KTH ROYAL INSTITUTE
OF TECHNOLOGY

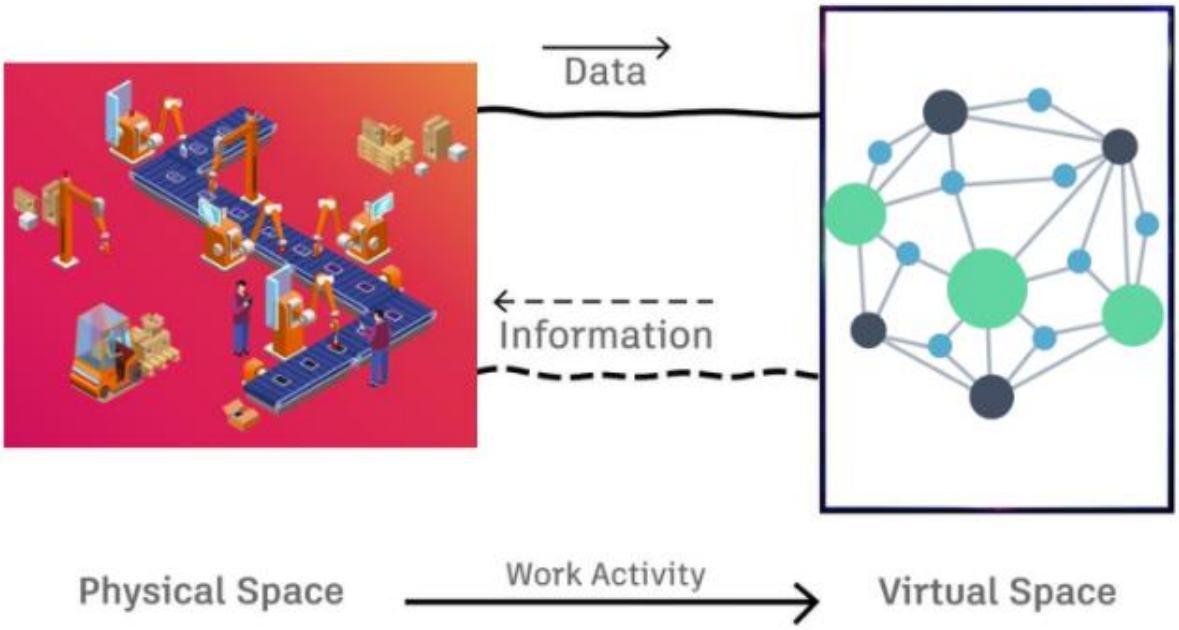
Degree Project in School of Electrical Engineering and Computer Science

Second Cycle 30 Credits

Digital Twin Knowledge Graphs for IoT Platforms

Towards a Virtual Model for Real-Time Knowledge Representation in IoT Platforms

ALEJANDRO JARABO PEÑAS



Stockholm, Sweden 2023

Digital Twin Knowledge Graphs for IoT Platforms

Towards a Virtual Model for Real-Time Knowledge Representation in IoT Platforms

ALEJANDRO JARABO PEÑAS

Master's Programme, Systems, Control and Robotics, 120 credits
Date: April 11, 2023

Supervisors: Bin Xiao, Bo Wahlberg, Pedro José Zufiria Zatarain
Examiner: Cristian Rojas
School of Electrical Engineering and Computer Science
Host company: Ericsson
Swedish title: Digital Twin Kunskapsgrafer för IoT-Plattformar
Swedish subtitle: Mot en Virtuell Modell för Kunskapsrepresentation i Realtid i IoT-Plattformar

Abstract

This thesis presents the design and prototype implementation of a digital twin based on a knowledge graph for Internet of Things (IoT) platforms. The digital twin is a virtual representation of a physical object or system that must continually integrate and update knowledge in rapidly changing environments. The proposed knowledge graph is designed to store and efficiently query a large number of IoT devices in a complex logical structure, use rule-based reasoning to infer new facts, and integrate unanticipated devices into the existing logical structure in order to adapt to changing environments. The digital twin is implemented using the open-source TypeDB knowledge graph and tested in a simplified automobile production line environment. The main focus of the work is on the integration of unanticipated devices, for which a similarity metric is implemented to identify similar existing devices and determine the appropriate integration into the knowledge graph. The proposed digital twin knowledge graph is a promising solution for managing and integrating knowledge in rapidly changing IoT environments, providing valuable insights and support for decision-making.

Keywords

Internet of Things, Digital Twin, Knowledge Graph, Similarity Metric, Semantic Data Integration.

Sammanfattning

I den här avhandlingen presenteras utformningen och prototypimplementeringen av en digital tvilling baserad på en kunskapsgraf för IoT-plattformar (Internet of Things). Den digitala tvillingen är en virtuell representation av ett fysiskt objekt eller system som måste integrera och uppdatera kunskap i snabbt föränderliga miljöer. Den föreslagna kunskapsgrafen är utformad för att lagra och effektivt söka efter en stor uppsättning IoT-enheter i en komplex logisk struktur, använda regelbaserade resonemang för att härleda nya fakta och integrera oväntade enheter i den befintliga logiska strukturen för att anpassa sig till föränderliga miljöer. Den digitala tvillingen genomförs med hjälp av kunskapsgrafen TypeDB med öppen källkod och testas i en förenklad miljö för bilproduktion. Huvudfokus ligger på integrationen av oväntade enheter, för vilka ett likhetsmått implementeras för att identifiera liknande befintliga enheter och bestämma lämplig integration i kunskapsgrafen. Den föreslagna kunskapsgrafen för digitala tvillingar är en lovande lösning för att hantera och integrera kunskap i snabbt föränderliga IoT-miljöer, vilket ger värdefulla insikter och stöd för beslutsfattande.

Nyckelord

Internet of Things, Digital Twin, Kunskapsgraf, Likhetmetrik, Integrering av Semantiska Data.

Resumen

Esta tesis presenta el diseño e implementación de un prototipo de gemelo digital basado en un grafo de conocimiento para plataformas de Internet de las Cosas (IoT). El gemelo digital es una representación virtual de un objeto o sistema físico que debe integrar y actualizar continuamente el conocimiento en entornos que cambian rápidamente. El grafo de conocimiento propuesto está diseñado para almacenar y consultar eficientemente un gran número de dispositivos IoT en una estructura lógica compleja, utilizar el razonamiento basado en reglas para inferir nuevos hechos e integrar dispositivos imprevistos en la estructura lógica existente para adaptarse a los cambios del entorno. El gemelo digital se implementa utilizando el grafo de conocimiento de código abierto TypeDB y se prueba en un entorno simplificado basado en una línea de producción de automóviles. El objetivo principal del trabajo es la integración de dispositivos no previstos, para lo cual se implementa una métrica de similitud para identificar dispositivos existentes similares y determinar la integración adecuada en el grafo de conocimiento. El grafo de conocimiento propuesto es una solución prometedora para la gestión del conocimiento y la integración en entornos IoT que cambian rápidamente, proporcionando información valiosa y apoyo a la toma de decisiones.

Palabras claves

Internet of Things, Gemelo Digital, Grafo de Conocimiento, Métrica de Similitud, Integración de Datos Semánticos.

Acknowledgments

I am really grateful to my supervisor at Ericsson, Bin Xiao, for his great support and guidance throughout this project. He has helped me to grow both academically and professionally.

I also want to extend my thanks to my academic supervisors, Cristian Rojas and Pedro J. Zufiria Zatarain, for their invaluable help throughout the process and editing of my thesis document. Without their support, this thesis would not have been possible.

I am also grateful to Ericsson for providing me with the resources and support that I needed to complete this thesis. It has been a true privilege to work with such a renowned company.

I am forever grateful for the love and support of my family and friends, and I cannot thank them enough for their unwavering support during this journey.

Lastly, I am incredibly grateful for the experience and knowledge that I have gained from working on this thesis. It has been a fantastic journey, and I am excited to see where it takes me next.

Stockholm, April 2023
Alejandro Jarabo Peñas

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

AI Artificial Intelligence

DT Digital Twin

IoT Internet of Things

KG Knowledge Graph

MQTT MQ Telemetry Transport

NLP Natural Language Processing

SDF Semantic Definition Format

UUID Universally Unique Identifier

Chapter 1

Introduction

The concept of a digital twin has gained increasing attention in recent years as a powerful tool for improving the efficiency and effectiveness of various industries, such as manufacturing, healthcare, and transportation. A digital twin [1] is a virtual representation of a physical system that continually integrates and updates knowledge in rapidly changing environments. It allows for real-time monitoring, prediction, and optimization of the system, leading to improved decision-making and reduced downtime.

Creating and updating the digital twin of an IoT platform [2] is particularly challenging due to the large number of interconnected devices that generate vast amounts of data in real-time. A knowledge graph [3], with its ability to store and organize complex data in a logical structure, is well-suited for representing the digital twin of an IoT platform.

In this work, we present the design of a digital twin knowledge graph for IoT platforms. The aim of this knowledge graph is to store a large number of IoT devices in a complex logical structure, allowing for efficient querying of data to retrieve specific information and using rule-based reasoning to infer new facts from explicit statements. Additionally, the knowledge graph is designed to seamlessly integrate unanticipated IoT devices into the existing logical structure, enabling it to adapt to changing environments.

To implement the digital twin, we utilize the open-source TypeDB knowledge graph [4] and design a test environment based on a simplified automobile production line. The test environment emulates a set of IoT devices that generate behavior or data that is highly dependent on the task being performed and the device class. These devices communicate using the MQTT protocol [5], sending messages in real-time. The consistency handler algorithm processes these messages to ensure that the digital twin knowledge graph accurately reflects the state of its physical counterpart.

The main focus of this work is on the integration of unanticipated devices into the existing logical structure of the knowledge graph. To achieve this, we have implemented a similarity metric for IoT devices. When a new, unanticipated device appears, the most similar existing devices are identified and their integration within the graph is assessed to infer how the new device should be integrated. If the similarity is high enough, we assume that the new device is a complementary device or a replacement for another device within one of the tasks. If the similarity is not high enough, we may need to create a new branch in the knowledge graph to integrate the new device.

By effectively integrating unanticipated devices into the digital twin knowledge graph, we can improve the adaptability and real-time accuracy of the digital twin, leading to better decision-making and improved system performance.

1.1 Background

The thesis focuses on designing a Digital Twin (DT) that accurately reflects an Internet of Things (IoT) device platform. Therefore, a good understanding of the fields of knowledge representation and reasoning in Artificial Intelligence (AI), as well as a familiarity with DTs, is essential.

As the DT needs to store and represent knowledge about the IoT device platform, and allow us to update or retrieve this knowledge with queries, it is natural to implement it as a semantic graph database or KG. This requires a way to semantically describe the IoT device classes that will participate in the data flow, to facilitate their definition and integration into the DT structure. To do this, we have used the Semantic Definition Format (SDF) standard for data and interactions of things to semantically describe each of the device classes involved in the IoT platform.

Additionally, since the physical system the DT reflects is an IoT device platform using the MQ Telemetry Transport (MQTT) messaging protocol, it is important to understand what an IoT device platform is and the basics of how the MQTT protocol works.

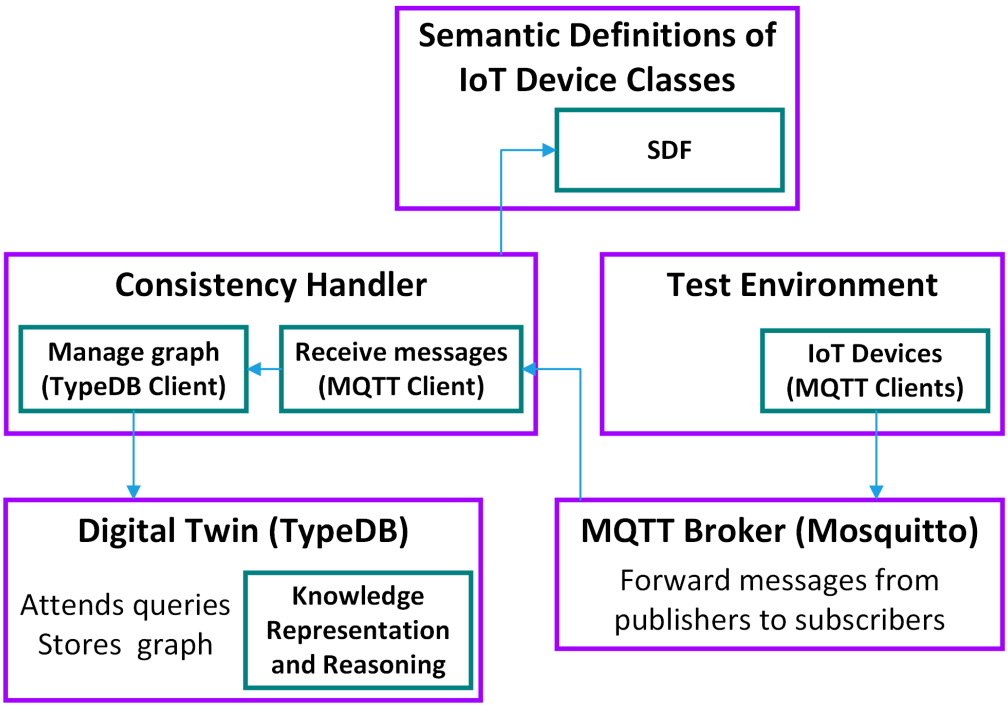


Figure 1.1: Overview of key background concepts and their connections.

The relationships between key background concepts in the research are shown in Figure 1.1. The test environment simulates IoT devices and sends messages via the MQTT protocol. These messages are processed by the consistency handler, which retrieves the semantic definition of the device class in SDF format if the class is unknown and updates the KG using the information in the messages. In Chapter 2, we will explain these concepts in detail and discuss related work and its impact on our research.

1.2 Problem

The thesis focuses on the design of a consistency handler algorithm for IoT device platforms. The consistency handler is responsible for managing the real-time data flow of the network and ensuring that the DT accurately reflects the current state of the IoT platform. The challenge is in designing the consistency handler to effectively update the DT in a way that:

1. Stores a large number of IoT devices as a complex logical structure.
2. Queries data efficiently to retrieve specific insights.
3. Applies rule-based reasoning to infer new facts from those explicitly stated.
4. Integrates unforeseen IoT devices into the logical structure.

The first two requirements fall under the field of knowledge representation and require the implementation of the DT as a semantic graph database or KG. A KG allows us to define an ontology representing domain-specific terminology and defining the types of objects, how they are related, and the attributes they possess. It also provides a query language to retrieve and update the graph structure and information.

The third requirement falls under the field of knowledge reasoning and requires a logical reasoning engine that infers new facts from those explicitly stated by using a set of rules. Reasoning engines are also often built-in in KGs.

The focus of the thesis is on the fourth requirement, the integration of unforeseen IoT devices into the KG. When a new device is added to the IoT platform, it begins to publish messages to the MQTT network along with the rest of the devices that are already integrated into the KG. The problem is then to analyze the messages sent by the new device to determine how it should be related to the other devices in the KG.

1.3 Purpose

The purpose of the project is to construct a real-time virtual model of an IoT platform in a fully automated way, represented as a complex logical structure with reasoning capabilities. This is very attractive because IoT platforms often have a large number of devices, making it impractical for an operator to track these devices and how they are related. The constructed virtual model can be used to analyze the behavior of the platform, detecting issues such as disconnected, malfunctioning, replaced, or newly installed devices within the platform, and integrating the changes automatically. This model can also be used to extract meaningful insights, such as the energy consumption of devices that perform a specific task, as well as triggering alarms in case of major issues.

For example, consider an IoT platform corresponding to an entire smart city. It is easy to see how managing and updating the model with operators becomes impractical. The virtual model would handle issues such as the replacement of sensors in a crosswalk and the installation of new automated lights. It would also alert the responsible parties in case of major malfunctioning issues, such as the loss of synchronization between streetlights.

