator, execute it and update current state. The target, instead receive message from Orchestrator, sample the next action and send it to Orchestrator.

8.1.3 WebSocket

WebSocket is designed for point to point connections that mainly works in an environment that support publish/subscribe architectures. In this case permits the communication between the Orchestrator and Bosch IoT Hub.

The Orchestrator, after collecting Digital Twins description connects to the Bosch IoT Thing Websocket endpoint, composing MDP, calculate optimal policy and issues a command to request events related to the Things:

START-SEND-EVENTS

The term "event" refers to any state change of the Digital Twin. After this the Orchestrator listen to the event originating, so it waits for target action, once it receives send the action to the right service and again he waits from the service the update state after performing the action, as we will explain in detail later in 8.4.3.

8.1.4 ThingAPI

The Things defined in Bosch IoT Things interact through HTTP API 2. We determine following API call:

- `get_thing`: returns all things passed in by the required parameter ids, which you (the authorized subject) are allowed to read.

We use this API call when we have to launch devices and we have to distinguish between services and target;

- `search_things`: used to search for things. The query parameter filter if it is not set, the result contains all things which the logged in user is allowed to read. If we set:

– `search_services`: search things with type=service (setting "filter=eq(attributes/type,'service')").

We use this API call when we download the JSON description of the services from the platform;

– `search_targets`: search things with type=target (setting "filter=eq(attributes/type,'target')").

We use this API call when we download the JSON description of the target from the platform.

- **send_message_to_thing**: send a message to a specific thing.

We use this API call when the Orchestrator has to send message to the chosen service Thing about the action to perform and also when it has to send the ack that the action is done to the target Thing;

- **receive_message_from_thing**: send a message from a specific thing
- **change_property**: create or update a specific property of a feature identified by the thingId and featureId path parameter.

8.2 Services as Digital Twins

Every service is implemented as Thing with JSON description language. Each Digital Twin is composed by:

- **thingId**: the ID of the Thing;
- **policyId**: the ID of the policy (3.4.2). In our case this parameter is not important, but it is created by default with the Thing.
- **attributes**: the type that represents whether the Thing is a service, the transitions, the initial state and the set of final states of the services, according to the definition proposed in 7.2;
- **features**: the current state that change every time the Thing performs an action and goes to the new state

Figure 8.2 shows the JSON description of the *provisioning service* of our use case, that has a single state and no possibility to broken. So, since we have only one action we assign to the transition the probability of 1.0 and since the reward represents the cost of doing an action, we assign a negative reward of -1.0.

```

1  {
2      "thingId": "com.bosch.service:provisioning_service",
3      "policyId": "com.bosch.service:provisioning_service",
4      "attributes": {
5          "type": "service",
6          "transitions": {
7              "available": {
8                  "provisioning": [
9                      {
10                         "available": 1.0
11                     },
12                     -1.0
13                 ]
14             }
15         },
16         "initial_state": "available",
17         "final_states": [
18             "available"
19         ]
20     },
21     "features": {
22         "current_state": {
23             "properties": {
24                 "value": "available"
25             }
26         }
27     }
28 }
29

```

Figure 8.2. Provisioning service Thing

Let's see how the definition of a service that can be broken changes. Figure 8.3 shows the JSON description of the *moulding service* of our use case. When we perform the action here, we have a chance that the machine will do everything right or it may break down. The probability of success is defined as ($1 - \text{the probability of a machine breaking}$). If the machine ends up in a broken state, it can be repaired and returned to the available state, but this is expensive in terms of cost (reward), which is -10. On the other hand, if the action is successful and goes in the done state, the machine will check the correctness of the action and return to the available state with a cost (reward) of 0.

```
1 {  
2     "thingId": "com.bosch.service:moulding_service",  
3     "policyId": "com.bosch.service:moulding_service",  
4     "attributes": {  
5         "type": "service",  
6         "transitions": {  
7             "available": {  
8                 "moulding": [  
9                     {  
10                    "done": 0.95,  
11                    "broken": 0.05  
12                },  
13                -1  
14            ]  
15        },  
16        "broken": {  
17            "check_moulding": [  
18                {  
19                    "available": 1  
20                },  
21                -10  
22            ]  
23        },  
24        "done": {  
25            "check_moulding": [  
26                {  
27                    "available": 1  
28                },  
29                0  
30            ]  
31        }  
32    },  
33    "initial_state": "available",  
34    "final_states": [  
35        "available"  
36    ]  
37},  
38    "features": {  
39        "current_state": {  
40            "properties": {  
41                "value": "available"  
42            }  
43        }  
44    }  
45}  
46}
```

Figure 8.3. Moulding service Thing

8.3 Target as Digital Twin

Also the target is implemented as Thing with JSON description language. It is composed by:

- **thingId**: the target identifier;
- **attributes**: the type that represents whether the Thing is a target, the transitions, the initial state and the set of final states of the target, according to the definition proposed in 7.2;
- **features**: the current action, that change every time the target perform an action, depending on the choice of the user.

Figure 8.4 shows a JSON description portion of the target of our use case. The depicted process is linear, thus we can imagine the target service having for each state a single action (with action probability 1.0) and a unitary (or zero) reward. It might seem that there are not many differences compared to the service definitions, but what changes are the mapping of the transition function and that in features, we define the current action that can undergo variation according to the user's choice.

```

1  {
2      "thingId": "com.bosch.service:target",
3      "policyId": "com.bosch.service:target",
4      "attributes": {
5          "type": "target",
6          "transitions": {
7              "s0": {
8                  "provisioning": [
9                      "s1",
10                     1,
11                     0
12                 ],
13             },
14             "s1": {
15                 "moulding": [
16                     "s2",
17                     1,
18                     0
19                 ],
20             },
21             "s2": {
22                 "drying": [
23                     "s3",
24                     1,
25                     0
26                 ],
27             },
28             "s3": {
29                 "check_drying": [
30                     "s4",
31                     1,
32                     0
33                 ],
34             },
35             "s4": {
36                 "first_baking": [
37                     "s5",
38                     1,
39                     0
40                 ],
41             }
42         },
43         "initial_state": "s0",
44         "final_states": [
45             "s0"
46         ]
47     },
48     "features": {
49         "current_action": {
50             "properties": {
51                 "value": "provisioning"
52             }
53         }
54     }
55 }
56

```

Figure 8.4. Target service Thing

8.4 Workflow and Communication Details

8.4.1 Target and Orchestrator

The first communication that is established in our system is between the Target and the Orchestrator. The Target is the core of the execution, it is considered as real service and it decides the action to do, in particular decides the actions that the user wants to do and sends them to the Orchestrator. The protocol starts with the request from the target of the action to the orchestrator; after this he waits from it the ack that the action is done. The orchestrator waits

until the target send him the action and decides (according to the optimal policy) to which service the action can be done. Once the action is done, what the orchestrator do is update the current state of the MDP and send a message to the target that the action is done. Once the target receives this ack can perform the next action and goes on. All this steps are pictured in Figure 8.5:

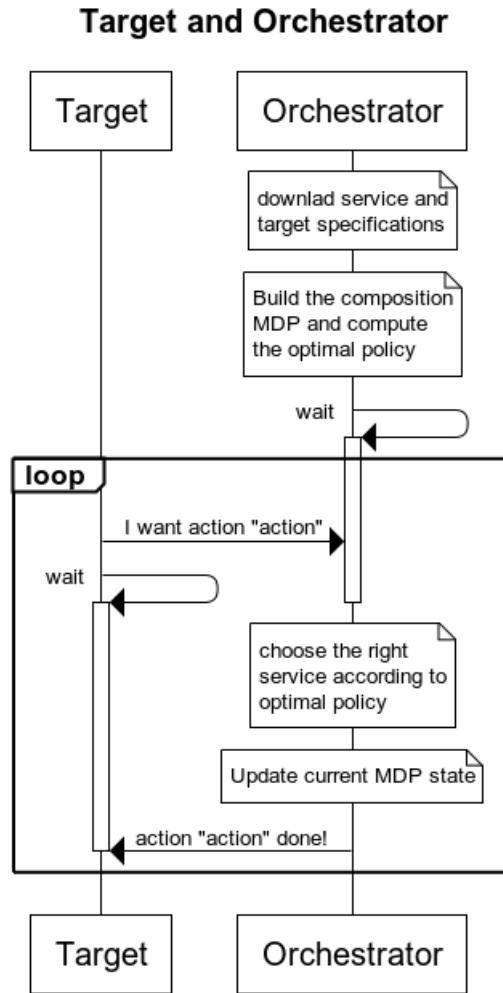


Figure 8.5. Communication between Target and Orchestrator

8.4.2 Orchestrator and Services

As much important is the role of the Orchestrator, which has the task of forwarding the action that the target wants to do to the correct service that can perform it. Until it receives the action from the target, it waits for it, then once it receives, according to the optimal policy calculated choose the right service and send the action to it. The service does the action and notifies the state in which it is. At this point, the Orchestrator update the current state of MDP and send an ack to the Target that the action is done. In Figure 8.6 we show a sketch of this communication

