An Architecture and Information Meta-model for Back-end Data Access via Digital Twins

Somayeh Malakuti,
Prerna Juhlin,
Jens Doppelhamer,
Johannes Schmitt
ABB AG Corporate Research Center Germany
{firstname.lastname}@de.abb.com

Thomas Goldschmidt,
Aleksander Ciepal
ABB Process Automation Digital
thomas.goldschmidt@de.abb.com, aleksander.ciepal@pl.abb.com

2021 IEEE 26th International Conference on Emerging Technologies and Factory Automation (ETFA) | 978-1-7281-2989-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ETFA45728.2021.9613724

***Abstract*—The lifecycle data of industrial devices is typically maintained in separate data sources operating in silos. The lack of interoperability between the data sources due to the usage of different APIs, data formats and data models results in time-consuming and error-prone manual data exchange efforts. The notion of *digital twins* is known to be a solution to the data silo problem and the associated interoperability issues. Despite many studies on digital twins, there is still a need for common architectures that offer means for defining digital twins, ingesting backend data in the digital twins, and enabling interoperable data exchange across the lifecycle of the devices via their digital twins. This paper aims to close this gap by proposing a cloud-based architecture, a common information meta-model for defining the digital twins, and variety of APIs to query the lifecycle data of interest via the digital twins.**

***Index Terms*—Digital Twin, Cloud-based Architecture, Common Information Meta-model, Data-as-a-Product, GraphQL**

I. Introduction

Industrial companies produce different kinds of industrial devices, which have different lifecycle phases such as type design, order/sales, engineering, production, installation, operation, and service. The lifecycle data of the devices is usually stored in different sources, utilizing different data formats and models considered suitable for respective application types.

The separation of data sources leads to the so-called *data silo* problem. Lack of systematic means to integrate the lifecycle data of devices to serve a variety of use cases leads to the so-called *solutions silos* problem: Use case-specific solutions may be provided to integrate data from various sources by defining dedicated data pipelines and data models that are not sufficiently interoperable. The associated lack of interoperability at multiple levels of data access APIs, data formats (syntax), and data models (semantics) results in error-prone and time-consuming manual data exchange efforts between the lifecycle phases, e.g. manually entered information or copy-paste between different data models. This also makes it difficult to combine the data for harnessing by analytics applications. As a result, the agility of the companies in responding to changing customers' requirements eventually reduces.

During past years the notion of digital twins has evolved from simulation models to the digital representation of an entity (e.g., device, system), which enables accessing dispersed lifecycle data via unified APIs of digital twins for different use cases [1]. Hence, instead of accessing individual data sources separately, applications can make use of digital twin APIs to access their desired lifecycle data.

Digital twin solutions should consider four aspects: physical entities, digital twin data, virtual models, and services [2]. Studies [3], [4], [5], [2] show that a significant body of work focuses on the simulation and optimization aspects (i.e., virtual models and services) of digital twins for specific use cases in a specific lifecycle phase of devices. These studies show that there is a research gap across the entire lifecycle of devices, especially considering earlier lifecycle phases. Besides, the need for architectures and platforms, which offer means for defining digital twins data, accessing back-end data via the digital twins, and enabling interoperable interactions among digital twins for various use cases, has been observed.

This paper fills this research gap via presenting a digital twin-based architecture and platform to address the data silo problem, in such a way that organization-wide digital twin models, data pipelines, and practices can be established to serve a variety of use cases in a common way.

Our solution has the following novelties, in comparison with our previous work [6], [7], [8]:

- An architecture and its realization in a cloud-based platform enabling the management of digital twins
- A common information meta-model for the digital twins in which various lifecycle data can be represented
- Declarative connectivity to the external data sources to access their data on-demand, and to map it to the common information meta-model
- Open-endedness with new data sources and modular extensibility of the digital twins with new models upon the availability of the new data sources
- Various APIs based on Boolean predicates and GraphQL [9] for querying the lifecycle data, regardless of whether the data is stored within the platform or in the external data sources

These novel features enable realizing the notions of *plug and provide* and *plug and consume* for data-driven applications. As a result, the required time and effort to develop and maintain the data-driven applications can be reduced.

II. Problem Statement

It is very common that in industrial systems, the lifecycle data of devices is stored and maintained in silos. The bottom part of Figure 1 illustrates an example case, where a device has two variants: low-voltage and medium-voltage. Such devices have different lifecycle phases such as selection, engineering, operation, and service. The terms Engineering Technology (ET), Information Technology (IT) and Operational Technology (OT) refer to the adopted technologies in these phases. Different tools are used for the selection and engineering phases, which generate the ET data of the devices. These tools are usually disjoint from each other; nevertheless, some device parameters from the selection phase are relevant to the engineering phase. The OT data represent the values of operating parameters, events and alarms of the devices that occur during the operation of the devices.