1.4 Goals

To achieve our desired purpose, we will design a fully automated consistency handler algorithm that can accurately model the IoT device platform as a KG. This project has

been divided into three sub-goals: designing the test environment that will emulate the IoT platform (the physical object), designing the virtual model based on a KG (the DT), and implementing the consistency handler algorithm.

1.4.1 Test environment design

Before we can design the consistency handler, we need to first decide on and develop the physical object or IoT platform that we want to virtually model. This process involves several steps:

1. Choosing the scenario of our IoT device platform (e.g. a smart home, a manufacturing plant, a car, etc.), and the devices present in that platform along with the tasks they perform.
2. Creating semantic descriptions of each of the device classes involved in the platform using SDF.
3. Designing a script that emulates a set of IoT devices exchanging sensor data using the MQTT messaging protocol, ensuring that the data generated by each device is consistent with the task it performs.

By completing these steps, we will have a well-defined physical object or IoT platform that we can use as the basis to build the DT.

1.4.2 Digital twin design

Since we aim to build a DT that represents the IoT platform as a complex logical structure with reasoning capabilities, the most straightforward solution is to implement it as a KG. To do this, we need to take the following steps:

1. Compare the available open-source KGs and choose the one that best fits our purposes.
2. Familiarize ourselves with the usage of the KG, including how to define a schema or ontology, how to insert and update data through queries, etc.
3. Design an ontology, which includes domain-specific terminology defining objects and their relationships and attributes, allowing us to store and represent the model as a complex logical structure.
4. Define the ontology as the schema of the KG.
5. Insert the initial devices and their relationships into the KG, which will represent the initial situation of the DT.

By completing these steps, we will have a well-defined DT that is ready to be used for modeling and reasoning about the IoT platform.

1.4.3 Consistency handler implementation

The consistency handler will listen to the data flow generated by the emulated IoT platform and process the information to update the KG accordingly. To do this, the consistency handler should be capable of:

1. Acting as a client in the MQTT messaging protocol and receiving all messages exchanged.

2. Interpreting the class of a device (e.g. what device it is, what modules it includes, what attributes its modules measure) from its SDF description.
3. Defining the device class in the KG schema or ontology.
4. Inserting the device into the KG and updating its measured attributes in real-time as new messages arrive.
5. Determining how a new or unforeseen device should be integrated into the existing graph.

By implementing these capabilities, the consistency handler will be able to accurately and dynamically model the IoT platform as a KG.

1.5 Methodology

To accomplish the goals of this project, we have decided to develop an emulated test environment that generates messages from IoT devices in a simple automobile manufacturing plant, and to design and evaluate a DT that accurately reflects this test environment in real-time by processing the data it generates. To protect confidentiality, real data from an IoT platform was not used, and instead, a simulated platform was created.

To develop the simulated test environment, a Python script and various libraries including numpy, threading, paho mqtt, and json were used. The simulated environment consisted of a group of IoT devices that produced data representative of a real automobile manufacturing plant. The data generated by these devices included a header with information about the device's unique id and its class, as well as a body containing data from the sensors.

To emulate the exchange of messages between devices, an MQTT broker was set up on a docker container using Mosquitto. This virtual broker allowed the simulated devices to connect and exchange messages, which were then forwarded to the appropriate subscribers by the broker.

In the design of the DT, we considered various options and ultimately chose to use TypeDB, a KG deployed on a docker container, to manage the knowledge. This knowledge is managed through the use of TypeQL queries. To ensure that the KG accurately reflects the physical test environment in real-time, we implemented an algorithm called the consistency handler. This algorithm receives messages from the test environment, processes the information, and uses it to create queries for the KG.

For further details on the technology and tools that were selected for the DT, refer to Chapter 2. More information on the implementation and validation methods can be found in Chapter 3.

1.6 Delimitations

This thesis focuses on a prototype design of a consistency handler that builds a virtual model or DT of an IoT platform in real-time. Due to confidentiality issues, we lack data generated by a real IoT platform, which triggered the need to emulate a virtual IoT platform. Although it is important that the emulated IoT platform behavior is as similar as possible to a real platform, this is not the main objective of the thesis. Therefore, some limitations have been imposed on the simulation of the IoT platform, to simplify its design as much

as possible without losing key behaviors needed to design and test the consistency handler functionalities.

One of the main simplifications in this work is the lack of causality between the data generated by the IoT devices. In a real-world scenario, there would be a causal relationship between the data generated by different devices, such as data from a camera detecting an object driving the movements of a pickup robot. However, in our simulation, this relationship does not exist. This simplification allows for the focus on building the virtual model based on the individual behavior of the devices, rather than their causal relationships with other devices. However, this could impact the performance of the consistency handler if implemented in a real-world scenario and causality is important for the system goals.

Another simplification is in the type of data generated by the devices, which does not reflect their real behavior while performing a certain task. Instead, we have designed the devices to generate random data that is highly dependent on their assigned tasks. For example, a periodic task like sealing bottles would generate periodic data, while a device measuring environmental variables like temperature would generate data that varies slowly around a mean value. This simplification is sufficient for our goals because it allows the consistency handler to relate a task to a device based on the data generated.

Additionally, the thesis assumes that a semantic description in SDF is available for each of the device classes in the IoT platform. This description is based on the device specifications, and opens the possibility of designing an algorithm that can automatically generate an SDF description from a device specification file.

It is also worth noting that the KG is not constructed from scratch, but from an initial point. This means that we start with an initial set of logically connected devices, and the goal of the consistency handler is to evolve the graph in real-time to accurately represent the changes observed in the messages exchanged by the devices on the platform.

1.7 Structure of the thesis

The thesis is structured into six chapters that delve into the different aspects addressed in the research work. Chapter 2 presents an introduction to the background of the work, including a review of related literature. The methodology used in the research is thoroughly discussed in Chapter 3. In Chapter 4, we detail the implementation of the prototype developed as part of the study. The results and their analysis are presented in Chapter 5, and finally, the thesis concludes with a summary of the findings, and recommendations for future work, in Chapter 6.

Chapter 2

Background

This chapter presents the foundational concepts and relevant prior research that are related to this project. In particular, we describe in more detail the concepts introduced in Section 1.1. Finally, we discuss how our research connects to and builds upon previous studies in this area, and describe the key contributions and innovations of our approach.

2.1 Knowledge Representation

This section provides a brief introduction to the AI field of Knowledge Representation and Reasoning, based on [6].

2.1.1 What is knowledge?

One widely accepted definition of knowledge is "true justified belief". This means that for a subject S to know a proposition p, the following conditions must be satisfied: (1) S believes that p, (2) p is true, and (3) S is justified in believing that p. If any of these criteria are not met, the proposition cannot be considered as knowledge.

In this context, knowledge is understood as a set of propositions that shape a subject's view of the world. For example, if our subject is John, the set of propositions that describe his world would be given by sentences of the form "John knows that p". The entire set of these sentences constitutes the knowledge that describes the way in which John perceives the world, that is, his knowledge base.

2.1.2 What is a representation?

A representation is a relationship between two domains, where the first domain stands for the second. For example, a company's logo is a visual representation of the company itself. We are particularly interested in a specific type of representation called symbols, which are a set of characters from a predefined alphabet that stand for or represent certain concepts.

Knowledge representation, then, is the use of formal symbols to represent a collection of propositions believed by a particular subject or agent. It is important to note that a knowledge representation does not necessarily explicitly state all that is believed by the agent; it is the role of reasoning to fill in the gaps between what is represented and what is believed.

	Storage	Querying	Reasoning	Other characteristics
Neo4j Comm Ed (Property Graph)	Horizontal scaling	Cypher Pattern matching Navigation	Hierarchies Increased capabilities if exported to RDF	ACID Big community Python support GPL v3
TypeDB (Hypergraph)	Horizontal scaling Decentralized	TypeQL Pattern matching	Hierarchies Complex rules Query enhancement Validation	Limited ACID Python support GPL v3
Apache Jena (RDF)	Horizontal scaling	SPARQL Pattern matching Navigation	Hierarchies Complex rules Forward and backward chaining Validation	ACID Apache License v2

Table 2.1: Comparison of main knowledge graph alternatives.

2.1.3 What is reasoning?

Reasoning can be understood as the formal manipulation of symbols that form a collection of propositions in order to infer new propositions that are true according to our knowledge, but are not explicitly represented. For example, consider a knowledge base (KB) consisting of the following propositions: "John has a dog", "John's dog is named Tobi", and "Tobi likes eating bones". Through reasoning, we can infer that "John's dog likes eating bones" is a logical consequence of the KB. In other words, the conclusion is true based on our current knowledge, even though it is not explicitly stated in the KB.

2.2 Knowledge Graphs

To store knowledge and reason with it as well as to efficiently query specific insights, we have chosen to use a knowledge graph database. Knowledge graph databases are optimized for representing and managing complex, interlinked data that represents real-world entities and the relationships between them. This makes them well-suited for applications that require a high degree of understanding and reasoning about the data they contain, such as knowledge management systems and expert systems. Additionally, knowledge graph databases provide powerful querying capabilities that make it easy to explore and analyze data and extract specific insights, making them a valuable tool for data-driven decision-making.

2.2.1 Open-source knowledge graphs comparison

A thorough analysis of the available open-source options was conducted, in order to choose the most suitable KG for implementing a DT. The results of this analysis are summarized in Table 2.1, which compares several key factors of the three KGs that best fit our purposes: Neo4j Community Edition (a property graph database) [7], TypeDB (a hypergraph database) [4], and Apache Jena (an RDF database) [8].

The storage column indicates how each database handles scaling. All three databases can scale horizontally. TypeDB is also decentralized. The querying column lists the languages and features used to query each database. Neo4j uses Cypher, TypeDB uses

TypeQL, and Apache Jena uses SPARQL. All three databases have pattern-matching capabilities, and Neo4j and Apache Jena also have navigation capabilities.

The reasoning column in our analysis describes the ability of each database to handle hierarchies, rules, and validation, as well as their ability to chain together multiple pieces of information. The other characteristics column lists additional features of each database, such as support for ACID transactions, native Python support, and the type of license they use.

After considering all of these factors, we decided to use TypeDB for our KG because it had sufficient reasoning capabilities for our needs, even though it was not as strong as Apache Jena in this area. However, the fact that TypeDB has native support for Python, which is the language we are using for our code development, was a key factor in our decision.

2.2.2 TypeDB

TypeDB, as stated in its documentation [9], is a KG that allows users to define a schema tailored to their specific domain. This schema serves as the backbone of the KG, and is composed of three main types: entities, relationships, and attributes. Entities represent domain objects, while relationships are n-ary connections based on roles. Attributes are values that are associated with entities or relationships. The direction of relationships is defined by the roles assigned to each side.

TypeDB can be queried using its query language, TypeQL. For example, to create a simple KG for phone calls, we could define the schema using the TypeQL query in Listing 2.1.

Listing 2.1: Definition of the schema for a phone call knowledge graph, including attributes, relations, and entities.

```
define

# ATTRIBUTES
name sub attribute, value string;
city sub attribute, value string;
phone-number sub attribute, value string;

duration sub attribute, value double;
age sub attribute, value double;

started-at sub attribute, value datetime;

is-customer sub attribute, value boolean;

# RELATIONS
contract sub relation, relates provider, relates customer;

call sub relation, relates caller, relates callee,
owns started-at, owns duration;

# ENTITIES
company sub entity,
owns name,
plays contract:provider;

```

```
person sub entity,
owns name, owns phone-number, owns city, owns age, owns is-
    customer,
plays contract:customer, plays call:caller, plays call:callee;
```

This query defines a set of attributes, relations, and entities for a database. The attributes include various types of information such as names, phone numbers, and ages, and they are all assigned data types like string and double.

There are two relations defined in the query: contract and call. The contract relation involves entities called provider and customer, while the call relation involves caller and callee entities and also includes information about the start time and duration of the call.

Finally, the query defines two entities: company and person. The company entity has a name attribute and is involved in the provider role of the contract relation, while the person entity has various attributes such as name, phone number, and age, and is involved in the customer and caller/callee roles of the contract and call relations, respectively. In Figure 2.1 we illustrate the resulting schema after running this query.

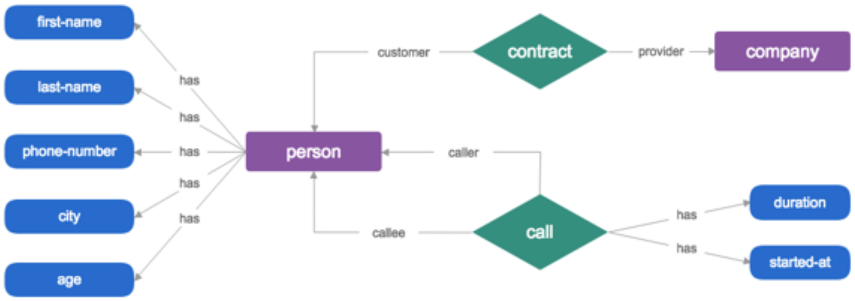


Figure 2.1: Schema of the example knowledge graph for phone calls.

Once the schema of the KG is defined, we can add specific data that adheres to the defined schema. The TypeQL query in Listing 2.2 inserts two people, a company, and two calls into the KG.

Listing 2.2: Data insertion for a phone call knowledge graph, defining companies, persons, their attributes, relations between them and call instances with duration.

```
insert

# COMPANY AND RELATIONS
$comp1 isa company, has name "Ericsson";
$cnt1 (provider: $comp1, customer: $p1, customer: $p2) isa
    contract;

# PERSONS AND THEIR ATTRIBUTES
$p1 isa person, has first-name "Ale",
has last-name "Jarabo", has phone-number "398559423";
$p2 isa person, has first-name "John",
has last-name "Legend", has phone-number "107530750";

# CALLS RELATING PERSONS
$call1 (caller: $p1, callee: $p2) isa call, has duration 37;
$call2 (caller: $p2, callee: $p1) isa call, has duration 302;
```

We can appreciate the KG with the data populated with this query in Figure 2.2.

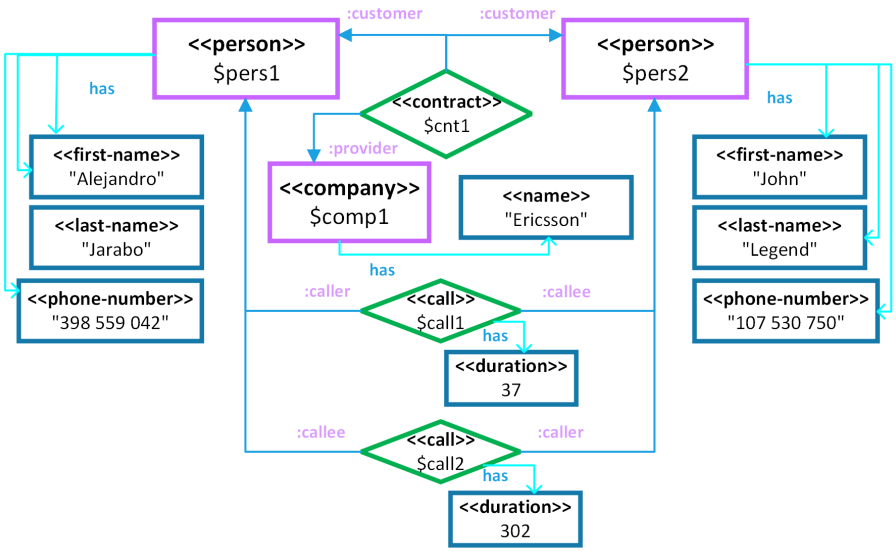


Figure 2.2: Populated data in the example knowledge graph for phone calls.

Through this example, we have gained an understanding of how TypeDB operates in general, including how to define a schema using domain-specific knowledge and how to populate the resulting KG with data that conforms to that schema. This is useful for understanding how we can use the KG to store and manage information about IoT devices.