Orchestrator and Services

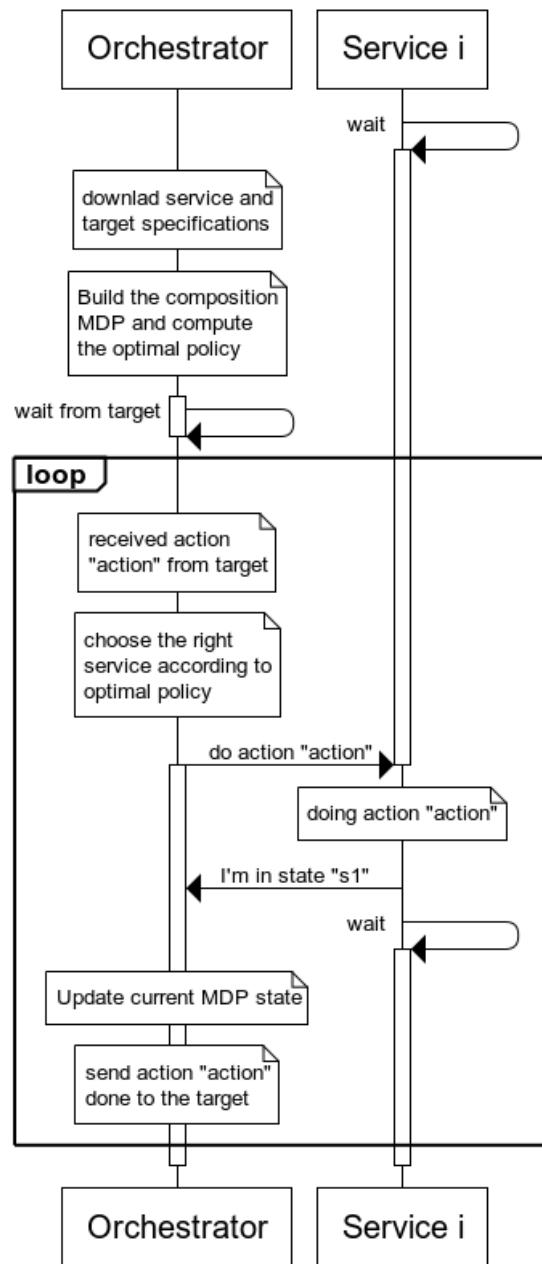


Figure 8.6. Communication between Orchestrator and Services

8.4.3 Target, Orchestrator and Services

The communication between all components of the system is depicted in Figure 8.7 and is divided in several steps.

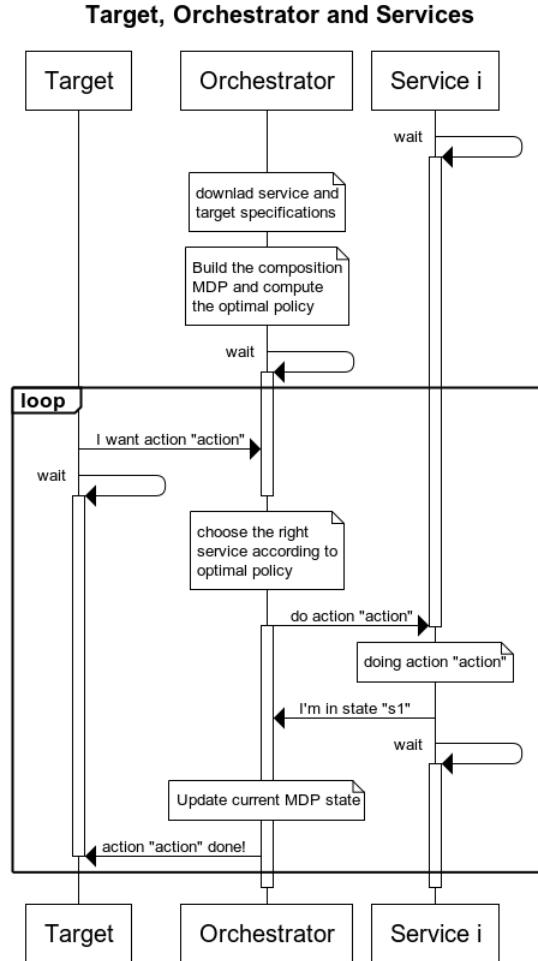


Figure 8.7. Communication between Target, Orchestrator and Services. Note that all the communications happen through the Bosch IoT Hub platform.

Every Digital Twin that represents the services and the target lives in its process. The available services are dedicated only to this orchestration, they don't talk between them and the communication starts from target to the service, through the Orchestrator, but not vice versa. The transitions of services and target are download by their Thing JSON description. Once having this, the Orchestrator connects to the Bosch IoT Thing Websocket endpoint, invoke the composition of MDP and the calculation of optimal policy as explained in 7.4, invoke the target simulator that defines the current state of the target and updates the state given an action, and starts from the initial state of system state as explained in After that the Orchestrator issues a command to request events related to things and listen to events originating. Now start the iterations and the communication between Target and Orchestrator as described in detail in 8.4.1, in particular the target action is taken from the JSON description file, the current state and the updating of the state from target simulator. Now the current state of the MDP is formed by system state, current target state and target action, at this point the Orchestrator calculates the optimal policy and begin the communication between Orchestrator and Services as described in detail in 8.4.2 and he goes on for every iteration.

8.4.4 Special case

Let's see what happens when the action request by the target cannot be dispatched, following Figure 8.8. As seen, the target requires an action that he needs, but no service at that moment may be in the conditions, for example in the right state, to be able to process that request. We assume that the orchestrator is always updated on the final status of the services. If the orchestrator cannot delegate the action to the service, it replies to the target that the action required cannot be done. On the other hand if the request of the target can be dispatched all the system behave as 8.4.3.

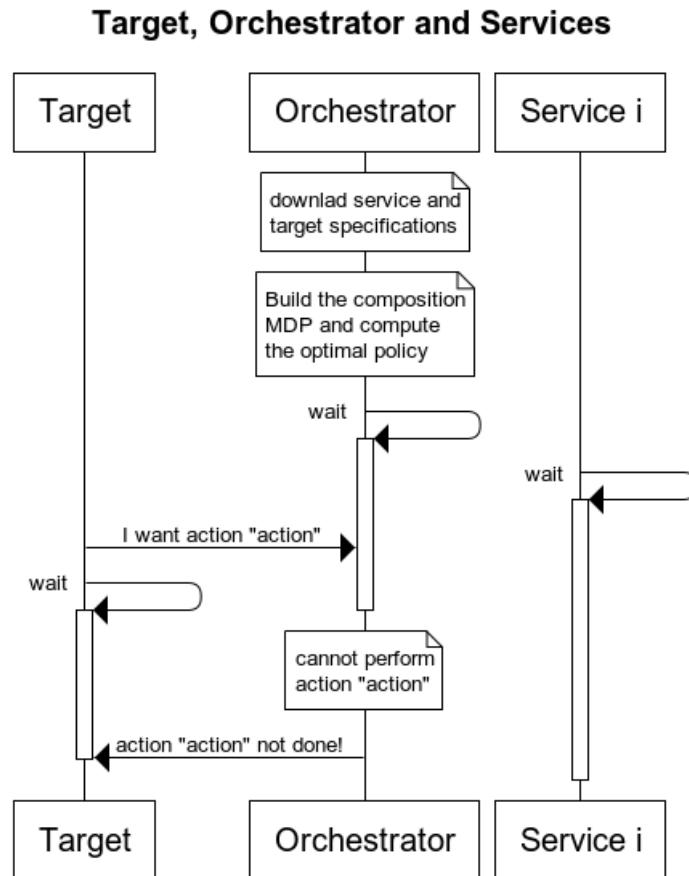


Figure 8.8. Communication between Target, Orchestrator and Services when target action cannot be dispatched

Chapter 9

Proof-of-Concept Implementation

In this chapter we show the proof-of-concept of our use case. The chapter is structured as follows:

- In **Section 9.1**: we illustrate DT implementation of the services and target on Bosch IoT Things, providing the relative automata;
- In **Section 9.2**: we describe in detail the main function of the project, showing the code;
- In **Section 9.3**: we describe the code of launching devices;
- In **Section 9.4**: we show the execution of the framework, describing how component communicates between them.

9.1 Bosch IoT Things DT Implementation

The communication between the orchestrator and the devices takes place through Bosch IoT Things and Bosch IoT Hub, as we showed before in Section 8.1 and in Figure 8.1. The devices are defined in Bosch IoT Things as Digital Twins. Once we connect on the platform we can request the action to a service; every time that the service performs an action the current state is updated to the next state (the value of this variable change whenever is performed an operation). In the following we show the set of services remanding the structure in 8.2:

Provisioning DT:

```
{
    "thingId": "com.bosch.service:provisioning_service",
    "policyId": "com.bosch.service:provisioning_service",
    "attributes": {
        "type": "service",
        "transitions": {
            "available": {
                "provisioning": [
                    {
                        "available": 1
                    },
                    -1
                ]
            }
        },
        "initial_state": "available",
    }
}
```

```

    "final_states": [
        "available"
    ],
    "features": {
        "current_state": {
            "properties": {
                "value": "available"
            }
        }
    }
}

```

The provisioning DT corresponds to the automata shows in Figure 9.1:

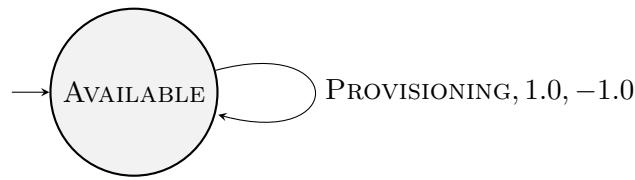


Figure 9.1. Provisioning automata.