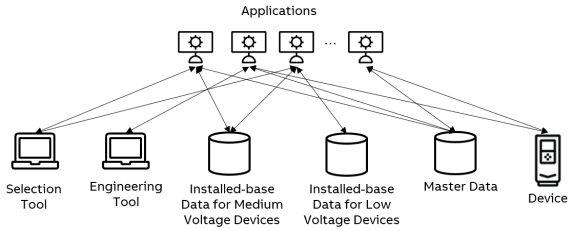

Fig. 1. Data silos in industrial systems.

The type design data and the installed-based data belong to the category of IT data. The type data is maintained in the master data system. The installed-based data contains installation information (customer, site, maintenance activities) as well as some technical information of the devices. As these two device variants are managed by different internal organizations, there are separate relational databases to maintain their installed-base data. The databases have different implementation technologies.

The combination of ET, IT and OT data is essential for various Industrie 4.0 use cases [7], such as information exchange across the value stream, cloud-based integrated engineering toolset, and plug and play of field devices. However, we face the following problems for systematically integrating dispersed data to realize these use cases:

- *Insufficient interoperability:* Existing data sources are specially designed with data models that are suitable for their particular domain, without sufficient attention to be interoperable with other domains. However, use cases spanning multiple data sources require both syntactic and semantic interoperability across the data sources, within and across the boundaries of the domains. Interoperability can be achieved via adopting industry standards for modeling the data. However, due to the standardization deadlock [10] problem, such standards may not always be available at the right time, or it may not be feasible to change the existing systems to adhere to the standards. Lack of industry standards and/or organization-wide data models to represent different domain data may cause different teams to develop their data models to integrate the data from different sources for each use case. Such heterogeneous data models reduce the syntactic and semantic interoperability across different use cases.
- *Data pipeline jungle:* Unharmonized data models and APIs will lead to dedicated data pipelines for connecting to the data sources, ingesting the data, transforming the data, etc. With *m* applications and *n* data sources, there might be *m*n* distinguished dependencies. The excessive number of data pipelines, the associated (sometimes repetitive) effort to develop, test and maintain them, and the tight coupling of the applications to the respective data sources will reduce the productivity of development teams to respond to changing customers' demands.

In summary, the lack of systematic means to integrate and manage data will result in a new technical debt that we name *solutions silos*, which reduce the agility of the organizations in responding to new customers' requests.

III. A Digital Twin-based Solution

Although the notion of digital twin is mostly associated with simulation and/or analytics models of devices, this notion has evolved over the years to address the data silo problem across products' lifecycle [1], [11], [12]. As Figure 2 shows, we explore means to implement digital twins for this matter.

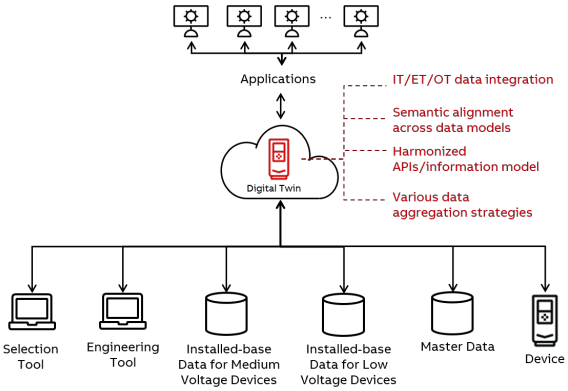

Fig. 2. Digital twin for addressing the data silo problem.

To systematically address the data silo problem without creating solutions silos, we require architectures, digital twin models and data pipelines that can serve a variety of use cases. In the following, we first outline the requirements that must be fulfilled towards this aim, followed by our proposed digital twin platform.

A. Requirements

- *Open-ended with new data sources and lifecycle data:* Since different use cases may require accessing different

data from different sources, mechanisms are required to discover/declare new data sources and the data that is offered by them. We consider the following data sources in this paper: a) Physical devices that are the source of their OT data; b) Databases that are the source of the IT data for several thousand or even million devices, but not all data may be of interest to each use case; c) Various engineering tools that are the source of the ET data are represented in various formats such as XML and JSON.