2.3 Digital Twin

A DT is a virtual replica of a physical object or system that is designed to accurately reflect its real-world counterpart. The physical object, such as a wind turbine, is equipped with sensors that collect data about its performance, such as energy output and temperature. This data is then processed and applied to the DT, which can be used to run simulations and study performance issues. The goal is to generate valuable insights that can be applied back to the physical object to improve its performance. Figure 2.3 represents the data flow between the DT and its real-world counterpart.

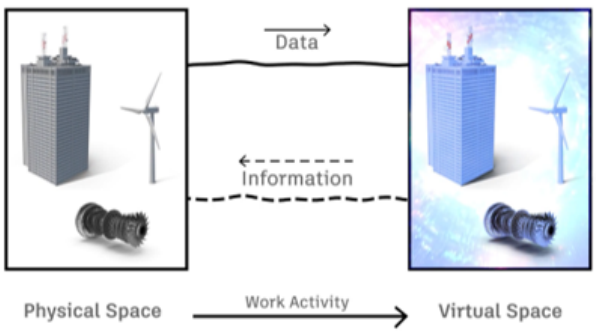


Figure 2.3: Interaction between the digital twin and its physical counterpart.

DTs are distinct from emulations because they are automatically constructed using real-time data and have a bi-directional flow of information, allowing them to share insights with the physical objects they represent. This allows DTs to be more dynamic

and responsive than emulations, which are typically based on pre-determined models and data.

2.4 Internet of Things

IoT refers to the network of physical objects, such as devices, vehicles, and buildings, that are embedded with electronics, software, sensors, and connectivity to enable them to collect, exchange, and act on data. This allows these objects to connect and communicate with each other and with external systems over the internet, enabling them to be monitored, controlled, and accessed remotely.

IoT enables a wide range of applications and services, such as smart homes, connected cars, and industrial automation. It has the potential to transform many industries and enable new business models, as well as improve efficiency, safety, and sustainability.

2.4.1 IoT device platforms

An IoT device platform is a set of technologies and services that enable the development, deployment, and management of IoT devices and applications. It typically includes hardware and software components, such as sensors, gateways, communication protocols, and cloud services, that provide the infrastructure and capabilities for IoT devices to connect, communicate, and exchange data with other devices and systems.

2.4.2 MQTT

We have chosen the MQTT protocol for exchanging messages between IoT devices, since it is a popular choice for IoT communication due to its simplicity, efficiency, and robustness in challenging network environments.

MQTT is a lightweight, publish-subscribe messaging protocol that is commonly used for communication in IoT systems. It is designed to be efficient, low-power, and easy to implement on small devices with limited resources.

In MQTT, communication occurs between a client and a server, known as a broker. Clients can publish messages to a topic on the broker, and other clients that have subscribed to that topic can receive the messages. The broker is responsible for forwarding the messages to the appropriate clients.

2.5 Semantic Definition Format

SDF [10] is a standard for defining the structure and meaning of data for a specific domain. It is used to formally specify the data model, constraints, and relationships, allowing the data to be interpreted and processed by software in a consistent and standardized way.

In this project, we have used SDF to semantically describe classes of IoT devices. The SDF descriptions capture the structure and meaning of the device classes, including the modules they contain and the attributes they measure. An example of a simple SDF description of a *Noise Sensor* IoT device class is shown in Listing 2.3.

Listing 2.3: Semantic definition of a *Noise Sensor* device class in SDF format, including namespace, version, copyright, license, description and properties of the device, with its noise level in decibels.

```
{
"namespace": {"eri": "https://ericsson.com/models"},
"defaultNamespace": "eri",
"info": {
    "title": "Noise Sensor",
    "version": "2022-12-12",
    "copyright": "Copyright 2022 Ericsson.",
    "license": "BSD-3-Clause"
},
"sdfThing": {
    "NoiseSensor": {"description": "Monitors environmental
        noise intensity",
      "sdfObject": {
          "noise_sensor": {
              "description": "Measures environmental noise
                  .",
              "sdfProperty": {
                  "noise": {"description": "Noise value",
                      "type": "number",
                      "unit": "dB"
}}}}}}
```

This semantic definition describes an IoT device class called *Noise Sensor*. It is a part of a namespace called "eri". A namespace is a container for a set of identifiers that helps to prevent name clashes between different elements in a system. By specifying a namespace, it is possible to use the same name for different elements in different parts of the system without causing conflicts. The definition also includes metadata such as a title, version, copyright, and license.

The *Noise Sensor* thing or class is described as a device that monitors environmental noise intensity, and it has a single object or module called *noise_sensor* that measures environmental noise. The *noise_sensor* module has a single property or attribute called *noise* which is a numerical value in decibels (dB).

The goal of this SDF class description is to enable the automatic definition of new device classes within the schema of the KG when a new device is detected. This is achieved by generating a define TypeQL query based on the SDF description. This allows us to accurately and efficiently capture the structure and semantics of new device classes, ensuring the consistency and interoperability of the data in the KG.

2.6 Related work

In the following section, we will review previous research relevant to our work and discuss how it informs and influences our prototype implementation.

2.6.1 Test environment

The primary objective of this thesis was to design a test environment that simulates a group of IoT devices performing various tasks and exchanging data on a network. We based the

design of this test environment on [11], which outlines a generic domain-specific ontology for a simple automobile manufacturing line. The demonstration automobile production line served as a reference for the devices and tasks included in our test environment. Additionally, the general ontology outlined influenced the ontology or schema we defined for our KG implementation of the DT.

2.6.2 Knowledge Representation

Semantic modeling is a way to ensure interoperability between various systems and applications in the Internet of Things (IoT). In [12], a comprehensive and lightweight semantic description model was developed for representing knowledge in the IoT. This type of modeling enables the effective access and use of information generated by IoT devices, which highlights the importance of representing the knowledge contained within the IoT platform being modeled.

In order to take advantage of and represent the knowledge contained in the information shared by IoT devices in our work, we have defined each device class semantically using SDF. This enables us to better understand and make use of the information provided by the devices.

2.6.3 Digital Twin

As the authors of [13] claim, the concept of a DT has garnered increasing interest in recent years. In order to establish solid foundations for future research and address current gaps, it is essential to consolidate the existing research on DTs. This will enable future research to focus on critical and relevant issues, such as the problem addressed in this thesis, which is the need for automated responses to unexpected changes in the data processed by the DT.

Furthermore, [14] provides a summary of the various types and definitions of DTs, as well as an analysis of the value they can bring to certain industries, particularly the manufacturing sector with the emergence of Industry 4.0. This motivated us to investigate the implementation of a DT using IoT devices in a production line.

It is also worth mentioning [15], in which a DT of an industrial production line is designed with the aim of optimizing existing production resources in the automotive industry through augmented production and planning strategies. This work motivated and inspired us to create a test environment based on an automobile production line, and illustrates the potential benefits and applications of such DTs.

2.6.4 Knowledge Graphs

Graph databases raised in popularity recently for efficiently storing and managing data with complex relationships, as outlined in [16]. This motivated us to use a graph database to store and manage the data of our DT.

A KG is a specialized type of graph database that represents relationships between entities based on a predefined domain-specific ontology. In [17] the authors provide an overview of the current state of KGs and discuss emerging topics such as graph completion. Graph completion is relevant for us since it is closely related to the integration of unanticipated devices into the KG, which involves adding new nodes and edges.

There have been several studies on representing the Internet of Things (IoT) through KGs, such as the "Graph of Things" project [18]. This project uses a KG to efficiently explore and query the data collected by a massive number of IoT devices in real-time. However, this KG is not created automatically, which motivated us to address the need for an automated method to evolve the KG in real-time, adapting to changes that may occur.

2.6.5 Text and Semantic Similarity

When a new, unexpected device appears and begins exchanging messages on the network, it is useful to examine its device class to see if it is similar to any of the device classes already integrated into the KG. This is because we expect similar types of devices to be connected in similar ways to other devices on the network. By analyzing the device class, we can better understand how this new device fits into the overall network and how it may be connected to other devices.

This motivates the need of measuring the similarity between device class definitions, which in our case are textual semantic descriptions following the SDF format. Estimating semantic similarity between text data is a challenging problem in the field of Natural Language Processing (NLP). In [19], the authors evaluate the various methods that have been developed to measure similarity between texts and discuss their strengths and weaknesses. These methods could potentially measure the similarity between devices in our approach.

However, to measure the similarity between devices, we have utilized the normalized Levenshtein distance metric, as outlined in [20]. The Levenshtein distance, also known as edit distance, is a measure of the similarity between two strings, determined by counting the minimum number of single-character edits needed to transform one string into the other. Despite not being the central focus of our thesis, we found this metric to be sufficient for our needs.

2.6.6 Time Series Similarity

It is crucial that, when a new device is introduced, we are able to collect and store the data it shares in a buffer. This time series data is essential for understanding the device's behavior and for determining how it is used in specific tasks and locations. By analyzing this data, we can gain a deeper understanding of the device and how it fits into the overall system.

The data reported by instances of a device class helps us distinguish between devices belonging to the same class. Therefore, it is necessary to measure the behavioral or time series similarity between individual device instances. In [21], the authors compare various methods for dealing with time series data, including techniques for dimensionality reduction (e.g., the Discrete Fourier Transform) and methods for measuring similarity (e.g., the Euclidean distance).

This thesis uses the MASS (Mueen's Algorithm for Similarity Search) algorithm, developed in [22], for measuring time series similarity. MASS is an algorithm for fast and efficient identification of similar time series sub-sequences with both Euclidean distance and Correlation Coefficient. It utilizes the Early Abandoning technique for increased performance and is independent of the data set and query. With a time complexity of $\mathcal{O}(n \log n)$, this algorithm allows for an efficient search of time series sub-sequences

within other time series, optimized for both complexity and speed.

2.6.7 Machine learning on Knowledge Graphs

Knowledge reasoning allows for the deduction of new, previously unstated facts based on a set of rules. This can be thought of as discovering new connections, or edges, in a KG based on existing edges. Similarly, we can use machine learning models powered by statistics to predict new facts about the world our graph models represent.

In [23], the authors present two main machine learning models for predicting new edges in KGs using statistical methods. The first model is based on latent feature models such as tensor factorization and multi-way neural networks which aim to learn hidden representations of entities and relationships in a KG to predict new facts. The second model is based on mining observable patterns in the graph to discover common patterns in the graph data and use them to predict new edges. These models and methods may be useful in integrating new devices into our model that are not similar enough to any devices already included. They can aid in identifying relationships that these new devices should have with existing devices and objects in our KG.

Chapter 3

Methodology

In this chapter, we describe the methodology used in this thesis for developing a simulated test environment based on an automobile manufacturing plant. The goal of this environment is to serve as a platform for evaluating a DT based on a KG, which is a representation of the manufacturing plant in real-time. The DT is designed to process data generated by the plant in order to accurately represent it.

To deploy the prototype DT, we have used a series of methods that are detailed in this section. These methods included setting up the simulated test environment, configuring the DT to interact with the simulated plant, and testing the functioning of the DT to ensure that it is accurately representing the plant in real-time.

The code was implemented and run on an Ubuntu server accessed via SSH. The server has a 64-bit, multi-core Intel Xeon E312xx processor with a base frequency of 2GHz and 6 cores, as well as 32GB of RAM. The operating system is Ubuntu 22.04.1 LTS. We used Python 3.10 with the following libraries and their respective versions: numpy 1.21.6 [24], paho-mqtt 1.6.1 [25], stumpy 1.11.1 [26], thefuzz 0.19.0 [27], and typedb-client 2.9.0 [28]. Furthermore, we employed the TypeDB KG docker image version vaticle/typedb:2.11.1, and the Mosquitto MQTT broker [29] docker image version eclipse-mosquitto:2.0.15 in our project. Additionally, we used TypeDB Studio [30] to visualize test and assess the evolution of the graph in real-time, by sending queries to the database and visualizing the response data as a graph.

3.1 Emulated Test Environment

3.1.1 Description

The simulated test environment was developed using a Python script and various libraries including numpy, threading, paho mqtt, and json. It consisted of a group of simulated IoT devices that produced data representative of a real automobile manufacturing plant. The data generated by these simulated IoT devices included a header with information about the device's unique id and its class, as well as a body containing data from the sensors.

To emulate the exchange of messages between devices, an MQTT broker was set up on a docker container using Mosquitto. This virtual broker allows the simulated devices to connect and exchange messages, which are then forwarded to the appropriate subscribers by the broker.

3.1.2 Setup and Configuration

The test environment was set up and configured using the following steps:

1. Deployment of MQTT broker using Mosquitto in a docker container. The broker will act as a server forwarding messages between publishers and their subscribers and will listen on port 8883.
2. Creation of a Python script defining an IoT device class that serves as a thread and implements an MQTT client to connect and send/receive messages from the broker.
3. Design of the semantic definitions of a set of IoT device classes (*Conveyor Belt*, *Air Quality*, etc.) using SDF.
4. Development of Python IoT device classes that match the defined SDF classes, and adapt the data generation of each device to its definition (the attributes it measures). These classes will inherit the previously mentioned *IoT Device* class to act as threads and communicate with the broker.
5. Development of a script that instantiates the defined device class threads and emulates the data generation of the device platform.
6. Development of test cases within the emulation script to test the DT's capabilities, such as the definition of new classes, updating device attributes, and integrating new devices within the graph.

3.2 Digital Twin Knowledge Graph

3.2.1 Description

The TypeDB KG is deployed within a docker container that listens for queries on port 1729. These queries allow for modification and management of the knowledge contained in the graph. The consistency handler, a Python script, is responsible for receiving messages from devices that are forwarded by the broker. It processes this information and updates the KG by automatically generating queries to do so.

The consistency handler is a Python script that uses libraries such as typedb client, paho mqtt, numpy, stumpy, and thefuzz. It includes a TypeDB client class that handles the initialization of the schema and data of the graph, as well as the generation and sending of queries to update the graph. This class is inherited by a KG agent class that implements an MQTT client to communicate with the broker. The KG agent class processes incoming device messages and determines the necessary actions and queries to update the graph accordingly.

3.2.2 Setup and Configuration

The KG-based DT and the consistency handler are set up and configured using the following steps:

1. Deploy the TypeDB KG in a docker container. The broker will act as a server storing and updating the graph knowledge, and will listen on port 1729.
2. Create a Python TypeDB client class that updates the graph through queries.
3. Develop a Python KG agent class, implementing an MQTT client that receives messages from devices forwarded by the broker. This class will inherit the TypeDB client class to handle the KG.

4. Develop the consistency handler algorithm, which will process incoming messages and determine the appropriate queries to generate in order to update the graph.
5. Design the process for integrating new devices into the graph. This process begins by using the Levenshtein distance (with the thefuzz library) to calculate the closest classes to the new device class. Then, time series pattern matching using the MASS algorithm (with the stumpy library) is used to compute the closest device instance. The results of this similarity analysis are used to determine how the new device should be integrated into the graph.

3.2.3 Data Validation

To ensure the validity of the data within the KG, we checked that its instances, relations, and attribute values always matched the present state of the emulated test environment as we updated the graph.

3.3 Evaluation

The performance and effectiveness of the KG-based DT were evaluated through a series of test cases.

First, we examined the names and relationships of the modules and attributes to ensure that the device classes in the schema were defined correctly. This involved checking that each module had a descriptive and accurate name, and that the attributes were linked to the correct modules. We also verified that the relationships between the modules and attributes were established correctly, ensuring that the schema accurately reflected the structure and properties of the devices being modeled.

Next, we compared the values in the last messages received with the values in the KG to confirm that the data in the KG remained accurate and up-to-date.

Finally, we tested the integration of new devices by introducing a new device class called *Air Quality Simplified* which was similar to the existing *Air Quality* class but with some simplifications. We introduced a device of this class into the message stream after an *Air Quality* device measuring indoor variables had disappeared. This allowed us to verify that the similarity metrics correctly identified the *Air Quality* class as the closest match to the new device class, and that the closest device was the inactive *Air Quality* device measuring indoor variables. This demonstrated that our integration algorithm was functioning correctly and able to detect the new device as a replacement for the closest device, replicating the relations of the closest device on the new device and removing the inactive closest device from the graph.