Moulding DT:

```

{
    "thingId": "com.bosch.service:moulding_service",
    "policyId": "com.bosch.service:moulding_service",
    "attributes": {
        "type": "service",
        "transitions": {
            "available": {
                "moulding": [
                    {
                        "done": 0.95,
                        "broken": 0.05
                    },
                    -1
                ],
                "broken": {
                    "check_moulding": [
                        {
                            "available": 1
                        },
                        -10
                    ]
                },
                "done": {
                    "check_moulding": [
                        {
                            "available": 1
                        },
                        0
                    ]
                }
            }
        }
    }
}

```

```

        ]
    }
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "done"
        }
    }
}
}—

```

The moulding DT corresponds to the automata shows in Figure 9.2:

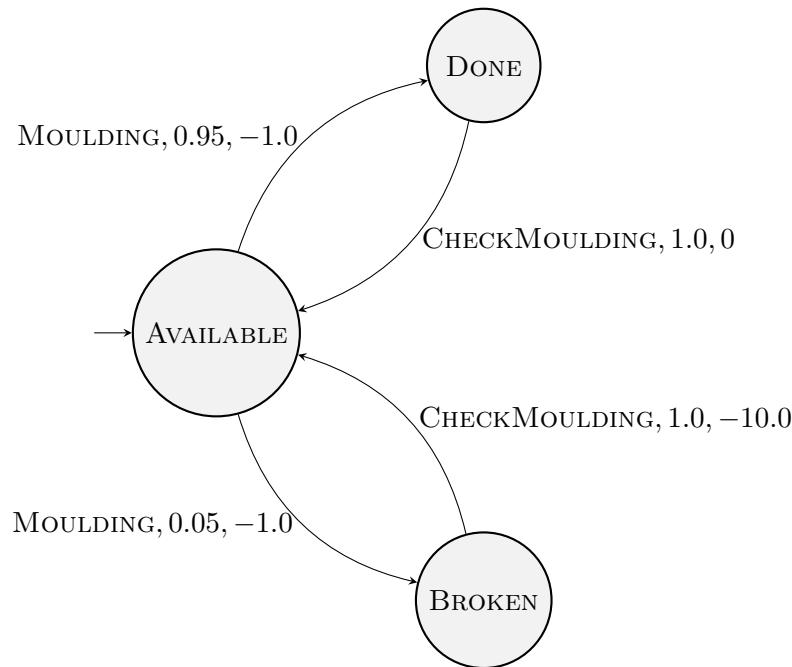


Figure 9.2. Moulding automata.

Drying DT:

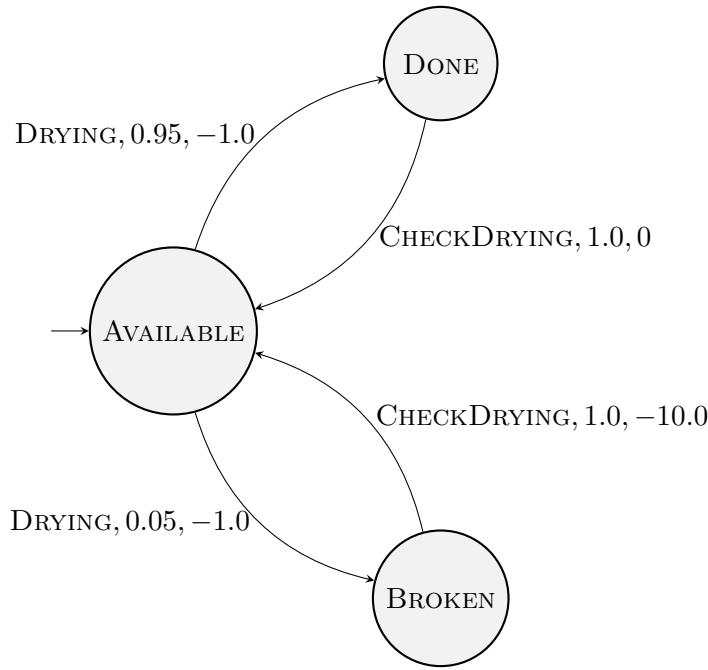
```
{
    "thingId": "com.bosch.service:drying_service",
    "policyId": "com.bosch.service:drying_service",
    "attributes": {
        "type": "service",
        "transitions": {
            "available": {
                "drying": [
                    {
                        "done": 0.95,

```

```

        "broken": 0.05
    },
    -1
]
},
"broken": {
    "check_drying": [
        {
            "available": 1
        },
        -10
    ]
},
"done": {
    "check_drying": [
        {
            "available": 1
        },
        0
    ]
}
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
}
}
```

The drying DT corresponds to the automata shows in Figure 9.3:

**Figure 9.3.** Drying automata.

First-baking DT:

```
{
  "thingId": "com.bosch.service:first_baking_service",
  "policyId": "com.bosch.service:first_baking_service",
  "attributes": {
    "type": "service",
    "transitions": {
      "available": {
        "first_baking": [
          {
            "done": 0.95,
            "broken": 0.05
          },
          -1
        ]
      },
      "broken": {
        "check_first_baking": [
          {
            "available": 1
          },
          -10
        ]
      },
      "done": {
        "check_first_baking": [
          {
            "available": 1
          },
          0
        ]
      }
    }
  }
}
```

```
        ]
    }
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
```

The first-baking DT corresponds to the automata shows in Figure 9.4:

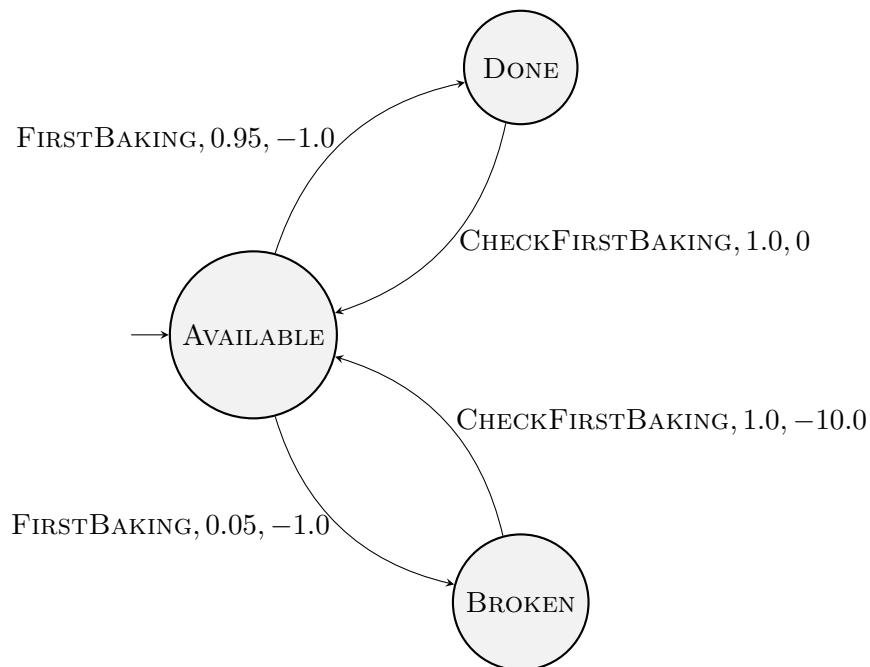


Figure 9.4. First-baking automata.