- *Data ingestion strategies:* Different use cases may require push-based and/or pull-based data ingestion mechanisms. The push-based mechanism is, for example, suitable for ingesting the OT data that is generated in high frequency, as well as the ET data that is generated by tools. The pull-based mechanism on the other hand may be more suitable for querying the IT data from databases on demand. To avoid data inconsistency issues, it should be possible to keep the original IT database as the source of truth from which data is retrieved on-demand.
- *Harmonized information meta-model:* Due to the long lifetime of industrial systems, it is not also feasible to replace existing domain data models with standardized data models even if such standards exist. Therefore, a common information meta-model must be in place to represent various lifecycle data, to achieve syntactic interoperability across use cases and to make applications agnostic of the heterogeneity of source data models. Mechanisms must be provided to map existing data models to the common information meta-model on-demand, and to modularly extend digital twin with new models upon their availability.
- *Harmonized APIs:* Unified APIs must be offered to the applications to query and browse ET, IT, and OT data from the corresponding data sources. These APIs must enable the applications to query different data without requiring familiarity with original APIs, data formats, and data models.
- *Data ownership:* Since different lifecycle data may be owned and managed by different organizations, it must be feasible for the organizations to decide on which data should be exposed, when, and with which extra semantic descriptions.

B. A Harmonized Information Meta-model for Digital Twins

In our use case, for example, the ET data is represented in the XML format, the IT data is represented in relational models, and the OT as OPC UA[13] models. The core part of our solution is a common information meta-model, whose core concepts are represented in Figure 3 as a UML class diagram.

The common information meta-model is based on the JSON format and complies with the object-oriented notions of types, object instances, and relations among the object instances. As the classes in the yellow color depict, each kind of lifecycle data is modeled using a dedicated type; the actual data is modeled as instances of the types. Each type of description consists of a set of properties. Besides, the type descriptions may contain tags to augment them with extra information.

We distinguish between two major kinds of lifecycle data: the one describing an entity along with its properties, and the other describing changing variables, events, or alarms associated with the entity. For example, a device has a set of fixed properties describing its technical characteristics, and it has operational variables that frequently change during its operation, as well as events and alarms that can be raised upon occurrence of a certain condition. The common information meta-model also enables defining variables, events and alarms.

Our common information meta-model allows for the representation of data source models in their original semantics, which may be proprietary or standardized, with the possibility for additional references to semantic standards, e.g. eCl@ss, as tags in the type descriptions. As a result, both proprietary and standardized semantics are supported by the information meta-model.

Regardless of whether the actual data is replicated within our platform or kept in original data sources, type descriptions are means for exposing the description (schema) of the data to the applications, to be used as the basis for querying the actual data. Since the original data is in a different format than our common information meta-model, e.g. relational vs. graphs of JSON objects, type descriptions are also a means for normalizing data models. For example, one-to-many relations are modeled in relational databases using foreign key relations, whereas, in a graph of JSON objects, we could embed such relations within one object.

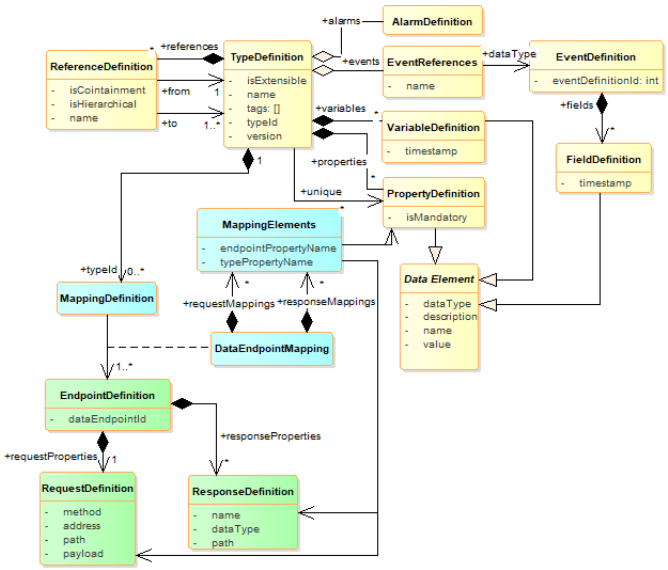

Fig. 3. The digital twin common information meta-model.

The actual data can be stored in our platform as instance models of the aforementioned types, or the data can be kept in the original data source and retrieved on-demand. If a model must be fetched from the external data sources, the model is marked as a 'shadow' object, meaning that it stores the necessary unique properties to distinguish it from other

models, but does not have the actual content for the remaining properties. Such a model is associated with the declarative endpoint and mapping descriptions used for fetching the data and mapping it to the model.

The classes in the green color in Figure 3 show the endpoint descriptions. Each endpoint description is designated with a unique identifier, and can have request and response properties.