Overall, these tests showed that the KG was performing effectively and accurately modeling the devices being tracked.

3.3.1 Limitations and Challenges

During the evaluation process, we identified a limitation in the script that processes the messages from the devices and turns them into queries for the KG. This script was optimized to be as fast as possible, but the query management by the TypeDB client in Python was found to be slow (around 100 ms), which limits the speed of the processing.

This limitation could potentially be addressed by using a faster machine with more computing power and/or optimizing the TypeDB Python client.

Another limitation is the lack of real-world data to ensure the validity of the prototype, as it was only tested in a simplified test environment. Testing the KG-based DT in real-world environments, such as manufacturing, healthcare, and transportation, would provide a more accurate picture of how well it performs in actual scenarios and how it can be implemented in practice, especially when dealing with edge cases.

The main challenge identified in this work is the integration of new unanticipated devices when they do not show similarities to the existing devices in the KG. This issue could be tackled by using machine learning for link discovery, in order to deduce how to connect these new devices to the other devices in the graph. This process might require the creation of a new branch or task in the graph, to properly organize these new devices and allow them to function correctly within the system.

Chapter 4

Prototype implementation

In this chapter, we will present the prototype implementation of our KG-based DT for IoT platforms. We will begin by providing an overview of the entire prototype, including its architecture and design. Then, we will delve into the main elements of the prototype in more detail.

First, we will focus on the test environment set up to emulate IoT devices, and how it runs and behaves. Next, we will focus on the KG, the core component of our prototype, and how it stores and represents the state of the physical system in real-time. Finally, we will discuss the consistency handler algorithm, which is in charge of maintaining consistency between the physical and virtual systems by processing the messages received from the IoT devices.

4.1 Overview

The prototype implementation is divided into three main components: the test environment based on a simple automobile manufacturing plant, the DT design based on a KG, and the consistency handler algorithm that dynamically updates the DT's knowledge based on the data generated by the IoT devices in the test environment.

Figure 4.1 illustrates the architecture of the solution, with the specific functionalities and interaction of these three components. First, we have a script emulating the physical object, in this case, IoT devices in a simple automobile manufacturing plant. These devices use the MQTT protocol to publish messages to the broker on specific topics.

The messages are divided into a header and a body. The header includes a Universally Unique Identifier (UUID) for the device that generated the message, the class of the device, and the time the message was generated. The body contains the actual data generated by the device, such as a temperature or position reading.

The broker is implemented using Mosquitto on a docker container and its main function is to manage the MQTT network. It keeps track of which devices are connected, what topics they publish to, and what topics they are subscribed to. The broker then forwards the messages to the appropriate subscribers. The consistency handler receives these messages through an MQTT client that is subscribed to all topics.

The consistency handler processes the information in the messages by checking whether the device class and specific instance are already integrated into the graph. If they are not, it defines the device class within the schema of the KG and determines the best position within the graph for the new device. If the device class and instance are

already present in the graph, it simply updates the values of the parameters that the device measures within the graph.

All these operations are carried out using database queries, which are generated based on the information contained in the message and sent via a client to the TypeDB KG running in a docker container. The container processes these queries and updates the graph accordingly.

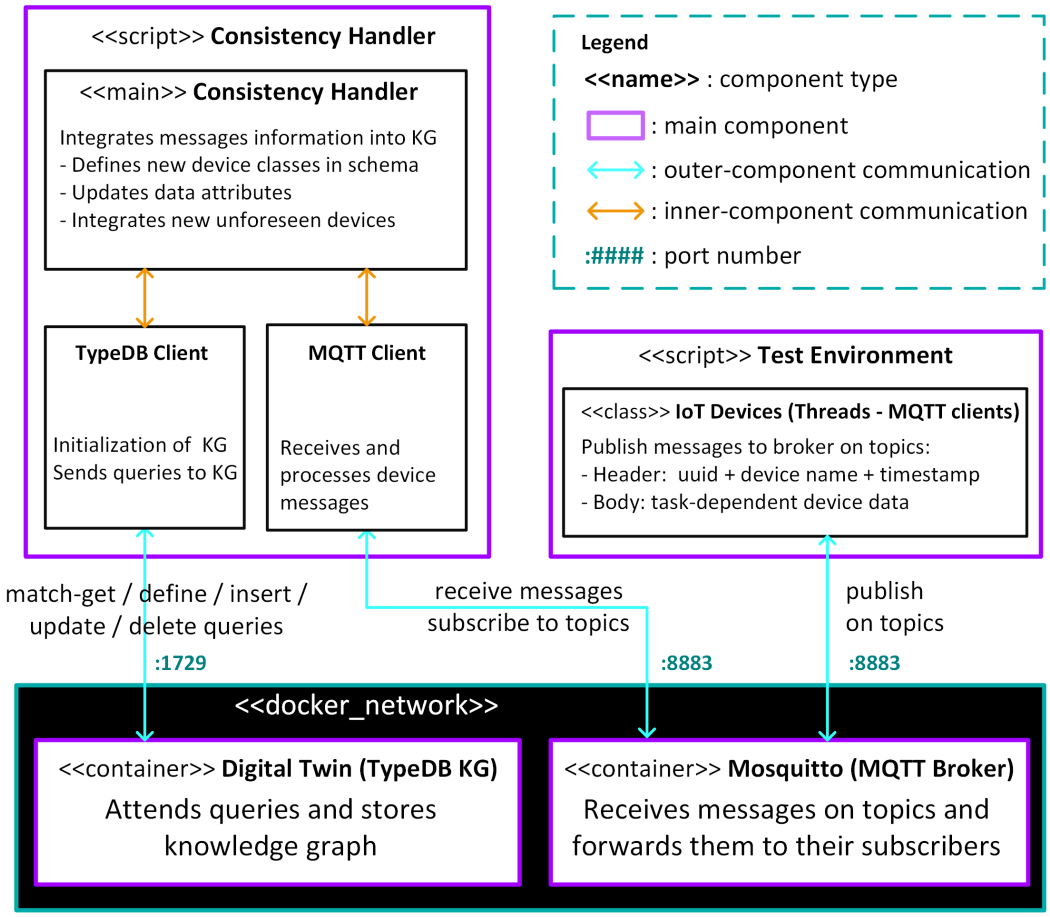


Figure 4.1: Architecture diagram with main components and their relation.

4.2 Test environment

The test environment is inspired by a simple automobile manufacturing plant, and will have an influence on the design of both the KG-based DT and the consistency handler.

This simple manufacturing plant is divided into two different departments, the production department, managing the actual manufacturing of the cars, and the safety department, which keeps track of ambient variables such as the temperature and triggers alarms if necessary.

The plant is divided into departments, each responsible for a set of tasks or functionalities performed by IoT devices. For example, the safety department includes an "indoor monitoring" task that uses IoT devices to monitor indoor ambient variables such as temperature, pressure, and noise.

The *Air Quality* device is one example of an IoT device that performs tasks within the plant. It has four modules for measuring temperature, humidity, pressure, and air quality

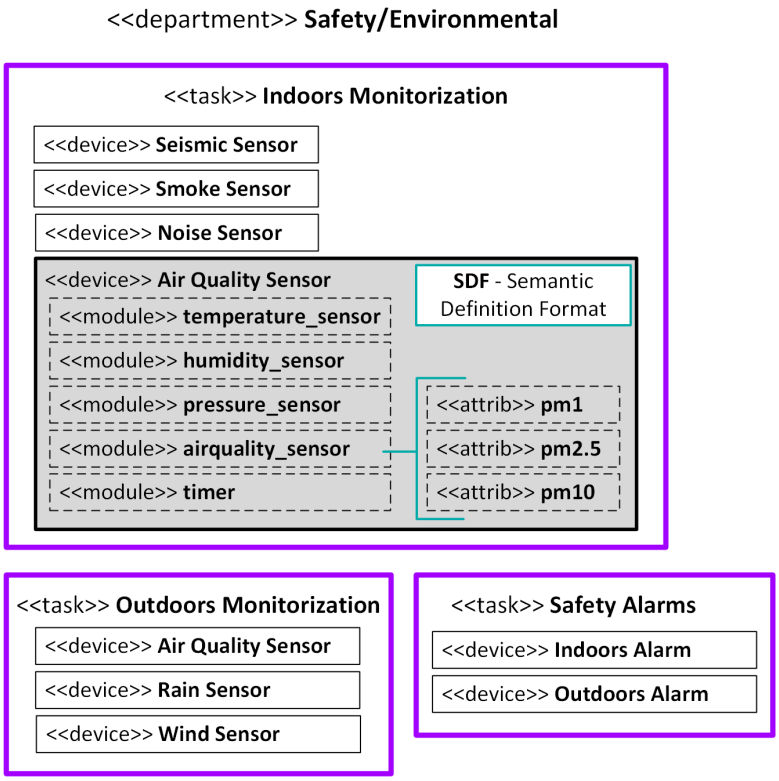


Figure 4.2: Overview of the tasks and devices in the safety department, including the *Air Quality* device class with its modules and attributes as defined in its SDF description.

(via inhalable particulate matters such as PM1 or PM10). Figures 4.2 and 4.3 show the complete set of tasks and devices in our test environment, with the former depicting the safety department and the latter depicting the production department.

4.2.1 Semantic definition of device classes

The SDF format is used to semantically interpret the data generated by the devices in the KG. Each device class is represented as a *sdfThing* and includes a UUID field for unique instance identification.

The device classes in SDF specify the set of modules that a device includes using the *sdfObject* type. These modules contain attributes or values that are measured, which are of the *sdfProperty* type. Each element in SDF has a name and a textual description, and properties also include information such as the type of value being measured and its unit (e.g., meters, volts, etc.).

An example of the semantic definition of the *Air Quality* device class using the SDF format is provided in Listing 4.1. The *uuid* property allows for the instantiation and identification of devices belonging to this class. The definition of modules, such as the *temperature_ sensor* and the *air_ quality _ sensor*, includes the attributes they measure, such as *temperature* in $^{\circ}\mathrm{C}$ and PM10 in $\mu\mathrm{g\,m^{-3}}$.

Listing 4.1: Semantic definition of *Air Quality* device class in SDF format, including properties for temperature, humidity, pressure, and pollutants measurement (PM1, PM2.5, PM10).

```
{
"namespace": {"eri": "https://ericsson.com/models"},
"defaultNamespace": "eri",
"info": {
    "title": "Air Quality",
    "version": "2022-12-12",
    "copyright": "Copyright 2022 Ericsson",
    "license": "BSD-3-Clause"
},
"sdfThing": {"AirQuality": {"description": "Monitors air quality.",
"sdfProperty": {"uuid": {"sdfRef": "sdfData/sdfProperty/uuid"}},
"sdfObject": {
"temperature_sensor": {
    "description": "Measures environmental temperature.",
    "sdfProperty": {
        "temperature": {"description": "Temperature value",
            "type": "number",
            "unit": "Cel"
}}},
"humidity_sensor": {
    "description": "Measures environmental humidity.",
    "sdfProperty": {
        "humidity": {"description": "Humidity value",
            "type": "number",
            "unit": "%"
}}},
"pressure_sensor": {
    "description": "Measures environmental pressure.",
    "sdfProperty": {
        "pressure": {"description": "Pressure value",
            "type": "number",
            "unit": "bar"
}}},
"air_quality_sensor": {
    "description": "Measures pollutants in the air.",
    "sdfProperty": {
        "pm1": {"description": "PM1 (viruses, exhaust gases)",
            "type": "number",
            "unit": "ug/m3"
        },
        "pm25": {"description": "PM2.5 (bacteria, spores, pollen, toner dust)",
            "type": "number",
            "unit": "ug/m3"
        },
        "pm10": {"description": "PM10 (pollen, desert dust)",
            "type": "number",
            "unit": "ug/m3"
}}}}}}}
```

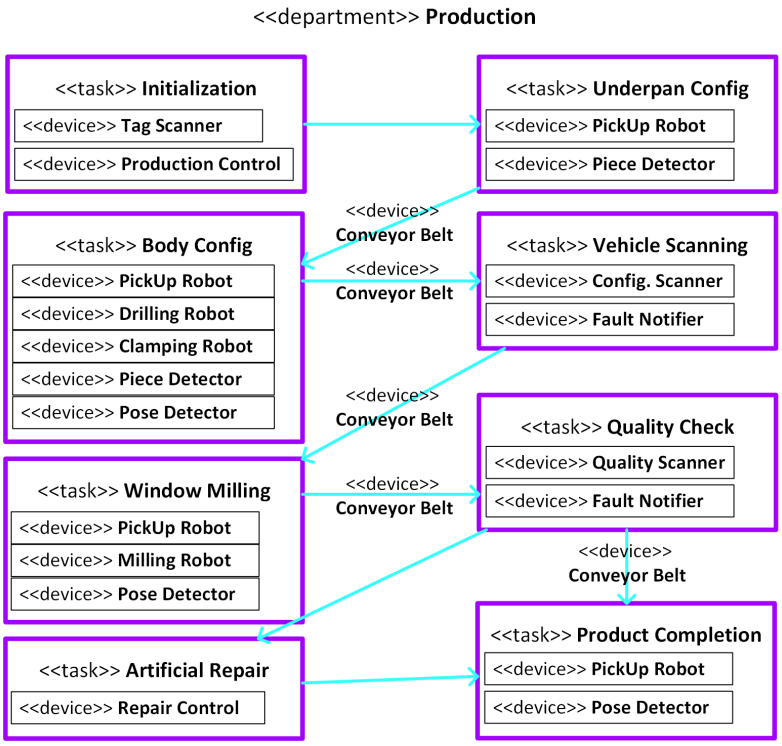


Figure 4.3: Overview of the tasks and devices in the production department. The blue arrows represent the order of the tasks, indicating if there is a conveyor belt connecting them.

4.2.2 Emulation of IoT devices

We have created a Python script to simulate the generation of data messages from IoT devices. The script defines a *IoT Device* class that inherits from the *Thread* class and acts as an MQTT client that connects to the broker. This class has attributes such as the interval between generated messages and the UUID of the device.

We can define specific device classes that inherit from the *IoT Device* class in order to customize the data generation for their respective device class. The purpose of these classes is to implement a data generation method, which is called periodically from the parent class to populate the body of the messages that are transmitted over the MQTT network. This allows us to tailor the data generation process to fit the specific needs of each device class.

The data generated by these devices is also tailored to the tasks they perform. This is accomplished in different ways for ambient variables (such as temperature and humidity) and variables indicating the behavior of devices (such as robot arms). For ambient variables, we have defined ground truth time series that indicate the value of these variables in both indoor and outdoor environments. The specific devices then sample from these ground truth variables with some added random noise, allowing us to obtain similar temperature readings for devices placed in similar locations.

For behavior variables, we have defined a set of parameters for each task to customize the data generated. For instance, in the case of robotic arms, the position data they generate will have a periodic behavior with a period, phase delay, and amplitude that are tailored to the task they are performing. To simplify the data generation process, we have assumed

that devices of the same class that are carrying out the same task will display similar periodic behaviors and peak frequencies. This is reasonable to assume if we consider, for example, two robots sealing plastic bottles in a factory. Even if their movements are not synchronized, their periodic repetitive behavior and frequencies will likely be similar.

Listing 4.2: JSON message generated by the *Air Quality* device, including sensor readings of temperature, humidity, pressure, and pollutants levels in micrograms per cubic meter (PM1, PM2.5, PM10) along with the device identifier and timestamp.