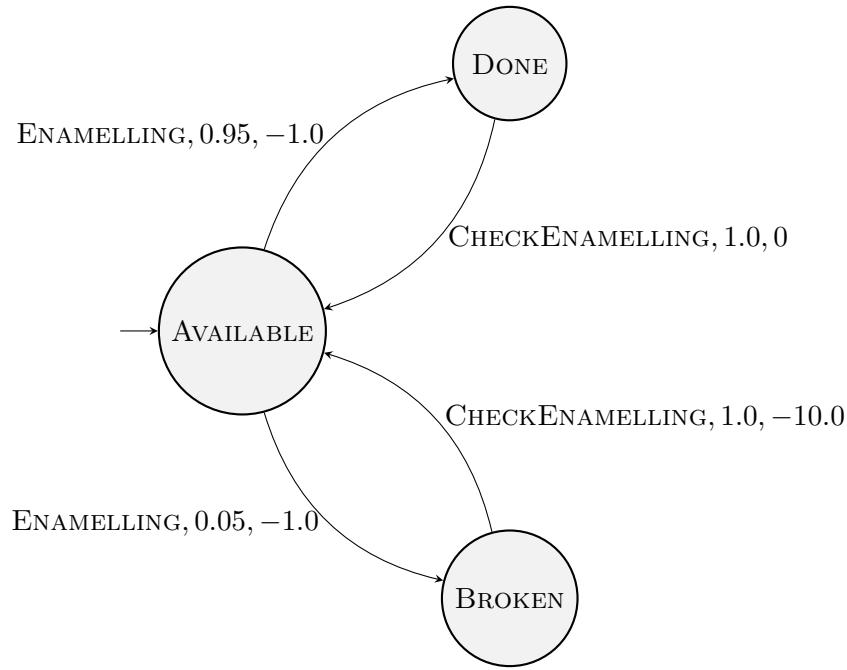
Enamelling DT:

```
{  
    "thingId": "com.bosch.service:enamelling_service",  
    "policyId": "com.bosch.service:enamelling_service",  
    "attributes": {  
        "type": "service",  
        "transitions": {  
            "available": {  
                "enamelling": [  
                    {  
                        "done": 0.95,  
                        "label": "Available"  
                    }  
                ]  
            }  
        }  
    }  
}
```

```

        "broken": 0.05
    },
    -1
]
},
"broken": {
    "check_enamelling": [
        {
            "available": 1
        },
        -10
    ]
},
"done": {
    "check_enamelling": [
        {
            "available": 1
        },
        0
    ]
}
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
}
}
```

The enamelling DT corresponds to the automata shows in Figure 9.5:

**Figure 9.5.** Enamelling automata.

Painting DT:

```
{
    "thingId": "com.bosch.service:painting_service",
    "policyId": "com.bosch.service:painting_service",
    "attributes": {
        "type": "service",
        "transitions": {
            "available": {
                "painting": [
                    {
                        "done": 0.95,
                        "broken": 0.05
                    },
                    -1
                ]
            },
            "broken": {
                "check_painting": [
                    {
                        "available": 1
                    },
                    -10
                ]
            },
            "done": {
                "check_painting": [
                    {
                        "available": 1
                    },
                    0
                ]
            }
        }
    }
}
```

```
        ]
    }
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
```

The painting DT corresponds to the automata shows in Figure 9.6:

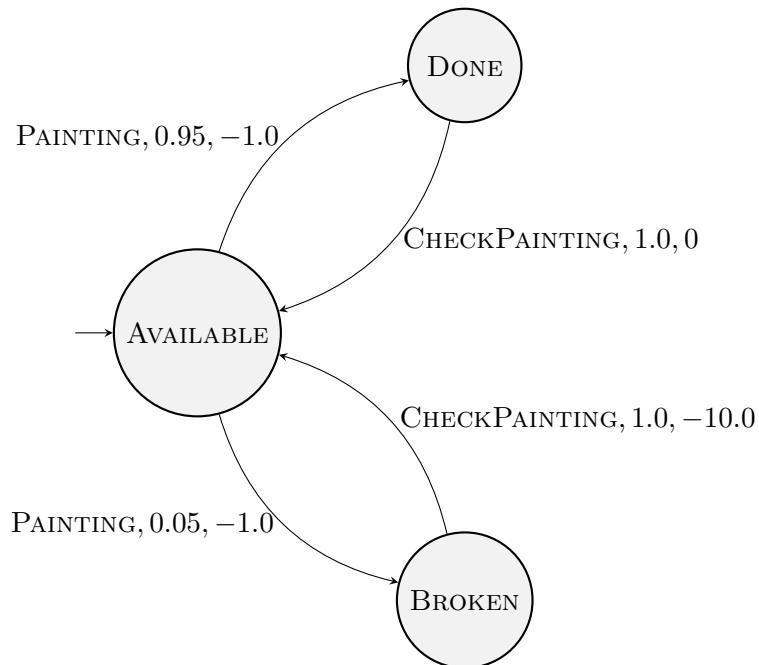


Figure 9.6. Painting automata.

Painting by Human DT:

```
{  
    "thingId": "com.bosch.service:painting_by_human_service",  
    "policyId": "com.bosch.service:painting_by_human_service",  
    "attributes": {  
        "type": "service",  
        "transitions": {  
            "available": {  
                "painting": [  
                    {  
                        "done": 1,  
                        "id": 1  
                    }  
                ]  
            }  
        }  
    }  
}
```

```

        "broken": 0
    },
    -5
]
},
"broken": {
    "check_painting": [
        {
            "available": 1
        },
        0
    ]
},
"done": {
    "check_painting": [
        {
            "available": 1
        },
        0
    ]
}
},
"initial_state": "available",
"final_states": [
    "available"
]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
}
}

```

The painting-by-human DT corresponds to the automata shows in Figure 9.7:

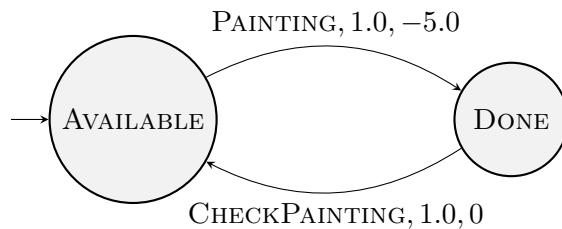


Figure 9.7. Painting-by-human automata.

Note that we have modelled the human service as a complex service with broken probability of 0.0.

Second Baking DT:

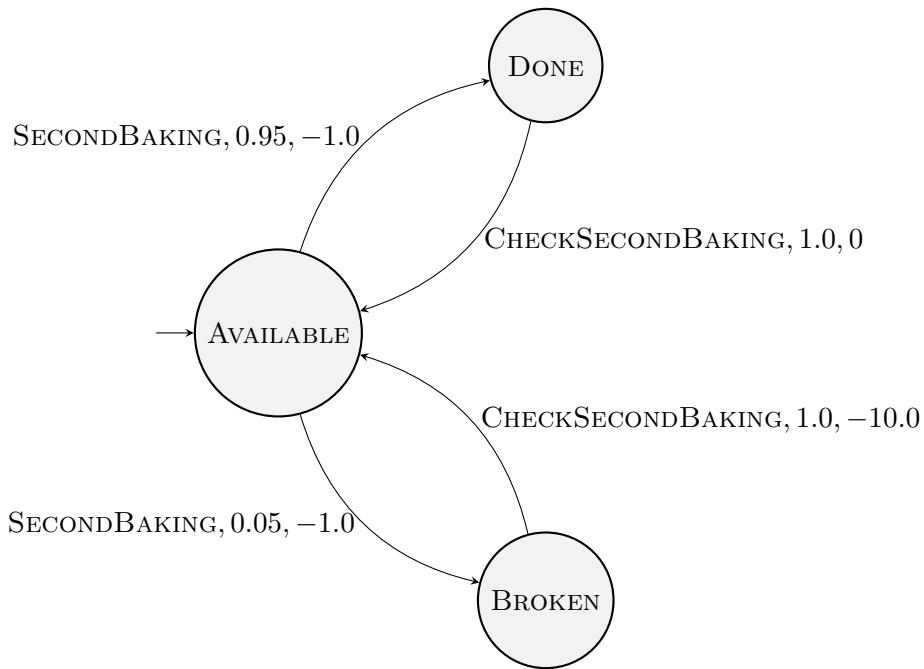
{

```

"thingId": "com.bosch.service:second_baking_service",
"policyId": "com.bosch.service:second_baking_service",
"attributes": {
    "type": "service",
    "transitions": {
        "available": {
            "second_baking": [
                {
                    "done": 0.95,
                    "broken": 0.05
                },
                -1
            ]
        },
        "broken": {
            "check_second_baking": [
                {
                    "available": 1
                },
                -10
            ]
        },
        "done": {
            "check_second_baking": [
                {
                    "available": 1
                },
                0
            ]
        }
    },
    "initial_state": "available",
    "final_states": [
        "available"
    ]
},
"features": {
    "current_state": {
        "properties": {
            "value": "available"
        }
    }
}
}

```

The second-baking DT corresponds to the automata shows in Figure 9.8:

**Figure 9.8.** Second-baking automata.

Shipping DT:

```
{
  "thingId": "com.bosch.service:shipping_service",
  "policyId": "com.bosch.service:shipping_service",
  "attributes": {
    "type": "service",
    "transitions": {
      "available": {
        "shipping": [
          {
            "available": 1
          },
          -1
        ]
      }
    }
  },
  "initial_state": "available",
  "final_states": [
    "available"
  ],
  "features": {
    "current_state": {
      "properties": {
        "value": "available"
      }
    }
  }
}
```

The shipping DT corresponds to the automata shows in Figure 9.9:

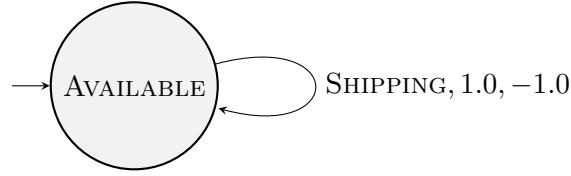


Figure 9.9. Shipping automata.