Each communication with an endpoint has two sides: request and response. For the request part, the invocation method (currently HTTP/REST GET or POST), the address of the external server, the URL within that address, and payload are specified. For the response part, the object properties that are returned from that endpoint are defined in the endpoint description.

The response object from an endpoint may be different from the corresponding type description in our platform because the endpoint might be an already existing one that serves multiple other (legacy) applications. Therefore, our solution supports the so-called mapping descriptions, which specify how the response object properties and the corresponding type descriptions map to each other. The classes in blue color represent the elements of the mapping descriptions. A mapping description is bound to one type and can map the request/response properties of multiple endpoints to the properties of that type.

C. An Example Digital Twin

Figure 4 shows an example type and event definition. The type *abb.installedbase.MVDevice* is defined in our platform for the device *MVDevice*, which contains the list of device properties, as well as its customer and installation site as nested JSON objects. The type *abb.installedbase.MVDevice.ServiceEvent* defines necessary properties related to maintenance service events that occur on this device.

Figure 5 provides endpoint and mapping descriptions. Here, we would like to illustrate how one model in our platform, e.g. an instance of *abb.installedbase.MVDevice*, can be populated from multiple endpoints. We assume that there is a server that offers HTTP/REST GET endpoints to retrieve the device and site information. The endpoint description with the identifier *MVDevice* defines the endpoint from which device information is retrieved. This endpoint returns the list of device properties, as well its customer information as a nested JSON object within the device object. The expression indicates that the property *name* of the nested JSON object *customer* is represented via the internal property *customerName_internal* within our platform. This internal name is further used in mapping descriptions, as it will be explained later.

```
"customerName_internal": {
  "path":  "customer.name", //the response from the server
  "dataType":  "string"
}
```

Likewise, the endpoint description *Site* defines the endpoint from which site information can be retrieved.

Since we have two endpoint descriptions, the mapping also has two parts *MVDevice* and *Site*, which both map the

```
{
 "typeId": "abb.installedbase.MVDevice",
 "unique": [" serialNumber" ],
 "properties": {
   "serialNumber": { "dataType": "string" },
   "code": { "dataType": "string" },
   "site": {
           "name": { "dataType": "string"},
           "city": { "dataType": "string" }
    },
   "customer": {
           "name": { "dataType": "string"},
           "address": { "dataType": "string" }
    },
    ...
 },
 "events" : {
   "serviceEvents" : {
      "dataType" : "abb.installedbase.MVDevice.ServiceEvent"
    }
  }
}
```

```
{
 "eventDefinitionId": "abb.installedbase.MVDevice.ServiceEvent",
 "fields": {
    "serialNumber": {
       "dataType": "string",
       "timestamp" : "integer"
    },
    "maintenanceDate": {
       "dataType": "string",
       "timestamp" : "integer"
    },
     …
  }
}
```

Fig. 4. An example type definition.

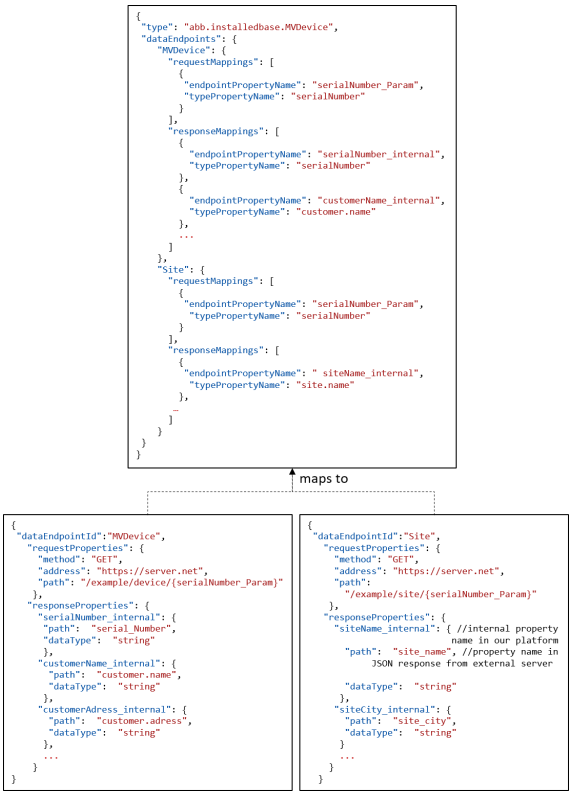

Fig. 5. An example endpoint and mapping description.

endpoints to the type *abb.installedbase.MVDevice*; i.e., two endpoints populate one model.