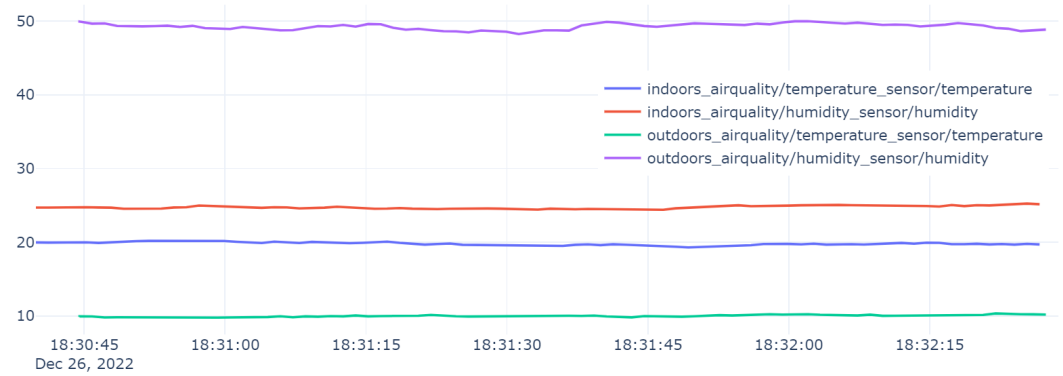
```
{
'category': 'DATA',
'class': 'AirQuality',
'topic': 'safetyenvironmental',
'uuid': 'indoors_airquality',
'timestamp': '2022-12-26T11:32:55.859200',

'data': {
    'temperature_sensor': {
        'temperature': 19.88
    },
    'humidity_sensor': {
        'humidity': 25.18
    },
    'pressure_sensor': {
        'pressure': 0.65
    },
    'air_quality_sensor': {
        'pm1': 0.46,
        'pm25': 8.70,
        'pm10': 17.48
}}}
```

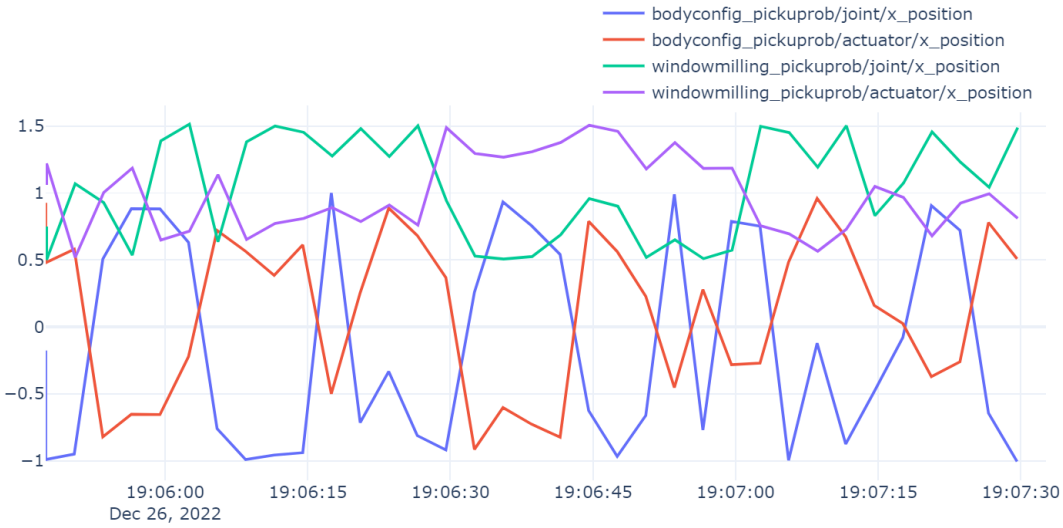
The devices generate messages at regular intervals, referred to as their period. For simplicity, we have assumed that all devices have a period between 2 and 5 seconds. These messages consist of a header and a body. The header includes information about the message category (either CONNECTED, DISCONNECTED, or DATA), the device class, the topic the message is published to, and the time the message was generated. The body, or data, contains the actual data generated by the device. An example of a message generated by a device of the class *Air Quality* is shown in Listing 4.2.

By storing the messages generated by each device in a buffer, we can monitor the time series evolution of their attributes. Figure 4.4a illustrates the time series data for two *Air Quality* devices, one placed indoors and the other placed outdoors. We can see that the temperature and humidity values fluctuate around different levels depending on the location of the devices. In addition, Figure 4.4b displays the time series data for two pick up robot devices performing the body configuration and window milling tasks. We can observe how the behavior of the same attribute is customized for each task, even though the devices belong to the same class.

The device emulation feature also allows us to simulate the disconnection and connection of devices at specific times. This functionality enables us to test the correct operation of the consistency handler when integrating new and unexpected devices.



(a) Temperature and humidity attributes for two *Air Quality* devices, one located indoors and the other located outdoors.



(b) Position of the joint and actuator for two *Pick-Up Robot* devices, one located in the body configuration task and the other located in the window milling task.

Figure 4.4: Emulated time series.

4.3 Digital Twin Knowledge Graph

To build the DT as a KG, we use TypeDB. This process involves defining an ontology or schema that is customized to the specific domain in which we are creating the graph. This schema outlines the types of objects that will be included in the graph and how they should be related to one another, as well as any attributes that these objects may have. This helps ensure that the DT accurately reflects the real-world system it represents and allows us to effectively organize and analyze the data within it. After the schema has been defined, we must populate the graph with initial data, which will serve as the starting condition for the graph. The consistency handler will then be responsible for managing the evolution of the graph over time, including the addition or removal of devices as needed.

4.3.1 Knowledge graph initial schema definition

A schema for TypeDB, depicted in Figure 4.5, was created based on a simple ontology for the problem. The schema includes four entity types: *department*, *task*, *device*, and

module. This schema is defined in the KG with a TypeQL query as shown in Listing A.1 in the appendix. The departments and tasks present were shown in Figures 4.2 and 4.3. The *device* and *module* entities are abstract and are not used directly, but rather serve as a foundation for sub-types of these entities. For example, a sub-type of *device* could be *Air Quality* or *Pick-Up Robot*, while a sub-type of *module* could be *temperature_sensor* or *joint*. These sub-types represent the actual device classes and their modules.

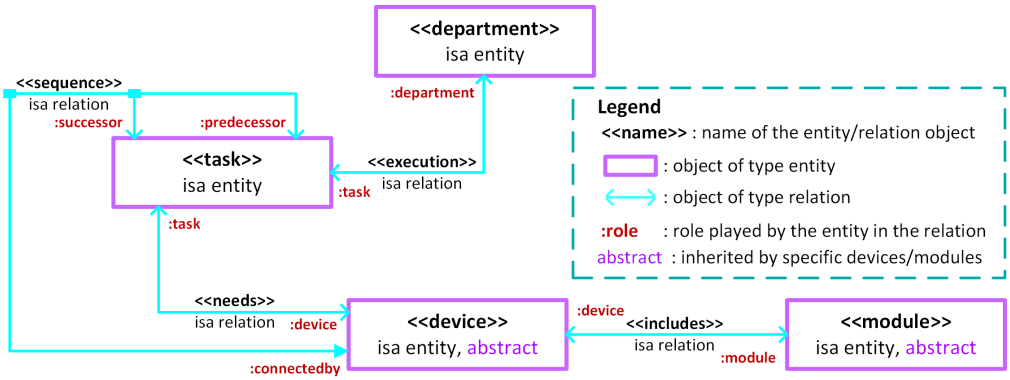


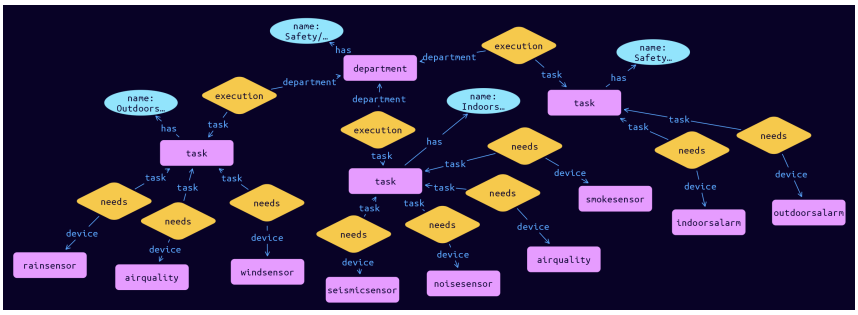
Figure 4.5: Schema designed for TypeDB knowledge graph to represent the IoT test environment.

In addition to the entity types, the schema includes four relationships: *execution*, *sequence*, *needs*, and *includes*. The *execution* relationship connects tasks to a specific department, the *needs* relationship connects tasks to the devices that perform them, and the *includes* relationship connects a device with the modules it implements, as described by its SDF definition. The *sequence* relationship signifies the order between tasks, if one exists, and also specifies the device used to connect them (often a conveyor belt). These relationships impose restrictions on the types of objects they connect through the definition of roles. For example, in the *includes* relationship, the *device* type plays the role *device* and the *module* type plays the role *module*, which means that only these types are allowed to participate in the *includes* relationship. No other types are permitted to have a relationship of type *includes*.

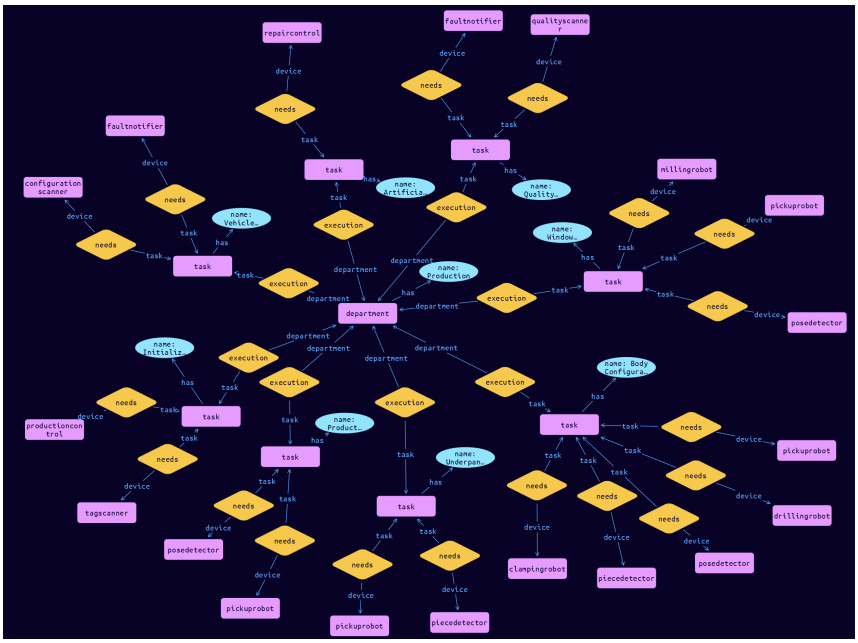
The schema also defines attributes that represent values that can be associated with entity or relation types. The attributes defined in the initial schema are *name*, a string type used to give a name to departments and tasks; *uuid*, a string type used to uniquely identify instances of device classes within the graph; and *timestamp*, a date type that indicates the moment when the device data was last updated.

4.3.2 Knowledge graph initial data population

Additionally, we must populate the graph with the initial data, representing the initial set of departments, their tasks, and their devices, as well as how they relate to each other according to the schema defined. This is done with a TypeQL query as shown in Listing A.2. Using TypeDBStudio, we can visualize the graph data initially populated, as shown in Figure 4.6 for the safety and production departments. The graphs shown here will evolve to include the data of the devices in real-time, as well as to include new devices or remove inactive devices, with such evolution being managed by the consistency handler.



(a) Safety department graph.



(b) Production department graph.

Figure 4.6: Initial data populated in TypeDB, depicting the relationships between departments, tasks, and devices.

It is the role of the consistency handler to ensure that the schema of the KG accurately reflects the devices and their corresponding modules and attributes as defined in their SDF specifications. To do this, the consistency handler uses TypeQL queries to define the devices and their corresponding modules as sub-types of the "device" and "module" types within the schema. This process is performed in real-time as messages containing information about the devices are received, allowing the handler to dynamically create the class semantic structure for any devices whose class is not yet known. In addition, the consistency handler also defines the attributes that the devices measure, enabling the graph to store data about these devices in a structured manner. We will explain the design and functioning of the consistency handler in detail in the following section.

4.4 Consistency handler

The consistency handler is the central component of the architecture and is responsible for processing the messages received from the devices in real-time through an MQTT client. This processing involves several tasks, including checking if the device class of the message sender is known and defining it in the schema if necessary, updating the attributes

of the device in the graph to match the incoming message, and analyzing the device to determine its appropriate place within the graph if it is not yet integrated.

In order to update the KG with the information received from the devices, the consistency handler must translate its operations into queries that are compatible with the TypeDB client. These queries are then sent to the KG, which manages their execution and updates the graph accordingly. The consistency handler relies on the TypeDB client to perform these operations on the graph in order to maintain the accuracy and integrity of the DT.

4.4.1 Flowchart

The flowchart for the consistency handler is shown in Figure 4.7. This flowchart illustrates the steps involved in processing incoming messages and updating the KG.

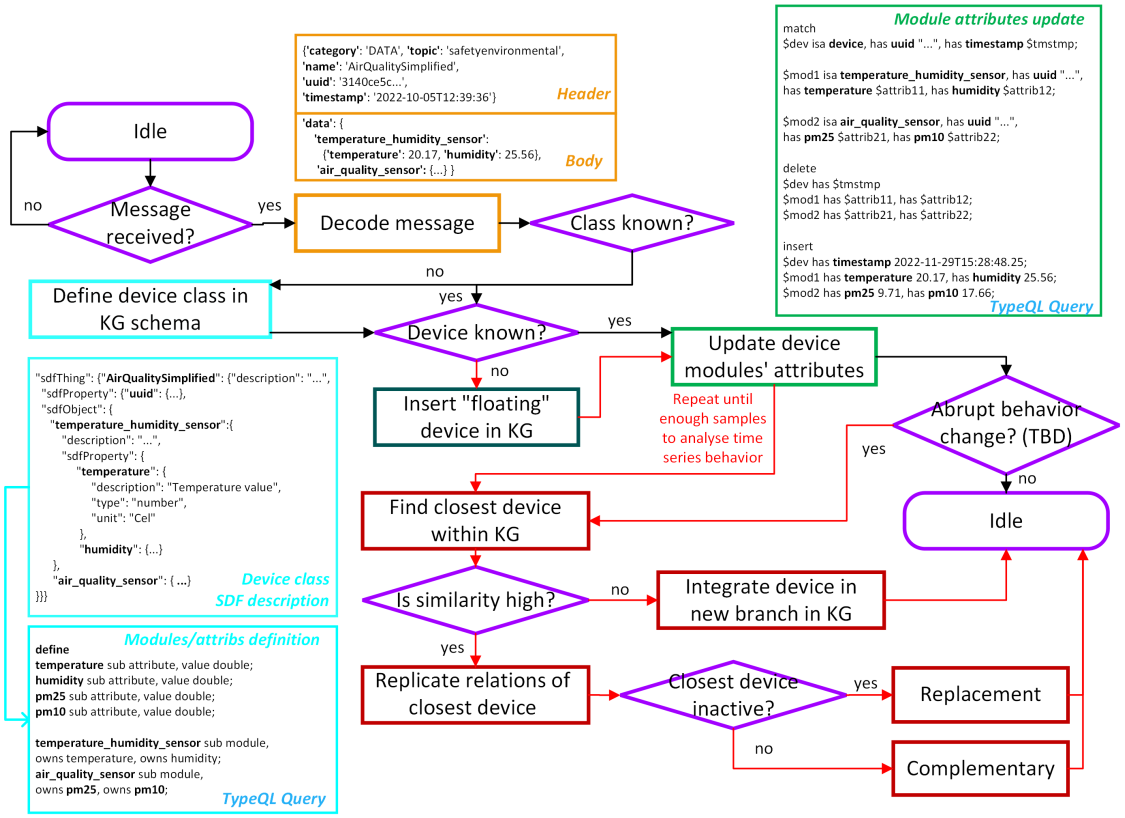


Figure 4.7: Consistency handler algorithm flowchart, depicting the key steps in the integration of device messages information into the knowledge graph, with examples of the messages receives and the queries generated.

When a message is received, it is processed in several steps. First, the header is decoded to obtain device and time information. The body of the message is then decoded to obtain the actual attribute data.