The target is defined as a Digital Twins too in Bosch IoT Things, reminding the structure illustrated in 8.3. Every time that the target asks an action to the orchestrator and this action is correctly performed, it updates the value of the current action, moving to the next request.

```
{
  "thingId": "com.bosch.service:target",
  "policyId": "com.bosch.service:target",
  "attributes": {
    "type": "target",
    "transitions": {
      "s0": {
        "provisioning": [
          "s1",
          1,
          0
        ]
      },
      "s1": {
        "moulding": [
          "s2",
          1,
          0
        ]
      },
      "s2": {
        "check_moulding": [
          "s3",
          1,
          0
        ]
      },
      "s3": {
        "drying": [
          "s4",
          1,
          0
        ]
      },
      "s4": {
        "check_drying": [
          "s5",
          1,
          0
        ]
      }
    }
  }
}
```

```

},
"s5": {
  "first_baking": [
    "s6",
    1,
    0
  ]
},
"s6": {
  "check_first_baking": [
    "s7",
    1,
    0
  ]
},
"s7": {
  "enamelling": [
    "s8",
    1,
    0
  ]
},
"s8": {
  "check_enamelling": [
    "s9",
    1,
    0
  ]
},
"s9": {
  "painting": [
    "s10",
    1,
    0
  ]
},
"s10": {
  "check_painting": [
    "s11",
    1,
    0
  ]
},
"s11": {
  "second_baking": [
    "s12",
    1,
    0
  ]
},
"s12": {
  "check_second_baking": [
    "s13",
    1,
    0
  ]
}

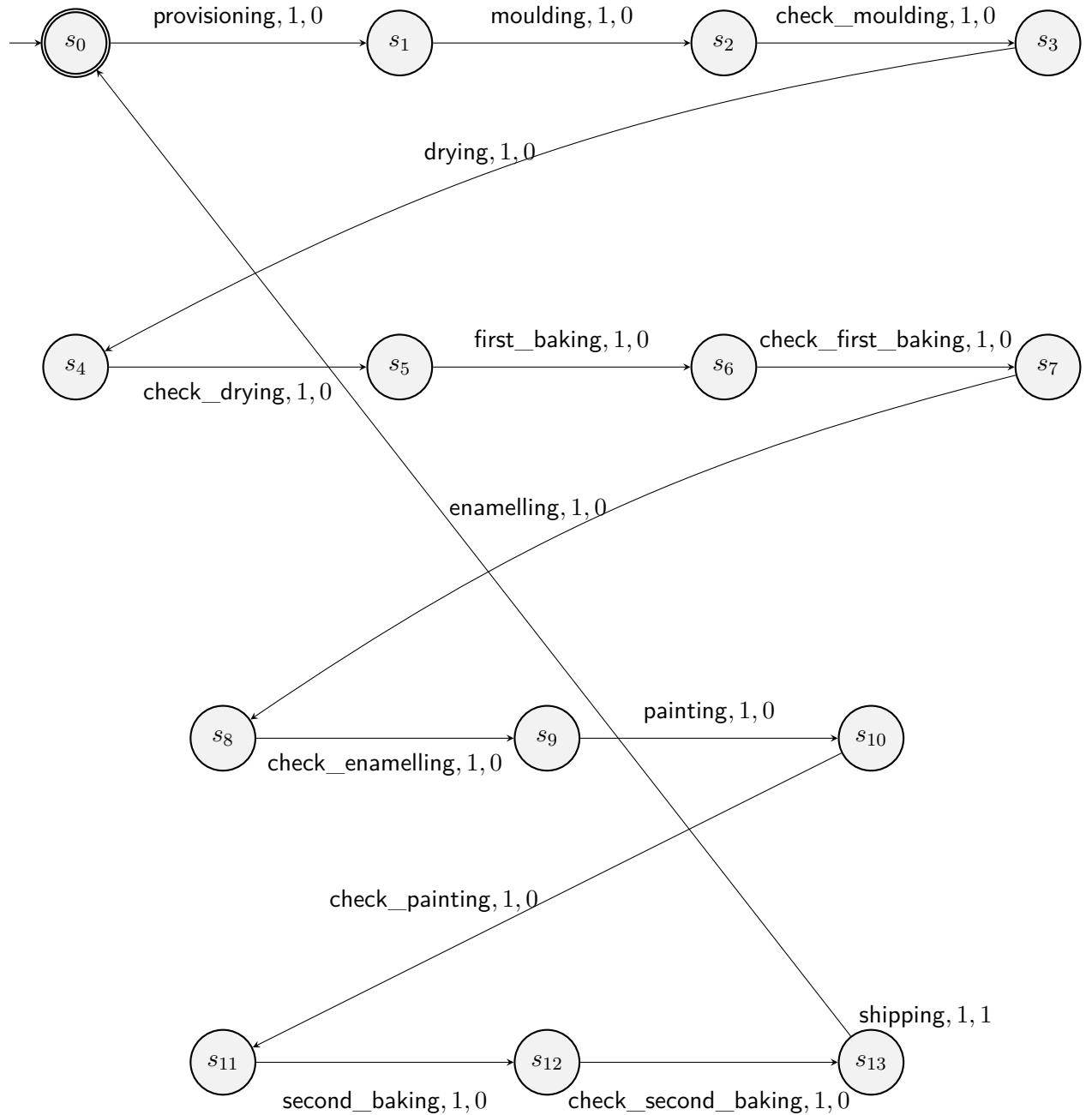
```

```

          0
        ]
      },
    "s13": {
      "shipping": [
        "s0",
        1,
        1
      ]
    },
    "initial_state": "s0",
    "final_states": [
      "s0"
    ]
  },
  "features": {
    "current_action": {
      "properties": {
        "value": "moulding"
      }
    }
  }
}

```

The target DT corresponds to the automata shows in Figure 9.10:

**Figure 9.10.** The state machine of the target.

9.2 Main

In the main is defined the core structure of the project: the orchestration between the Digital Twins, in particular between services and target.

In the first part of the main we download the JSON description of the services and the target, we can do this by searching the things defined on Bosch IoT Things thanks to the API call “search”, as we illustrate in subsection 8.1.4.

```
async def main(config: str, timeout: int):
```

```

    """Run the main."""
services = []
service_ids = []
configuration = config_from_json(Path(config))
api = ThingsAPI(configuration)
data = api.search_services("")
for element in data:
    service = service_from_json(element)
    services.append(service)

    service_ids.append(element["thingId"])

data = api.search_targets("")
assert len(data) == 1
target: Target = target_from_json(data[0])
target_thing_id = data[0]["thingId"]

```

Then, we compose the MDP as illustrated in 7.4 and we calculate the optimal policy as explained in 7.5

```

mdp: MDP = composition_mdp(target, *services)
orchestrator_policy = mdp.get_optimal_policy()

```

After this we establish the connection with Websocket endpoint, the Orchestrator starts to send events to Bosch IoT Things platform, waits for the “ACK” from the platform, if the “ACK” is not received it raises an exception.

```

print("Opening websocket endpoint...")
ws_uri = "wss://things.eu-1.bosch-iot-suite.com/ws/2"
async with websockets.connect(ws_uri,
                               extra_headers=websockets.http.Headers({
                                   'Authorization': 'Bearer ' + api.get_token()
                               })) as websocket:
    print("Collecting problem data...")

    event_cmd = "START-SEND-EVENTS"
    print("EVENT_CMD: ", event_cmd)
    event_cmd = urllib.parse.quote(event_cmd, safe='')
    await websocket.send(event_cmd)
    print("Listening to events originating")
    message_receive = await websocket.recv()
    print("Message received: ", message_receive)
    if message_receive != "START-SEND-EVENTS:ACK":
        raise Exception("Ack not received")

```

From this point the target state is updated and is taken the initial state of the system. The Orchestrator waits from the target action, that tells him the action that the user wants. The user choose the action following the optimal policy (that minimize the cost). Once the target asks for the action it waits for a message from the orchestrator that the action is done. The orchestrator is always updated to the current state of the services, so if it cannot dispatch the action to a service the execution failed. Now the Orchestrator can send the action to the right Thing that can perform it through the API call “send_message_to_thing” as explained in subsection 8.1.4. Once the thing gets response it updates the state after executing the action and sends it to the Orchestrator (that was waiting about it). At this point, the Orchestrator

can send a message to the target that the action is done, since the target is also defined as a thing it can do this through API call “send_message_to_thing” as explained in subsection 8.1.4.

```

iteration = 0
target_simulator = TargetSimulator(target)
system_state = [service.initial_state for service in services]

while True:
    # waiting for target action
    print("Waiting for messages from target...")
    target_message = await websocket.recv()
    target_message_json = json.loads(target_message)
    target_action = target_message_json["value"]
    current_target_state = target_simulator.current_state
    target_simulator.update_state(target_action)
    print(f"Iteration: {iteration}, target action: {target_action}")
    current_state = (tuple(system_state),
                     current_target_state, target_action)

    orchestrator_choice =
        orchestrator_policy.get_action_for_state(current_state)
    if orchestrator_choice == "undefined":
        print(f"Execution failed: no service can execute
              {target_action} in system state {system_state}")
        break

    # send_action_to_service
    service_index = orchestrator_choice
    chosen_thing_id = service_ids[service_index]
    print("Sending message to thing: ", chosen_thing_id,
          target_action, timeout)
    response = api.send_message_to_thing(chosen_thing_id,
                                         target_action, {}, timeout)
    print(f"Got response")
    print("Waiting for update from websocket...")
    message_receive = await websocket.recv()
    print(f"Update after change: {message_receive}")
    json_message = json.loads(message_receive)
    next_service_state = json_message["value"]
    # compute the next system state
    system_state[service_index] = next_service_state

    # send "DONE" to target
    response = api.send_message_to_thing(target_thing_id,
                                         "done", {}, timeout)
    iteration += 1

if __name__ == "__main__":
    arguments = parser.parse_args()
    result = asyncio.get_event_loop().run_until_complete(main(
        arguments.config, arguments.timeout)))

```

9.3 Launch Devices

This script launch the devices (all services and the target), every device is launched in a dedicated process. We can do this thanks to the standard Python library “multiprocessing”.

```
DEVICES = [
    "provisioning_service",
    "drying_service",
    "enamelling_service",
    "moulding_service",
    "first_baking_service",
    "painting_service",
    "painting_by_human_service",
    "second_baking_service",
    "shipping_service",
]
TARGET = "target"

if __name__ == "__main__":
    pool = multiprocessing.Pool(len(DEVICES) + 1)

    run_device_config = partial(run_device, path_to_json=
        CURRENT_DIRECTORY / ".." / "config.json")
    target_result = pool.apply_async(run_device_config, args=[TARGET],
        kwds=dict(is_target=True))
    results = pool.map(run_device_config, DEVICES)

    try:
        for result in results:
            result.get()
        target_result.get()
    except Exception:
        print("Interrupted.")
    pool.terminate()
```

9.4 Execution

According to what was said regarding the communication in chapter 8 and regarding the code in previous section 9.2 and 9.3, we start the software launching the main as Figure 9.11 shows. The orchestrator:

- opens the WebSocket endpoint;
- collects problem data;
- sends events to Bosch IoT Things platform;
- listens to events originating;
- receives the “ACK” from Bosch IoT Things platform;
- waits for the messages from target.

Once launching the main that waits from the target action, we can launch the devices that connect to Bosch IoT Hub through MQTT client, as Figure 9.12 shows:

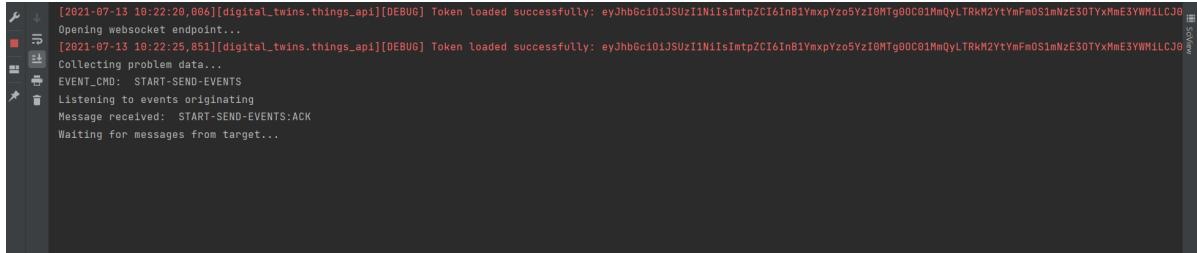


Figure 9.11. Starting software



Figure 9.12. Launching Devices

It is provided the target's update state (for verifying in which state is currently the target) and chosen the command by a hypothetical user (here the choice is simulated).

At this point the communication between the components begins as Figure 9.13 and Figure 9.14 show.

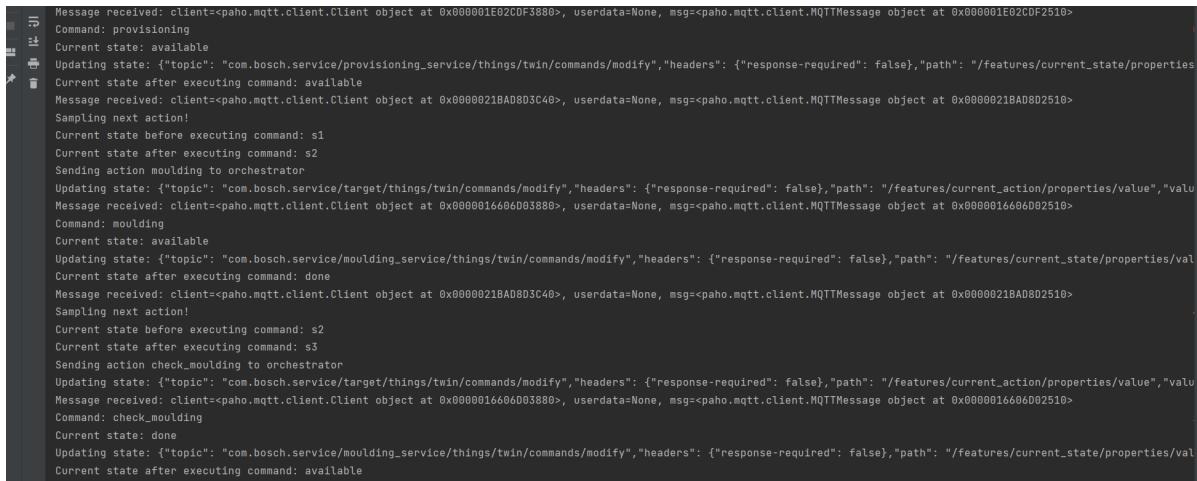


Figure 9.13. Devices flow

- the Orchestrator receives the action from target's device;
- the Orchestrator sends the message to the thing that can perform the action;
- the Thing get the message and it updates the state in which it is after executing the action;
- the Orchestrator gets the state updating; item the Orchestrator send that the action is done to the target;
- it is released the next action from the target and everything starts again as before.

```
Iteration: 0, target action: provisioning
Sending message to thing: com.bosch.service:provisioning_service provisioning 0
Got response
Waiting for update from websocket...
Update after change: {"topic":"com.bosch.service/provisioning_service/things/twin/events/modified","headers":{"response-required":false,"correlation-id":"7f2c1565-46bb-420d-93d6-000000000000"}
Waiting for messages from target...
Iteration: 1, target action: moulding
Sending message to thing: com.bosch.service:moulding_service moulding 0
Got response
Waiting for update from websocket...
Update after change: {"topic":"com.bosch.service/moulding_service/things/twin/events/modified","headers":{"response-required":false,"correlation-id":"c7c6387c-9c1b-4aee-a61c-add6-000000000000"}
Waiting for messages from target...
Iteration: 2, target action: check_moulding
Sending message to thing: com.bosch.service:moulding_service check_moulding 0
Got response
Waiting for update from websocket...
Update after change: {"topic":"com.bosch.service/moulding_service/things/twin/events/modified","headers":{"response-required":false,"correlation-id":"68da03aa-4534-4e4a-abc6-0931-000000000000"}
Waiting for messages from target...
Iteration: 3, target action: drying
Sending message to thing: com.bosch.service:drying_service drying 0
```

Figure 9.14. Target-Orchestrator flow

Chapter 10

Conclusion and Future Works

This chapter summarizes the thesis and offers conclusions and the future directions of research. The chapter is structured as follows:

- In **Section 10.1**: we present an overview of the most important concepts addressed in the thesis;
- In **Section 10.2**: we define the most important remarks of the framework;
- In **Section 10.2**: we list all the future works that can be done with this project.

10.1 Overview

This work underline the importance of Digital Twins as enabling technology in Industry 4.0. The use of Digital Twins allows a connection with AI techniques such as planning over Markov Decision Processes. In particular, we have seen that we have been able to model devices as stochastic non-deterministic services with the possibility of failure. In this way when a machine is wearing out we can understand it from the high probability of ending up in a broken state. If a machine breaks, it can be repaired and become available again but all this has a high cost, which will be represented by negative rewards.

Also the target have been modelled as a stochastic non-deterministic services and represents the choice of the user. Obviously, in an Industry 4.0 scenario the target process of production will be linear and one action at the time can be done. Through techniques based on probabilistic reasoning and MDP it is possible to build an optimal orchestrator with respect to the optimal policy. This orchestrator guarantees that: if a policy exists it will be found and it is optimal compared to costs; if the composition does not exist in every case it will try to maximize rewards. The orchestrator in a certain sense defines the strategy to be adopted and for each action coming from the target dispatches it to the right service that can perform it.

Everything that we have identified has been implemented in a proof-of-concept that has allowed us to verify how these systems work in practice. For this purpose we have also presented a simple case study which, even if simple, is directly linked to real problems in Industry 4.0.

10.2 Remarks

The manufacturing process itself, the involved devices, and how they interact, is designed till now by human experts in a traditional way. In the thesis we envision an architecture where humans can instead specify a goal and take advantage of technologies such as digital twins to automatically compose the corresponding physical processes, sharing some analogies with the

notion of Web service composition. The technologies used in the thesis for doing this are very innovative, in particular the use of Digital Twins has crucial importance on the Industry 4.0 world. The implementation of the project is done in Python language, while the implementation of Digital Twins is made through Bosch IoT platform in JSON description language. What is interesting to see is how the orchestrator implemented in Python communicates with Bosch IoT Things and vice versa.