If the class of the device is not yet known, it must be defined within the KG schema. This is done by querying the SDF file of the class, analyzing it, and creating a query to the KG. If the device is not yet in the graph, it is inserted as a "floating" device. A floating device is one that is in the graph but has no connections to other objects in the graph.

The device attributes in the graph are then updated based on the data in the received message. If the device was not previously integrated into the graph, this update process

is repeated until enough buffered values are available to analyze the device's time series behavior. If the device was already integrated into the graph, we check for abrupt changes in behavior. For example, if a device that was previously measuring temperatures around 13 degrees Celsius starts measuring temperatures around 25 degrees Celsius, it may indicate a change in the device's function. In this case, the device would need to be reevaluated and potentially relocated within the graph. The detection of abrupt changes in behavior is left as future work and would require the use of an anomaly detector.

If the device needs to be integrated into the graph, a similarity analysis is performed. First, we use natural language processing NLP to determine which known device classes are most semantically similar to the class of the device under analysis. We then compare the time series of the attributes of the instances of these closest classes to the time series of the device under analysis. If one of the devices exhibits a very similar time series, they are considered to belong to the same task and the relations of the closest device are replicated in the new device. If there is no device that is close enough, the new device must be integrated into the graph from scratch, potentially creating a new branch.

Finally, if the new device was close enough to an existing device, we check if the closest device has been inactive lately. If it has, it is removed from the graph (the new device is a replacement), otherwise, it is kept in the graph (the new device is a complementary device).

The key steps of this process involve defining device classes in the schema, updating the attributes of existing devices, and integrating new devices into the graph using the implemented similarity metrics. These steps are described in more detail in the rest of the section.

4.4.2 Schema definition of device classes

Every time we receive a message from a device whose class is not yet known, we need to define its class within the KG schema to be able to integrate the device information within the graph. This process can be broken down into the following steps:

1. Check if the class name is known. If it is not, retrieve the SDF definition of the device class. This SDF definition will be examined to determine what modules and attributes the device implements. With this information, a full query will be constructed.
2. For each module within the device class, check if it has already been defined within the schema (e.g., the *Pick-Up Robot* and *Milling Robot* may share the *joint* module). If it has not been defined, add its definition to the query.
3. For each attribute within the modules, check if it has already been defined (e.g., several device classes may use the *temperature* attribute). If it has not been defined, add its definition to the query.
4. Now that the definition query is fully constructed, send it to the KG to define the device class within the schema.
5. Finally, construct an insert query to populate the device with default values for its attributes to initialize them. Send the insert query to the KG to commit these changes.

We can illustrate this process with an example. Imagine we receive the first message from a device of the class *Air Quality*, whose SDF definition is shown in Listing 4.1. Since no devices of this class or a similar class were previously present in the graph, we

need to define the entire device class in the KG schema. To do this, we examine the SDF definition, shown in Listing 4.3, and generate a TypeQL definition query which is then automatically sent to the KG. This query defines all the attributes and modules present in the SDF definition, since none of them were already in the schema.

Listing 4.3: Definition of *Air Quality* device class within schema. The attributes are defined first, and linked to the modules in their definion.

```
define

# ATTRIBUTES
temperature sub attribute, value double;
humidity sub attribute, value double;
pressure sub attribute, value double;
pm1 sub attribute, value double;
pm25 sub attribute, value double;
pm10 sub attribute, value double;

# MODULES
temperature_sensor sub module;
temperature_sensor owns temperature;

humidity_sensor sub module;
humidity_sensor owns humidity;

pressure_sensor sub module;
pressure_sensor owns pressure;

air_quality_sensor sub module;
air_quality_sensor owns pm1, owns pm25, owns pm10;

# DEVICE CLASS
airquality sub device;
```

Once the class is defined, we can add an initialized device of this class to the graph. This is done using the TypeQL insertion query shown in Listing 4.4. After the device has been initialized, it is already in the graph and its attributes will be updated as new messages from this device arrive.

Listing 4.4: Initialization of *Air Quality* device in the graph, assigning the default values to its attributes, and relating them to the appropriate modules and device instance.

```
insert

# DEVICE, UUID, AND TIMESTAMP
$dev isa airquality, has uuid "indoors_airquality", has
   timestamp 2022-12-28T11:54:49.38;

# MODULES AND ATTRIBUTES
$mod1 isa temperature_sensor,
has temperature 0.0;

$mod2 isa humidity_sensor,
has humidity 0.0;

$mod3 isa pressure_sensor,
has pressure 0.0;
```

```
15
16 $mod4 isa air_quality_sensor,
17 has pm1 0.0, has pm25 0.0, has pm10 0.0;
18
19 # RELATION INCLUDING MODULES IN DEVICE
20 $includes (device: $dev, module: $mod1, module: $mod2,
       module: $mod3, module: $mod4) isa includes;
```

Now that the *Air Quality* class has been defined, we have received a new message from a device of a new class called *Air Quality Simplified*, which is based on the *Air Quality* class but with some modifications and simplifications. The SDF definition for this new class can be found in Listing 4.5.

Listing 4.5: Semantic definition of *Air Quality Simplified* device class in SDF format, including properties for temperature, humidity, and pollutants measurement (PM2.5, PM10).

```
1  {
2  "namespace": {"eri": "https://ericsson.com/models"},
3  "defaultNamespace": "eri",
4  "info": {
5      "title": "Air Quality Simplified",
6      "version": "2022-12-12",
7      "copyright": "Copyright 2022 Ericsson",
8      "license": "BSD-3-Clause"
9  },
10 "sdfThing": {"AirQualitySimplified": {"description": "Monitors
       air quality through a set of sensors.",
11 "sdfProperty": {"uuid": {"sdfRef": "sdfData/sdfProperty/uuid"}},
12 "sdfObject": {
13 "temperature_humidity_sensor": {
14     "description": "Measures environmental temperature and
           humidity.",
15     "sdfProperty": {
16         "temperature": {"description": "Temperature value",
17             "type": "number",
18             "unit": "Cel"
19         },
20         "humidity": {"description": "Humidity value",
21             "type": "number",
22             "unit": "%"
23 }}},
24 "air_quality_sensor": {
25     "description": "Measures air pollutants.",
26     "sdfProperty": {
27         "pm25": {"description": "PM2.5 (bacteria, spores,
               pollen, toner dust)",
28             "type": "number",
29             "unit": "ug/m3"
30         },
31         "pm10": {"description": "PM10 (pollen, desert dust)",
32             "type": "number",
33             "unit": "ug/m3"
34 }}}}}}}
```

This new *Air Quality Simplified* class does not have any additional attributes compared to the *Air Quality* class. The only difference is the inclusion of a new module called

temperature_humidity_sensor. As a result, the definition query for this class is simpler, as shown in Listing 4.6.

Listing 4.6: Definition of *Air Quality Simplified* device class within schema. The attributes are defined first and linked to the modules in their definion.

```
define

# MODULES
temperature_humidity_sensor sub module;
temperature_humidity_sensor owns temperature, owns humidity;

air_quality_sensor owns pm25, owns pm10;

# DEVICE CLASS
airqualitysimplified sub device;
```

After that, this device can be initialized within the graph using the query shown in Listing 4.7.

Listing 4.7: Initialization of *Air Quality Simplified* device in the graph, assigning the default values to its attributes, and relating them to the appropriate modules and device instance.

```
insert

# DEVICE, UUID, AND TIMESTAMP
$dev isa airqualitysimp, has uuid "indoors_airqualitysimp",
   has timestamp 2022-12-28T12:23:57.04;

# MODULES AND ATTRIBUTES
$mod1 isa temperature_humidity_sensor,
has temperature 0.0, has humidity 0.0;

$mod2 isa air_quality_sensor,
has pm25 0.0, has pm10 0.0;

# RELATION INCLUDING MODULES IN DEVICE
$includes (device: $dev, module: $mod1, module: $mod2) isa
   includes;
```

We can also visualize the device class schema that we just defined as a graph using TypeDB Studio. An example of such a graph, resulting from the definition of the *Air Quality* and *Air Quality Simplified* classes, is shown in Figure 4.8. Notice how the modules *temperature_humidity_sensor* and *temperature_sensor* share the ownership of the *temperature* attribute. This can be helpful for understanding the relationships and structure of the data within the KG.

Using these queries, we have created a class in the schema and initialized the data for a device within the KG. This will allow us to store and manage information about the device in a structured manner. In the next step, we will explain how to update the values of the attributes of these integrated devices as we receive new messages from them.

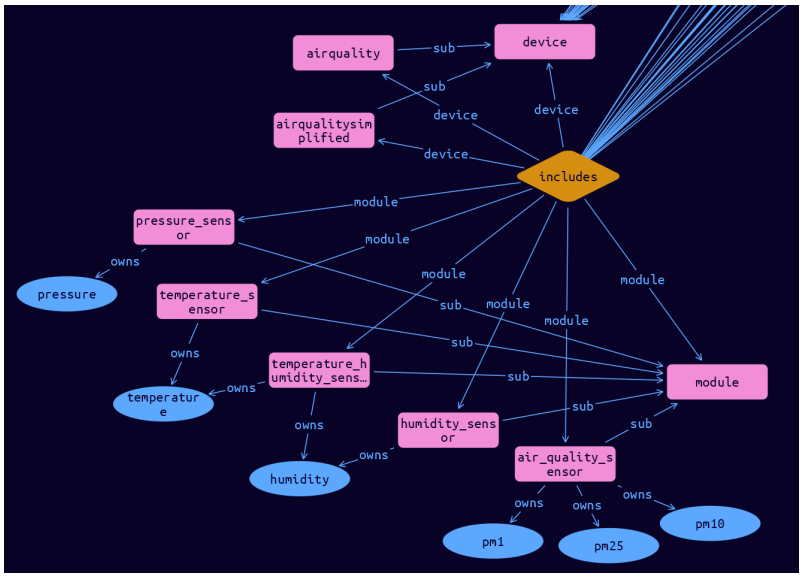


Figure 4.8: Visualization of the schema defined within the TypeDB knowledge graph of the *Air Quality* and *Air Quality Simplified* classes as a graph.

4.4.3 Update of device attributes

Once a device class is already defined and its instance initialized within the KG, we can update its attributes as new messages arrive. The process of updating the device attributes involves the following steps:

1. Check that the UUID of the device is already present in the graph. If it is, begin constructing the attributes update query.
2. Iterate over the body of the received message, which contains the attribute values to be updated.
3. For each attribute, identify the module that owns the attribute, delete the attribute ownership (since attributes of the same value might be owned by multiple objects), and insert a new attribute owned by the module with the new value contained in the message body.
4. Once all attributes have been processed, send the update query to the KG to commit the changes.

The outcome of this process is a TypeQL update query. For example, if the message received is the one shown in Listing 4.2, the generated update query will be the one presented in Listing 4.8.

Listing 4.8: Update of the attributes of an indoors *Air Quality* device according to the message data. First, we remove the ownership of the old attributes, and then we insert the ownership of the new attribute values.

```
match
$dev isa airquality, has uuid "indoors_airquality", has
    timestamp $tmstmp;

$mod1 isa temperature_sensor, has uuid "indoors_airquality",
    has temperature $attrib11;
$mod2 isa humidity_sensor, has uuid "indoors_airquality", has
    humidity $attrib21;
```

```
$mod3 isa pressure_sensor, has uuid "indoors_airquality", has
   pressure $attrib31;
$mod4 isa air_quality_sensor, has uuid "indoors_airquality",
   has pm1 $attrib41, has pm25 $attrib42, has pm10 $attrib43;

delete
$dev has $tmstmp;

$mod1 has $attrib11;
$mod2 has $attrib21;
$mod3 has $attrib31;
$mod4 has $attrib41, has $attrib42, has $attrib43;

insert
$dev has timestamp 2022-12-26T11:32:55.859;

$mod1 has temperature 19.88;
$mod2 has humidity 25.18;
$mod3 has pressure 0.65;
$mod4 has pm1 0.46, has pm25 8.70, has pm10 17.48;
```

We can also visualize how the device data looks in the graph after this query, shown in Figure 4.9.

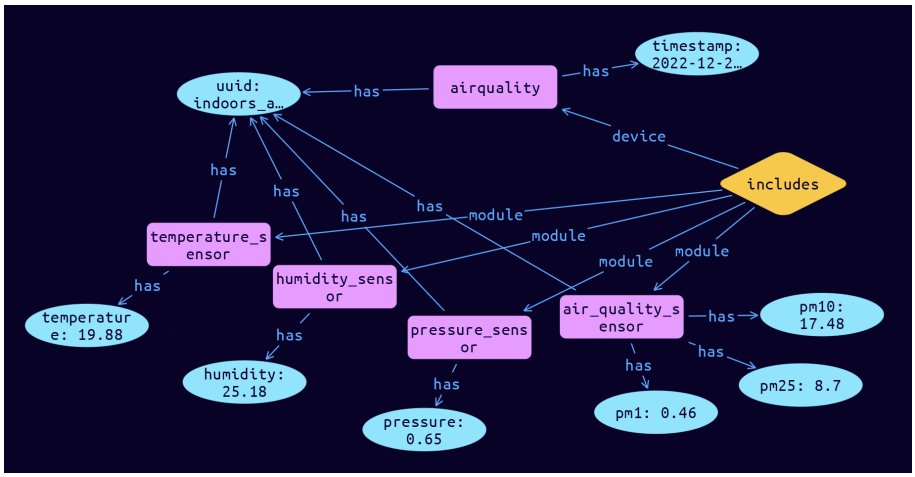


Figure 4.9: Visualization of the knowledge graph updated data of the indoors *Air Quality* device after processing its message.

It is important to note that the KG only stores the most recent value reported by each device, and not the entire set of values that have been reported. To store the time series of values reported by each device attribute, we would need to use a traditional database. However, this was not the focus of our work, so we instead stored these time series in a buffer within the script that runs the consistency handler. This allows us to perform analysis on the data and make use of it in our algorithms, but the data is not persisted in the KG itself.

4.4.4 Integration of new devices

To integrate a new device into the KG DT, the following steps are taken:

1. Receive a message from the device and check if it is already present in the system using its unique identifier (UUID). If the device is not present, begin the integration process.
2. Gather a sufficient number of samples from the new device to analyze its time series behavior. In this case, we gather 20 samples.
3. Determine the class of the new device and compare it to the existing classes in the system to find the top 3 most similar ones semantically. This is done using a specific process described in the following subsection.
4. Compare the time series behavior of the new device's attributes with the attributes of the device instances belonging to the previously identified closest classes. This allows us to further refine the analysis and identify the device instances that are most similar to the new device being integrated. This process is also described in more detail in a later subsection.
5. In case the similarity between the new device and its most closely matched one is not adequate, it is necessary to incorporate the new device into the graph from the beginning, with limited understanding of where it should be positioned. This is a task that has been deferred for future work and may involve the formation of a new branch or task within the graph.
6. If the similarity is high enough, consider the new device to be either a replacement or a complementary device to its closest match. In both cases, the relations within the graph of the closest match will be replicated on the new device. However, if the closest match has been inactive lately, consider the new device to be a replacement and remove the old device from the graph.

An example of the integration process for a new device is shown in Figure 4.10. The figure displays the console logs of the consistency handler as it receives and processes messages in real-time. The first two devices only need to update their attribute values in the graph because they are already integrated. The third message initiates the integration process for a new device. In this specific case, the consistency handler receives the 21st message from the new device and begins the integration process.

First, it identifies the closest classes to the new device's class, which are the *Air Quality*, *Noise Sensor*, and *Rain Sensor* classes. It then compares the time series attributes of the devices in these closest classes to the new device to infer that the closest device is the indoor *Air Quality* device. Since the similarity to this device is high enough, the consistency handler replicates its relations on the new device. Additionally, since the closest device has been inactive lately, the handler assumes that it has been replaced and removes it from the graph. Finally, once the new device is fully integrated, it updates its attributes on the graph.

4.4.5 Device class similarity

Device classes in the SDF format are defined using sdfThings, sdfObjects, and sdfProperties with specific names that have semantic meanings. To measure the similarity or distance between these classes, we need to calculate the semantic distance between them.