Although the framework is implemented in a “normal” computer machine, it adheres perfectly in an Industry 4.0 prototype and can be executing in an Digital Twins industrial context. This solution is very powerful because permits to have non-deterministic stochastic available services and also in the use of the target. Another important aspect is having a feedback from the services when an action is performed correctly, this feedback doesn’t arrive to the target in fact the target doesn’t know who execute its action. This is at the base of smart manufacturing, because creates a decoupling between the user that use the target and how it is actually realized.

10.3 Future works

There are many future directions that can be taken, due to the novelty of the work. Some of them are:

- it could be interesting to put the framework developed in a real industry. Our thesis give an important contribution to this because it can already still runs on a computer/cloud that is in charge of orchestrating; but it is also interesting that this orchestration takes place in a specific environment for DTs and in fact in the literature this type of environment is being studied;
- *Handling exceptions*: our current model does not explicitly capture a critical aspect of many real-world scenarios, exception handling: if the target/composite service terminates before a terminal state has been reached, work done so far has to be undone. This work is distributed across different services. For example, if while booking a vacation, we book a flight but cannot book a hotel, we must cancel the flight reservation, which can be costly. If we also booked a car by now, the cost would be higher. We can augment the MDP defined earlier to take these costs into account by adding a negative reward to states (s_z, s_t, a) and service choice i such that i cannot supply action a in its current state. The size of the reward can depend on the states of the various services, as reflected in s_z , which reflects the work that needs to be undone in each of the existing services;
- *Separate rewards specifications*: in the setting considered here, we have coupled the rewards with the likelihood of the client making certain action requests into the target service to be realized. In fact it may be convenient to keep the two specification separated, and use the target service only to specify the likelihood of action request, in line with what happens in the deterministic case. Rewards in this case could be expressed dynamically on the history of actions executed so far by the target, through a transducer. More precisely a transducer $R = (\Sigma, \Delta, S, s_0, f, g)$ is a deterministic transition system with inputs and outputs, where Σ is the input alphabet, Δ is the output alphabet, S is the set of states, s_0 the initial state, $f : S \times \Sigma \rightarrow S$ is the transition function (which takes a state and an input symbol and returns the successor state) and $g : S \times \Sigma \rightarrow \Delta$ is the output function (which returns the output of the transition).

In our case the input alphabet would be the set of actions A , the output alphabet the possible rewards expressed as reals R . In this way the output function $g : S \times A \rightarrow R$, would correspond the reward function. The point is that now the rewards do not depend on the state of the target, but on the sequence of actions executed so far. Interestingly if we take the synchronous product of the target T (without rewards, but with stochastic transitions) and of R (which is deterministic but outputs rewards), we get a target of the form specified in Section 5.3 though this time computed from the two separated

specifications, and we can apply the MDP construction presented here (or its extension with stochastic available services discussed previously).

- In line with the above point, it has long been observed that many performance criteria call for more sophisticated reward functions that do not depend on the last state only. For example, in Robotics, we may want to reward a robot for picking up a cup only if it was requested to do so earlier, where the pick-up command may have been given a number of steps earlier. Similarly, we may want to reward an agent for behavior that is conditional on some past fact – for example, if the person was identified as a child earlier, we must provide her with food rich in protein, and if he is older, in food low in sodium.

All these proposals share the idea of specifying rewards on (partial traces or histories) through some variant of linear-time temporal logic over finite traces LTL_f . The research on variants of LTL_f has become very lively lately with promising results. A key point is that formulas in these logics can be “translated” into standard deterministic finite state automata DFAs that recognize exactly the traces that fulfill the formula. Such DFAs can be combined with probabilistic transition systems to generate suitable MDPs to be used for generating optimal solutions. This can be done also in our context. Essentially we replace (or enhance) the target specification with a declarative set of logical constraints. Then we compute the synchronous product with a target transition system that us the likelihood of action choice, hence getting a target specification as that of Section 5.3, analogously to the case of the transducer above. This can be solved by the techniques presented earlier.

- *Learning*: although we focus on this thesis on model specification and model-based solution techniques, we point out that for Web services and Industry 4.0, statistics gathering is very simple, and in fact, is carried out routinely nowadays. Consequently, it is not difficult to learn the stochastic transition function of existing services online, and use it to specify the probabilistic elements of the model
- *Safety*: next to the target we can put safety specifications, so that if the system, wherever it evolves, cannot go into dangerous states and it cannot happen that it is guided in a direction that is not correct.
- *Resilience*: in the techniques described in this thesis, the system is usually assumed to work in normal conditions. However, it is also important that the systems work in limit condition. If the system is operating and something strange happens, in any case it "holds", i.e. it resists at these perturbations that bring it to the limit. In our case we have resilience when for example a machine ends up in a broken state, then in a more or less short time is repaired. We can improve this aspect, instead of turn-off if the system is not optimal, it goes to the next element that makes it better than the previous one.

Bibliography

- Kevin Ashton et al. That ‘internet of things’ thing. *RFID journal*, 22(7):97–114, 2009.
- Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-services that export their behavior. In *ICSO*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.
- Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, volume 5, pages 613–624, 2005.
- Jon Bokrantz, Anders Skoogh, Cecilia Berlin, and Johan Stahre. Maintenance in digitalised manufacturing: Delphi-based scenarios for 2030. *International Journal of Production Economics*, 191:154–169, 2017.
- Ruth N Bolton, Janet R McColl-Kennedy, Lilliemay Cheung, Andrew Gallan, Chiara Orsingher, Lars Witell, and Mohamed Zaki. Customer experience challenges: bringing together digital, physical and social realms. *Journal of Service Management*, 2018.
- Ronen I Brafman, Giuseppe De Giacomo, Massimo Mecella, and Sebastian Sardina. Service composition in stochastic settings. In *Conference of the Italian Association for Artificial Intelligence*, pages 159–171. Springer, 2017.
- Jakob Branger and Zhibo Pang. From automated home to sustainable, healthy and manufacturing home: a new story enabled by the internet-of-things and industry 4.0. *Journal of Management Analytics*, 2(4):314–332, 2015.
- Jorge S. Cardoso and Amit P. Sheth. Introduction to semantic web services and web process composition. In *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2004.
- Tiziana Catarci, Donatella Firmani, Francesco Leotta, Federica Mandreoli, Massimo Mecella, and Francesco Sapi. A conceptual architecture and model for smart manufacturing relying on service-based digital twins. In *2019 IEEE international conference on web services (ICWS)*, pages 229–236. IEEE, 2019.
- Yubao Chen. Integrated and intelligent manufacturing: perspectives and enablers. *Engineering*, 3(5):588–595, 2017.
- Guo-Jian Cheng, Li-Ting Liu, Xin-Jian Qiang, and Ye Liu. Industry 4.0 development and application of intelligent manufacturing. In *2016 international conference on information system and artificial intelligence (ISAI)*, pages 407–410. IEEE, 2016.
- Li Da Xu. Enterprise systems: state-of-the-art and future trends. *IEEE Transactions on Industrial Informatics*, 7(4):630–640, 2011.

- Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.
- Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardiña. Automatic behavior composition synthesis. *Artif. Intell.*, 196:106–142, 2013.
- Giuseppe De Giacomo, Massimo Mecella, and Fabio Patrizi. Automated service composition based on behaviors: The roman model. In *Web Services Foundations*, pages 189–214. Springer, 2014.
- PC P De Silva and PC A De Silva. Ipanera: An industry 4.0 based architecture for distributed soil-less food production systems. In *2016 Manufacturing & Industrial Engineering Symposium (MIES)*, pages 1–5. IEEE, 2016.
- Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.
- Adam Dziedzic, Aaron J Elmore, and Michael Stonebraker. Data transformation and migration in polystores. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.
- Abdulmotaleb El Saddik. Digital twins: The convergence of multimedia technologies. *IEEE multimedia*, 25(2):87–92, 2018.
- Edward Glaessgen and David Stargel. The digital twin paradigm for future nasa and us air force vehicles. In *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*, page 1818, 2012.
- Michael Grieves. *Virtually perfect: Driving innovative and lean products through product lifecycle management*. Space Coast Press, 2011.
- Mario Hermann, Tobias Pentek, and Boris Otto. Design principles for industrie 4.0 scenarios. In *2016 49th Hawaii international conference on system sciences (HICSS)*, pages 3928–3937. IEEE, 2016.
- Erik Hofmann and Marco Rüsch. Industry 4.0 and the current status as well as future prospects on logistics. *Computers in industry*, 89:23–34, 2017.
- Yuxiao Hu and Giuseppe De Giacomo. A generic technique for synthesizing bounded finite-state controllers. In *ICAPS*. AAAI, 2013.
- Richard Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM Conferences (2)*, volume 5332 of *Lecture Notes in Computer Science*, pages 1152–1163. Springer, 2008.
- Ludek Janak and Zdenek Hadas. Machine tool health and usage monitoring system: An initial analyses. *MM Science Journal*, 4:794–798, 2015.
- Jürgen Jasperneite. Was hinter begriffen wie industrie 4.0 steckt. *Computer & Automation*, 19, 2012.
- Okyay Kaynak. The exhilarating journey from industrial electronics to industrial informatics. In *2007 2nd IEEE Conference on Industrial Electronics and Applications*, pages xli–xlvi. IEEE, 2007.