A semantic distance measures relatedness or similarity between concepts based on their meanings, and can be determined through various methods such as analyzing semantic relationships, comparing co-occurrence in a text corpus, or using natural language processing techniques.

```
(safetyenvironmental) -> NoiseSensor[7fc17e] msg received <N=28>
     |------> attributes updated <Tq=75ms>
     |------> msg processed <Tp=75ms | Avg.Tp=79ms>

(safetyenvironmental) -> AirQuality[outdoo] msg received <N=73>
     |------> attributes updated <Tq=142ms>
     |------> msg processed <Tp=144ms | Avg.Tp=127ms>

(safetyenvironmental) -> AirQualitySimplified[indoor] msg received <N=21>
     |------> closest classes computed in 3148ms
  candidate  score
 AirQuality     12
NoiseSensor      8
 RainSensor      4
     |------> closest devices computed in 2347ms
                    candidate  score
 AirQuality/indoors_airquality      3
AirQuality/outdoors_airquality      1
     |------> device integrated (relations replicated in KG) <Tq=61ms>
     |------> old device and its modules disintegrated from KG <Tq=105ms>
     |------> attributes updated <Tq=102ms>
     |------> msg processed <Tp=5802ms | Avg.Tp=391ms>
```

Figure 4.10: Console logs showing the integration of an *Air Quality Simplified* device as a replacement for an *Air Quality* device in the KG. The logs demonstrate the processing of messages from the *Noise Sensor* and outdoor *Air Quality* devices, where their attributes are updated. After receiving the 21st message from the new device (sufficient to analyze its time series behavior), the integration process begins. This includes computing the closest classes and closest devices, and then integrating the new device as a replacement for an old, non-active indoor *Air Quality* device.

This enables us to determine that "temperature" is more similar to "warmth" than to "tension", even if the Levenshtein distance shows that "tension" is closer to "temperature". The Levenshtein distance is a measure of the similarity between two strings, calculated by determining the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into the other.

For this work, we can use the Levenshtein distance because the names of the device classes, modules, and attributes were carefully chosen, but using a semantic distance would be more robust and is a future line of work.

Another challenge in comparing the similarity between device classes is the varying number of modules and rows among them, which leads to vectors of different dimensions when attempting to compare them. We have addressed this issue by dividing the definition of each device class into rows, with each row representing a specific attribute and the module it belongs to. This approach can be seen in Table 4.1, where the *Air Quality Simplified* class (as presented in Listing 4.5) and other classes are transformed into a set of rows with modules and their corresponding attributes.

To find the most similar classes to the *Air Quality Simplified* class, we will compute the Levenshtein distance between each row in the *Air Quality Simplified* class and each row in the other classes.

For each row in the *Air Quality Simplified* class, we find the rows in the other classes that have the smallest Levenshtein distance. We then give votes to the classes of these closest rows, with more votes being given to rows that are closer in distance.

Finally, we add up the votes for all rows in the *Air Quality Simplified* class to get a

sdfThing (Class)	sdfObject (Module)	sdfProperty (Attribute)
Air Quality Simplified	temperature_humidity_sensor	temperature humidity
	air_quality_sensor	pm25 pm10
Air Quality	temperature_sensor	temperature
	humidity_sensor	humidity
	pressure_sensor	pressure
	air_quality_sensor	pm1 pm25 pm10
Rain Sensor	rain_sensor	cumulative_depth
Pick-up Robot	joint	x_position y_position z_position
	actuator	x_position y_position z_position

Table 4.1: Device classes with their modules and attributes, where each row is associated with one attribute.

ranking of the most similar classes. The class with the most votes is the most similar to the *Air Quality Simplified* class, followed by the class with the second-most votes, and so on.

As shown in Figure 4.10, the result of the class similarity computation indicates that the *Air Quality* class is the most similar to the *Air Quality Simplified* class, followed by the *Noise Sensor* and *Rain Sensor* classes. This aligns with our expectations.

4.4.6 Device instance time series similarity

Once the most similar classes to the target class have been identified, the objective is to find devices within those classes that show similar behavior in the attributes they measure. To accomplish this, we will compare each attribute in the target device to the attributes of the devices in the most similar classes. We will only consider the attributes of the devices that are most similar to the target device, as identified by the closest classes computation. This helps us identify devices likely to exhibit similar behavior to the target device.

To identify devices with similar behavior in their attributes, we will compare the patterns of the attributes of the target device to the attributes of other devices. When we say that two devices have similar behavior in their attributes, we mean that the attributes of these devices show similar patterns when compared to each other.

Hence, to compare the attributes of the target device to the attributes of other devices, we can extract a time window from the time series of the attributes of the target device and slide it along the attributes of the other devices. By analyzing the patterns within this time window, we can determine which device exhibits the most similar behavior to the target device. It is important to note that this comparison assumes that the devices being compared generate data at equal time intervals.

We have selected the MASS algorithm to identify the device that behaves most

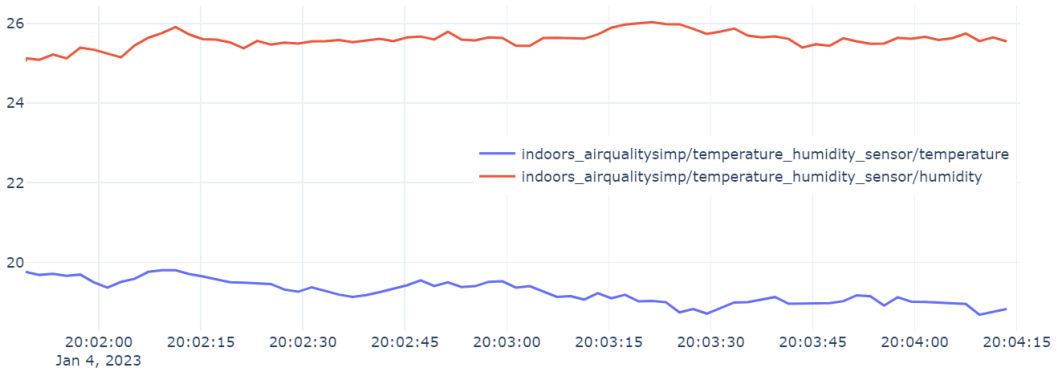


Figure 4.11: Temperature and humidity attributes for an indoors *Air Quality Simplified* device.

similarly to the target device. This algorithm computes the Euclidean distances between the attributes of the target device and the attributes of other devices after normalizing the data using z-normalization (standardizing the data by subtracting the mean of the data and dividing it by the standard deviation). These distances can be utilized to locate the devices with the most comparable attributes to the target device. MASS is known for its speed and efficiency, particularly for large datasets.

In Figure 4.11, we can see the time series of the "temperature" and "humidity" attributes measured by an *Air Quality Simplified* device placed indoors. If we compare these attribute time series to the time series of an *Air Quality* device placed both indoors and outdoors (as shown in Figure 4.4a), using the MASS algorithm, the closest device will be the indoor *Air Quality* device. This aligns with our expectations and with the results shown in the demo in Figure 4.10.

Chapter 5

Results and Analysis

In this chapter, the results of the prototype implementation of a DT based on a KG are presented. The prototype includes a KG-based DT, an IoT-based test environment, and a device data ingestion algorithm. The goal is to demonstrate how a KG can be used to create a DT that can help understand the behavior of an IoT platform. The test environment simulates IoT devices and sends data to the DT which ingests this data and updates the KG in real-time. The efficiency, validity, and reliability of the prototype are also evaluated. The potential impact of the prototype is discussed.

5.1 Knowledge Graph-based Digital Twin

The DT prototype was built on the TypeDB KG due to its scalability, ability to handle large amounts of data, reasoning capabilities, and support for a Python client. The ontology or schema was tailored to the specific domain of the automobile production line, including the relevant concepts and relationships. The KG was initialized with the initial state of the plant, including the devices and their connections. The adaptability of the DT was then tested using the consistency handler algorithm, which processes the messages generated by the IoT devices to reflect the state of the environment on the graph in real-time.

5.2 IoT-Based Test Environment

To evaluate the ability of the DT to ingest knowledge from IoT devices, an emulated test environment was established. This test environment simulated a simplified automobile manufacturing plant and included a range of devices such as sensors and actuators. These devices generated messages with pseudo-random data that was tailored to their class and purpose. A semantic description of the device classes was created using SDF, including the modules and attributes of each device. This allowed for the evaluation of the DT's ability to interpret and store knowledge from the messages generated by the emulated devices.

5.3 Device Data Ingestion

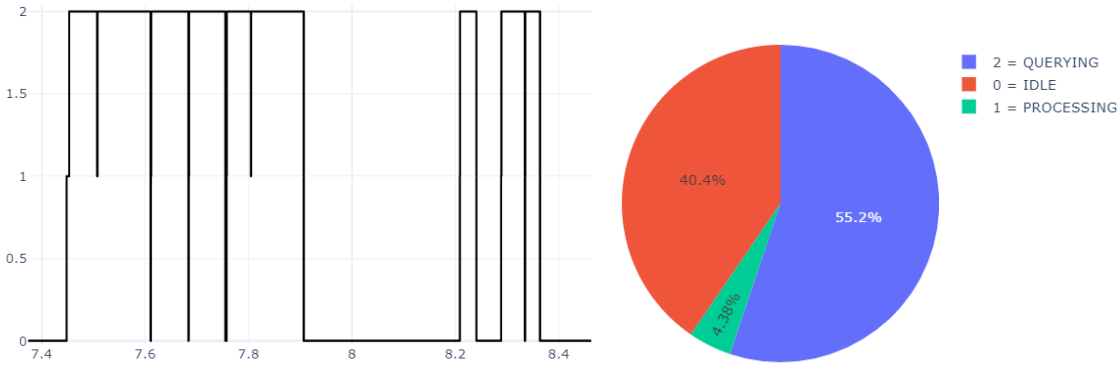
The consistency handler algorithm was developed to integrate the data from the IoT devices into the KG. This algorithm processes the messages by analyzing fields such as the class

of the device and its UUID. Based on the current state of the graph, it determines the appropriate actions to take in response to the messages.

The consistency handler algorithm is a significant result of this work. However, it is still incomplete. Specifically, it lacks the integration of devices when they do not show any similarity to the existing devices. In addition, it is not fully robust. The algorithm for computing the similarity between classes and behavior between device instances can be improved. Despite these limitations, the consistency handler algorithm demonstrates progress towards the integration of unforeseen devices within the graph. It also highlights the main challenges that must be overcome to achieve this goal.

5.4 Efficiency Analysis

The performance of the KG-based DT is dependent on its ability to efficiently process data from IoT device messages. The agent responsible for this task, known as the KG agent, receives messages from the MQTT broker, processes them, and generates the appropriate queries to update the KG. Currently, this agent processes messages sequentially, meaning that it handles one message at a time before moving on to the next. This results in three main states for the agent: IDLE (state 0), PROCESSING (state 1), or QUERYING (state 2).



(a) Zoomed-in evolution over time.

(b) Percentage of time spent.

Figure 5.1: Insights about the state of the knowledge graph agent.

The efficiency of the algorithm can be analyzed by comparing the amount of time spent processing messages or querying the KG to the amount of time spent waiting for new messages. Figure 5.1a shows that during a specific time window, the agent first receives a message requiring two queries, then receives two messages requiring only attribute updates, followed by another message that requires both class definition and attributes update queries. After this, the agent remains IDLE for 300ms before updating the attributes of three new messages.

Figure 5.1b indicates that the agent spends the majority of its time in IDLE or QUERYING states and very little time in the PROCESSING state. This suggests that increasing the messaging rate of the devices would produce a bottleneck, as the time spent querying the KG limits the message rate that can be supported. To improve efficiency, we could focus on optimizing the KG client to process queries faster or process messages in parallel while accounting for potential conflicts caused by shared variables.

5.5 Reliability Analysis

To ensure the reliability of our DT prototype, several experiments and tests were conducted. In each experiment, the functionalities of the prototype were compared against the expected result, by comparing its decisions to those that would be made by a human expert. The experiments were repeated at least 5 times to ensure the consistency of the results.

The experiments conducted to test the reliability of our prototype are the following:

- Test environment emulation: verify that the data generated by the devices is personalized to the class of the device and the task it performs, and that the devices publish messages at the appropriate intervals. Ensure proper connection of devices to the MQTT broker, and verify that the broker is functioning properly by receiving and sending messages.
- Knowledge graph deployment: verify that the container hosting the graph can receive queries, process them, and return the appropriate information.
- Schema definition: verify that the initial schema is correct, as well as the definition of the classes within the schema by visualizing its graph description with TypeDB Studio.
- Attributes update: verify that the values of the device attributes (such as temperature, position, etc.) match the last values shared by the devices in the generated messages by comparing the information in the message with the values in the graph after a device message is processed.
- New devices integration: check that the new device was integrated where it was expected within the graph. This included checking that the similarity analysis is returning the expected closest devices and that the new device was connected with the expected devices and tasks within the graph after the integration. We did not include an analysis of the correct integration of devices that did not show any similarity to any other device within the graph.

There were some sources of error, such as incorrect devices being detected as the closest ones to the new device in some cases. We discovered that this was mostly caused by two factors: the sliding window, used to compare the time series, being too small, and the excessive randomization of the data generated by the devices. To increase reliability, we solved these issues by increasing the size of the sliding window and increasing the dependency of the data generated on the class of the device and the task it performs. The remaining sources of error were mostly related to an incorrect generation of the queries to the database, and they were solved by improving the algorithm that generates them.

5.6 Validity Analysis

To validate our DT prototype, several verification and validation tasks were conducted. To ensure the results were as relevant to real-world applications as possible, the test environment was designed based on a simplified automobile manufacturing plant, and the data generated by the devices was designed to have properties such as periodicity (for repetitive tasks) and oscillation around a ground truth value (for example, for temperature measurements). However, to ensure the validity of the prototype in a real-world scenario, further testing with real data from a manufacturing plant needs to be conducted as a future

line of work.

5.7 Discussion

Overall, our prototype effectively demonstrated the feasibility of using a KG to store and manage data for a DT that reflects an IoT platform. The scalability and flexibility of the TypeDB KG enabled handling the data from the IoT devices in our test environment, while being capable of storing more complex and larger data structures in future applications. The consistency handler algorithm effectively integrated the data into the graph and the real-time updates and integration of unforeseen devices highlighted the adaptability of the system.

However, there were also some challenges and limitations in our approach. Defining the ontology or schema required significant domain-specific knowledge and careful consideration of the concepts and relationships involved. Also, generating relevant data for the emulated IoT devices was difficult, as it had to mimic the behavior of real devices. Additionally, the consistency handler algorithm was complex and required significant development effort, particularly in the integration of unforeseen devices into the graph when they did not show meaningful similarities to any of the other devices.

Chapter 6

Conclusions and Future work

In this chapter, we will summarize the main conclusions of our work on the design of a KG-based DT for IoT platforms. We will also discuss areas for future work, highlighting opportunities for further research and development in this field. Additionally, we will reflect on the economic, social, environmental, and ethical implications of our work.

6.1 Conclusions

In conclusion, this thesis presented a prototype implementation of a DT for IoT platforms that utilizes a KG as its underlying representation. The use of a KG enables efficient management of the vast amount of data generated by interconnected devices in real-time. The primary focus of the prototype was on the integration of unanticipated devices into the existing logical structure of the KG, allowing the system to adapt to changing environments.

One of the major challenges that we encountered during the development of the prototype was integrating new devices that do not have high similarity to the existing ones in the KG. This is a current and ongoing challenge that requires further research. To address this challenge, we used the open-source software TypeDB KG and a Python client for efficient querying and reasoning. However, there are still areas for improvement, such as enhancing the similarity metrics and optimizing the communication between the Python client and the graph.

In summary, this research demonstrates a promising approach for managing and understanding the complexity of IoT systems using KGs and DT technology. However, further research is needed to address the challenges of integrating new devices that do not show high similarity to the existing ones.