- Jin Ho Kim. A review of cyber-physical system research relevant to the emerging it trends: industry 4.0, iot, big data, and cloud computing. *Journal of industrial integration and management*, 2(03):1750011, 2017.
- Branko Kolarevic. *Architecture in the digital age: design and manufacturing*. taylor & Francis, 2004.
- Werner Kritzinger, Matthias Karner, Georg Traar, Jan Henjes, and Wilfried Sihn. Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022, 2018.
- Andrew Kusiak. Smart manufacturing must embrace big data. *Nature News*, 544(7648):23, 2017.
- Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing letters*, 3:18–23, 2015.
- Zheng Liu, Norbert Meyendorf, and Nezih Mrad. The role of data fusion in predictive maintenance using digital twin. In *AIP Conference Proceedings*, volume 1949, page 020023. AIP Publishing LLC, 2018.
- Antonio Maccioni and Riccardo Torlone. Augmented access for querying and exploring a poly-store. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 77–88. IEEE, 2018.
- Azad M Madni and Michael Sievers. Model-based systems engineering: Motivation, current status, and research opportunities. *Systems Engineering*, 21(3):172–190, 2018.
- Azad M Madni, Carla C Madni, and Scott D Lucero. Leveraging digital twin technology in model-based systems engineering. *Systems*, 7(1):7, 2019.
- Oleg Makarov, Reinhard Langmann, S Nesterenko, and B Frank. Problems of the time deterministic in applications for process control from the cloud. *International Journal of Online Engineering*, 10(4), 2014.
- Jia Mao, Qin Zhou, MD Sarmiento, Jun Chen, P Wang, F Jonsson, LD Xu, LR Zheng, and Zhuo Zou. A hybrid reader tranceiver design for industrial internet of things. *Journal of Industrial Information Integration*, 2:19–29, 2016.
- Mousa Marzband, Narges Parhizi, Mehdi Savaghebi, and Josep M Guerrero. Distributed smart decision-making for a multimicrogrid system based on a hierarchical interactive architecture. *IEEE Transactions on Energy Conversion*, 31(2):637–648, 2015.
- Sheila A. McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In *KR*, pages 482–496. Morgan Kaufmann, 2002.
- Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12(4):333–351, 2003.
- Arnab Mitra, Anirban Kundu, Matangini Chattopadhyay, and Samiran Chattopadhyay. A cost-efficient one time password-based authentication in cloud environment using equal length cellular automata. *Journal of Industrial Information Integration*, 5:17–25, 2017.
- Mohsen Moghaddam and Shimon Y Nof. Collaborative service-component integration in cloud manufacturing. *International Journal of Production Research*, 56(1-2):677–691, 2018.
- Anca Muscholl and Igor Walukiewicz. A lower bound on web services composition. *Log. Methods Comput. Sci.*, 4(2), 2008.

- Steffen Nienke, Hendrik Frölian, Violett Zeller, and Günther Schuh. Energy-management 4.0: roadmap towards the self-optimising production of the future. In *Proceedings of the 6th International Conference on Informatics, Environment, Energy and Applications*, pages 6–10, 2017.
- Tim Niesen, Constantin Houy, Peter Fettke, and Peter Loos. Towards an integrative big data analysis framework for data-driven risk management in industry 4.0. In *2016 49th Hawaii international conference on system sciences (HICSS)*, pages 5065–5074. IEEE, 2016.
- Nikolaos Papakostas, Konstantinos Efthymiou, Konstantinos Georgoulias, and George Chrys-solouris. On the configuration and planning of dynamic manufacturing networks. In *Robust Manufacturing Control*, pages 247–258. Springer, 2013.
- Olivia Penas, Régis Plateaux, Stanislao Patalano, and Moncef Hammadi. Multi-scale approach from mechatronic to cyber-physical systems for the design of manufacturing systems. *Computers in Industry*, 86:52–69, 2017.
- Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259. Professional Book Center, 2005.
- Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.
- Qinglin Qi and Fei Tao. Digital twin and big data towards smart manufacturing and industry 4.0: 360 degree comparison. *Ieee Access*, 6:3585–3593, 2018.
- Frank Rennung, Caius Tudor Luminosu, and Anca Draghici. Service provision in the framework of industry 4.0. *Procedia-Social and Behavioral Sciences*, 221:372–377, 2016.
- Alfredo Alan Flores Saldivar, Yun Li, Wei-neng Chen, Zhi-hui Zhan, Jun Zhang, and Leo Yi Chen. Industry 4.0 with cyber-physical integration: A design and manufacture perspective. In *2015 21st international conference on automation and computing (ICAC)*, pages 1–6. IEEE, 2015.
- Michael Schluse, Linus Atorf, and Juergen Rossmann. Experimentable digital twins for model-based systems engineering and simulation-based development. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–8. IEEE, 2017.
- Jianwen Su. Semantic web services: Composition and analysis. *IEEE Data Engineering Bulletin*, 31(3):2, 2008.
- Lu Tan and Neng Wang. Future internet: The internet of things. In *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, volume 5, pages V5–376. IEEE, 2010.
- Fei Tao, Jiangfeng Cheng, Qinglin Qi, Meng Zhang, He Zhang, and Fangyuan Sui. Digital twin-driven product design, manufacturing and service with big data. *The International Journal of Advanced Manufacturing Technology*, 94(9):3563–3576, 2018.
- Lane Thamess and Dirk Schaefer. Software-defined cloud manufacturing for industry 4.0. *Procedia cirp*, 52:12–17, 2016.
- Anitha Varghese and Deepaknath Tandur. Wireless requirements and challenges in industry 4.0. In *2014 international conference on contemporary computing and informatics (IC3I)*, pages 634–638. IEEE, 2014.
- Li Wang, Li Da Xu, Zhuming Bi, and Yingcheng Xu. Data cleaning for rfid and wsn integration. *IEEE transactions on industrial informatics*, 10(1):408–418, 2013.

- A Weber. Industry 4.0: myths vs. reality. *Assembly Magazine*, pages 28–37, 2016.
- Dan Wu, Bijan Parsia, Evren Sirin, James A. Hendler, and Dana S. Nau. Automating DAML-S web services composition using SHOP2. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2003.
- Tangbin Xia and Lifeng Xi. Manufacturing paradigm-oriented phm methodologies for cyber-physical systems. *Journal of Intelligent Manufacturing*, 30(4):1659–1672, 2019.
- Li Da Xu, Eric L Xu, and Ling Li. Industry 4.0: state of the art and future trends. *International Journal of Production Research*, 56(8):2941–2962, 2018.
- Xun Xu. Machine tool 4.0 for the new era of manufacturing. *The International Journal of Advanced Manufacturing Technology*, 92(5):1893–1900, 2017.
- Nitin Yadav and Sebastian Sardiña. Decision theoretic behavior composition. In *AAMAS*, pages 575–582. IFAAMAS, 2011.
- Jian Yang and Mike P. Papazoglou. Service components for managing the life-cycle of service compositions. *Inf. Syst.*, 29(2):97–125, 2004.
- Yale Yu and Sharma Madiraju. Enterprise application transformation strategy and roadmap design: A business value driven and it supportability based approach. In *2014 Enterprise Systems Conference*, pages 66–71. IEEE, 2014.
- Przemysław Zawadzki and Krzysztof Żywicki. Smart product design and production control for effective mass customization in the industry 4.0 concept. *Management and production engineering review*, 7, 2016.
- Pai Zheng, Zhiqian Sang, Ray Y Zhong, Yongkui Liu, Chao Liu, Khamdi Mubarok, Shiqiang Yu, Xun Xu, et al. Smart manufacturing systems for industry 4.0: Conceptual framework, scenarios, and future perspectives. *Frontiers of Mechanical Engineering*, 13(2):137–150, 2018.
- Xianrong Zheng, Patrick Martin, Kathryn Brohman, and Li Da Xu. Cloud service negotiation in internet of things environment: A mixed approach. *IEEE Transactions on Industrial Informatics*, 10(2):1506–1515, 2014.
- Ray Y Zhong, QY Dai, Ting Qu, GJ Hu, and George Q Huang. Rfid-enabled real-time manufacturing execution system for mass-customization production. *Robotics and Computer-Integrated Manufacturing*, 29(2):283–292, 2013a.
- Ray Y Zhong, Z Li, LY Pang, Y Pan, Ting Qu, and George Q Huang. Rfid-enabled real-time advanced planning and scheduling shell for production decision making. *International Journal of Computer Integrated Manufacturing*, 26(7):649–662, 2013b.
- Ray Y Zhong, George Q Huang, Shulin Lan, QY Dai, T Zhang, and Chen Xu. A two-level advanced production planning and scheduling model for rfid-enabled ubiquitous manufacturing. *Advanced Engineering Informatics*, 29(4):799–812, 2015.