6.2 Future work

Future research in this field could involve conducting more tests and validation of the KG-based DT in real-world settings, such as manufacturing, healthcare, and transportation, to determine its effectiveness in different industries and to quantify the potential cost savings and efficiency gains that can be achieved.

Additionally, there is potential to improve the similarity metric used to integrate unanticipated devices into the KG. The similarity metric used in this work, which is based

on device classes, could be enhanced by using semantic distance as it may provide a more robust approach. Also, research on alternative methods of integrating unanticipated devices, such as machine learning techniques, can be done.

Another area of future work is to optimize the communication between the KG agent and the KG, which can be done by experimenting with different libraries and protocols, in order to find the best way to retrieve and update the graph data efficiently and effectively.

Scalability should also be considered, as the KG must handle a large number of IoT devices and vast amounts of data. The system should be tested for high loads and optimized to handle a high number of devices, in order to remain efficient and effective even at a large scale.

To summarize, future work should aim to validate the results and scalability of the system by collecting and analyzing real-world data from IoT device platforms. Additionally, it would be beneficial to conduct further research on alternative methods for integrating unanticipated devices into the system and optimizing the communication between the agent and the graph. These improvements will help to ensure the robustness and efficiency of the DT in various industries, making it a valuable tool for monitoring, prediction, and optimization of systems.

6.3 Reflections

Our work on implementing a KG-based DT for IoT platforms can bring cost savings to various industries by reducing downtime and improving decision-making through real-time monitoring and optimization. It can also improve efficiency by reducing the need for manual integration and maintenance. It also has the potential to improve the performance and reliability of systems, which could lead to improved productivity and better service for the end user. From an environmental standpoint, our work could contribute to the reduction of waste and energy consumption in various industries, by improving the efficiency of systems and reducing downtime. However, it also raises questions about privacy and data security, and it is important to ensure that data generated by IoT devices is collected, stored, and used responsibly, with proper precautions in place to protect against data breaches and unauthorized access. Also, the potential impact on employment has to be considered if the implementation of the digital twin leads to the automation of certain jobs.

References

[1] IBM, "What is a digital twin?" [Online]. Available: https://www.ibm.com/topics/what-is-a-digital-twin

[2] Oracle, "What is the Internet of Things?" [Online]. Available: https://www.oracle.com/internet-of-things/what-is-iot/

[3] S. University, "What is a Knowledge Graph?" [Online]. Available: https://web.stanford.edu/~vinayc/kg/notes/What_is_a_Knowledge_Graph.html

[4] Vaticle, "Vaticle TypeDB." [Online]. Available: https://vaticle.com/

[5] MQTT, "MQTT - The Standard for IoT messaging." [Online]. Available: https://mqtt.org/

[6] R. Brachman and H. Levesque, *Knowledge representation and reasoning*. Elsevier, 2004.

[7] Neo4j, "Neo4j Graph Data Platform." [Online]. Available: https://neo4j.com/

[8] A. S. Foundation, "Apache Jena." [Online]. Available: https://vaticle.com/

[9] Vaticle, "TypeDB and TypeQL documentation." [Online]. Available: https://docs.vaticle.com/docs/general/introduction

[10] OneDM, "Semantic Definition Format (SDF) for Data and Interactions of Things." [Online]. Available: https://onedm.org/sdflanguage/

[11] H. Cheng, P. Zeng, L. Xue, Z. Shi, P. Wang, and H. Yu, "Manufacturing ontology development based on Industry 4.0 demonstration production line," in *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*. IEEE, 2016, pp. 42–47.

[12] W. Wang, S. De, G. Cassar, and K. Moessner, "Knowledge representation in the internet of things: semantic modelling and its applications," *Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije*, vol. 54, no. 4, pp. 388–400, 2013.

[13] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks, "Characterising the Digital Twin: A systematic literature review," *CIRP Journal of Manufacturing Science and Technology*, vol. 29, pp. 36–52, 2020.

[14] M. Singh, E. Fuenmayor, E. P. Hinchy, Y. Qiao, N. Murray, and D. Devine, "Digital twin: Origin to future," *Applied System Innovation*, vol. 4, no. 2, p. 36, 2021.

[15] J. Vachálek, L. Bartalskỳ, O. Rovnỳ, D. Šišmišová, M. Morháč, and M. Lokšík, "The digital twin of an industrial production line within the industry 4.0 concept," in *2017 21st international conference on process control (PC)*. IEEE, 2017, pp. 258–262.

[16] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference*, 2010, pp. 1–6.

[17] S. Ji, S. Pan, E. Cambria, P. Marttinen, and S. Y. Philip, "A survey on knowledge graphs: Representation, acquisition, and applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, 2021.

[18] D. Le-Phuoc, H. N. M. Quoc, H. N. Quoc, T. T. Nhat, and M. Hauswirth, "The graph of things: A step towards the live knowledge graph of connected things," *Journal of Web Semantics*, vol. 37, pp. 25–35, 2016.

[19] D. Chandrasekaran and V. Mago, "Evolution of semantic similarity—a survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–37, 2021.

[20] L. Yujian and L. Bo, "A normalized Levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[21] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1542–1552, 2008.

[22] A. Mueen, S. Zhing, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh, "The Fastest Similarity Search Algorithm for Time Series Subsequences under Euclidean Distance," August 2022, http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html.

[23] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, "A review of relational machine learning for knowledge graphs," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 11–33, 2015.

[24] NumPy.org, "NumPy documentation." [Online]. Available: https://numpy.org/doc/

[25] CloudMQTT, "MQTT for Python documentation." [Online]. Available: https://www.cloudmqtt.com/docs/python.html

[26] Stumpy, "Stumpy documentation." [Online]. Available: https://stumpy.readthedocs.io/en/latest/index.html

[27] seatgeek, "Fuzzy String Matching in Python." [Online]. Available: https://github.com/seatgeek/thefuzz

[28] Vaticle, "Vaticle TypeDB - Python Client." [Online]. Available: https://github.com/vaticle/typedb-client-python

[29] Eclipse, "Eclipse Mosquitto documentation." [Online]. Available: https://mosquitto.org/documentation/

[30] Vaticle, "Vaticle TypeDB - Studio." [Online]. Available: https://github.com/vaticle/typedb-studio

Appendix A

Initial Knowledge Graph

A.1 Schema definition in TypeQL

The initial schema or ontology of the KG is defined using the TypeQL query shown in Listing A.1.

Listing A.1: Initial schema definition query for a type system using TypeQL. The schema defines a system for managing departments, tasks, devices, and modules within an organization. It includes three attributes (uuid, name, timestamp) used to define the properties of the entities. Four relations (execution, sequence, needs, includes) are also defined to describe the relationships between the entities. Four entities (department, task, device, module) are defined, each making use of the attributes and relations defined earlier. The device and module entities are also defined as abstract, indicating that they are not meant to be instantiated on their own.

```
define

# ATTRIBUTES
uuid sub attribute, value string;
name sub attribute, value string;
timestamp sub attribute, value datetime;

# RELATIONS
execution sub relation,
    relates department,
    relates task;

sequence sub relation,
    relates predecessor,
    relates successor,
    relates connectedby;

needs sub relation,
    relates task,
    relates device;

includes sub relation,
    relates device,
    relates module;

# ENTITIES
```

```
department sub entity,
    owns name,
    plays execution:department;

task sub entity,
    owns name,
    plays execution:task,
    plays sequence:predecessor, plays sequence:successor,
    plays needs:task;

# Device Abstract Class
device sub entity, abstract,
    owns uuid, owns timestamp,
    plays needs:device,
    plays includes:device,
    plays sequence:connectedby;

# Module Abstract Class
module sub entity, abstract,
    owns uuid,
    plays includes:module;
```

A.2 Data insertion in TypeQL

The initial data for the KG is inserted using the TypeQL query shown in Listing A.2.

Listing A.2: Initial data insertion query for a type system using TypeQL. The query is inserting sample data for departments, tasks, connectors between tasks, and devices for an organization. The data being inserted includes two departments, Production and Safety/Environmental, with their respective tasks.

```
insert

# Department
$dept1 isa department, has name "Production";
$dept2 isa department, has name "Safety/Environmental";

# Connectors between tasks
$convbelt1 isa conveyorbelt, has uuid "fbeaa5f3-e532-4e02-8429-c77301f46470";
$convbelt2 isa conveyorbelt, has uuid "f169a965-bb15-4db3-97cd-49b5b641a9fe";
$convbelt3 isa conveyorbelt, has uuid "3140ce5c-0d08-4aff-9bb4-14a9e6a33d12";
$convbelt4 isa conveyorbelt, has uuid "a6f65d7a-019a-4723-9b81-fb4a163fa23a";
$convbelt5 isa conveyorbelt, has uuid "f342e60b-6a54-4f20-8874-89a550ebc75c";

# Tasks
# Production
$task1 isa task, has name "Initialization";
$execution1 (department: $dept1, task: $task1) isa execution;
```

```
$sequence1 (predecessor: $task1, successor: $task2) isa sequence;
$task2 isa task, has name "Underpan Configuration";
$execution2 (department: $dept1, task: $task2) isa execution;
$sequence2 (predecessor: $task2, successor: $task3, connectedby:$convbelt1) isa sequence;
$task3 isa task, has name "Body Configuration";
$execution3 (department: $dept1, task: $task3) isa execution;
$sequence3 (predecessor: $task3, successor: $task4, connectedby:$convbelt2) isa sequence;
$task4 isa task, has name "Vehicle Scanning";
$execution4 (department: $dept1, task: $task4) isa execution;
$sequence4 (predecessor: $task4, successor: $task5, connectedby:$convbelt3) isa sequence;
$task5 isa task, has name "Window Milling";
$execution5 (department: $dept1, task: $task5) isa execution;
$sequence5 (predecessor: $task5, successor: $task6, connectedby:$convbelt4) isa sequence;
$task6 isa task, has name "Quality Check";
$execution6 (department: $dept1, task: $task6) isa execution;
$sequence6 (predecessor: $task6, successor: $task7) isa sequence;
$sequence7 (predecessor: $task6, successor: $task8, connectedby:$convbelt5) isa sequence;
$task7 isa task, has name "Artificial Repair";
$execution7 (department: $dept1, task: $task7) isa execution;
$task8 isa task, has name "Product Completion";
$execution8 (department: $dept1, task: $task8) isa execution;

# Safety
$task9 isa task, has name "Indoors Monitorization";
$execution9 (department: $dept2, task: $task9) isa execution;
$task10 isa task, has name "Outdoors Monitorization";
$execution10 (department: $dept2, task: $task10) isa execution;
$task11 isa task, has name "Safety Alarms";
$execution11 (department: $dept2, task: $task11) isa execution;

# Initial Devices - PRODUCTION LINE
# Initialization Task
$dev1 isa tagscanner, has uuid "8a40d136-8401-41bd-9845-7dc8f28ea582";
$needs1 (task: $task1, device: $dev1) isa needs;
$dev2 isa productioncontrol, has uuid "3d193d4c-ba9c-453e-b98b-cec9546b9182";
$needs2 (task: $task1, device: $dev2) isa needs;

```

```
# Underpan Configuration Task
$dev3 isa pickuprobot, has uuid "5f3333b9-8292-4371-b5c5-c1ec21d0b652";
$needs3 (task: $task2, device: $dev3) isa needs;
$dev4 isa piecedetector, has uuid "45d289e7-4da6-4c10-aa6e-2c1d48b223e2";
$needs4 (task: $task2, device: $dev4) isa needs;

# Body Configuration Task
$dev5 isa pickuprobot, has uuid "bodyconfig_pickuprob";
$needs5 (task: $task3, device: $dev5) isa needs;
$dev6 isa clampingrobot, has uuid "5ee2149f-ef6e-402b-937e-8e04a2133cdd";
$needs6 (task: $task3, device: $dev6) isa needs;
$dev7 isa drillingrobot, has uuid "98247600-c4fe-4728-bda6-ed8fadf81af2";
$needs7 (task: $task3, device: $dev7) isa needs;
$dev8 isa piecedetector, has uuid "d7295016-4a54-4c98-a4c1-4f0c7f7614b5";
$needs8 (task: $task3, device: $dev8) isa needs;
$dev9 isa posedetector, has uuid "2c91bd9d-bdfc-4a6b-b465-575f43897d59";
$needs9 (task: $task3, device: $dev9) isa needs;

# Vehicle Scanning
$dev10 isa configurationscanner, has uuid "0d451573-243e-423b-bfab-0f3117f88bd0";
$needs10 (task: $task4, device: $dev10) isa needs;
$dev11 isa faultnotifier, has uuid "f1b43cb8-127a-43b5-905d-9f145171079es";
$needs11 (task: $task4, device: $dev11) isa needs;

# Window Milling
$dev12 isa pickuprobot, has uuid "windowmilling_pickuprob";
$needs12 (task: $task5, device: $dev12) isa needs;
$dev13 isa millingrobot, has uuid "5ce94c31-3004-431e-97b3-c8f779fb180d";
$needs13 (task: $task5, device: $dev13) isa needs;
$dev14 isa posedetector, has uuid "1df9566a-2f06-48f0-975f-28058c6784c0";
$needs14 (task: $task5, device: $dev14) isa needs;

# Quality Check
$dev15 isa qualityscanner, has uuid "fd9ccbb2-be41-4507-85ac-a431fe886541";
$needs15 (task: $task6, device: $dev15) isa needs;
$dev16 isa faultnotifier, has uuid "5bb02f4b-0dfe-45d4-8a87-e902e6ea0bf6";
$needs16 (task: $task6, device: $dev16) isa needs;

# Artificial Repair
$dev17 isa repaircontrol, has uuid "4525aa12-06fb-484f-be38-58afb33e1558";
$needs17 (task: $task7, device: $dev17) isa needs;

# Product Completion
```

```
98  $dev18 isa pickuprobot, has uuid "ae5e4ad3-bd59-4dc8-b242-
       e72747d187d4";
99  $needs18 (task: $task8, device: $dev18) isa needs;
100 $dev19 isa posedetector, has uuid "f2d73019-1e87-48a7-b93c-
       af0a4fc17994";
101 $needs19 (task: $task8, device: $dev19) isa needs;
102
103 # Initial Devices - SAFETY / ENVIRONMENTAL
104 # Indoors Monitorization
105 $dev20 isa airquality, has uuid "indoors_airquality";
106 $needs20 (task: $task9, device: $dev20) isa needs;
107 $dev21 isa noisesensor, has uuid "7fc17e8f-1e1c-43f8-a2d1-9
       ff4bcfbf9ff";
108 $needs21(task: $task9, device: $dev21) isa needs;
109 $dev22 isa smokesensor, has uuid "5a84f26b-bf77-42d3-ab8a-83
       a214112844";
110 $needs22(task: $task9, device: $dev22) isa needs;
111 $dev23 isa seismicsensor, has uuid "4f1f6ac2-f565-42af-a186-
       db17f7ed94c2";
112 $needs23(task: $task9, device: $dev23) isa needs;
113
114 # Outdoors Monitorization
115 $dev24 isa airquality, has uuid "outdoors_airquality";
116 $needs24 (task: $task10, device: $dev24) isa needs;
117 $dev25 isa rainsensor, has uuid "70a15d0b-f6d3-4833-b929-74
       abdff69fa5";
118 $needs25 (task: $task10, device: $dev25) isa needs;
119 $dev26 isa windsensor, has uuid "f41db548-3a85-491e-ada6-
       bab5c106ced6";
120 $needs26 (task: $task10, device: $dev26) isa needs;
121
122 # Safety Alarms
123 $dev27 isa indoorsalarm, has uuid "4d36d0c4-891f-44ec-afe1
       -278258058944";
124 $needs27 (task: $task11, device: $dev27) isa needs;
125 $dev28 isa outdoorsalarm, has uuid "b60108c2-46a3-4b67-9b8d
       -38586cb3039d";
126 $needs28 (task: $task11, device: $dev28) isa needs;
```

TRITA-EECS-EX-2023:116