The mapping description has two major parts: *requestMappings* and *responseMappings*. The *requestMappings* defines

how properties in *abb.installedbase.MVDevice* are mapped to query parameters in the underlying endpoint. The *responseMappings* defines how the internal properties in the endpoint description are mapped to the properties of *abb.installedbase.MVDevice*. For example, the property *siteAddress_internal* is mapped to the property *name* of the nested JSON object *site*.

When an application queries for an instance of *abb.installedbase.MVDevice* with a specific serial number, all the endpoints that are referred to in the mapping descriptions are invoked to retrieve pieces of information and populate an instance of the *abb.installedbase.MVDevice* type.

As Figure 6 shows, in our solution the digital twin of a device is the entirety of its models; the models are typed and are linked together using a common identifier (i.e., *objectId* that is internal in our platform.

For each kind/category of industrial devices (e.g., drive, motor, sensor) and each kind of tool that is used for the selection of the device, there can be a separate type; here, we use the name *abb.selection.MVDevice* just an example. The model of the type *abb.engineering.MVDevice* describes the engineering properties of the device. The model of the type *abb.installedbase.MVDevice* describes the basic installed-base properties (e.g., technical properties, customer and site information) of the device.

The model of the type *abb.installedbase.MVDevice.ServiceEvent* is linked to *abb.installedbase.MVDevice*, and describes the list of maintenance service events that have been performed on the device. In contrary to other models, this model has event-based nature because a device may undergo multiple service events during its lifetime. The model of the type *abb.operational.MVDevice* describes the firmware parameters of the device.

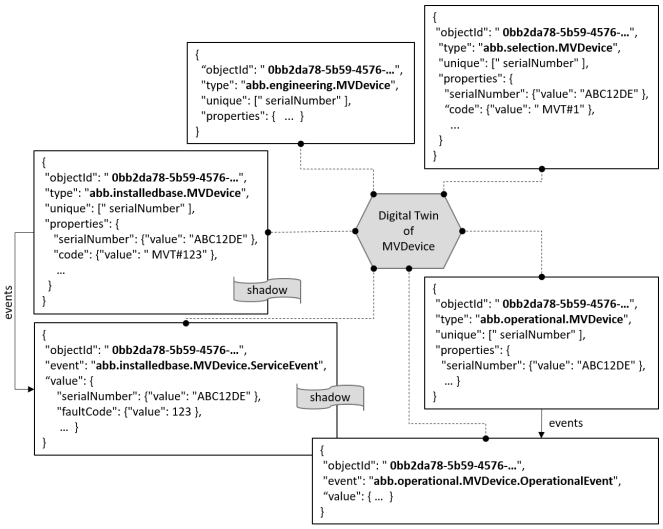

Fig. 6. An example digital twin.

It is in general the decision of domain experts and requirements of the use cases whether data must be stored within digital twins, or only a shadow reference to the data must be maintained. In our use case, since installed-based data may change more often during the lifetime of a device, the content of the installed-based models are fetched on-demand from the corresponding installed-based databases; hence, these models are associated with endpoint and mapping descriptions. The selection and engineering data are less prone to change and are stored within the platform. Likewise, the operational values are stored in the platform and are updated every time new values arrive.

D. A Digital Twin-centric Architecture and Platform

Figure 7 depicts the abstract architecture of our solution, which has been implemented based on Microsoft Azure services.

1) The Common Information Model Components: Type Storage is a MongoDB database in which type descriptions for digital twin models are stored; the component *Type Processor* is a microservice that interfaces this storage. *Object Storage* is a Cosmos DB database in which object instances are stored; the component *Object Processor* is a microservice that interfaces this storage. Both *Type Processor* and *Object Processor* offer dedicated REST APIs to applications for create, read, update and delete (CRUD) operations.

Event Storage is the telemetry storage based on Azure Timeseries Insight technology. The communication with devices takes place using AMQP or MQTT protocols and Azure IoT Hub service is used to facilitate the communication. If a device does not natively provide its OT data in our common information model, an industrial edge device can be used as the interface to translate the device OT data model to our common information model format.

The component *Event Ingestor* implements the functionality to receive the events and telemetry data from IoT Hub and to insert them in *Event Storage*. The component *Event Query Processor* interfaces *Event Storage* to enable accessing device events via REST APIs.

In addition to telemetry data, some other lifecycle data of devices also have event-based nature. For example, a device may have multiple maintenance events throughout its lifecycle. Such data can also be stored in *Event Storage* and be accessed via the same REST APIs.

2) The External Connectivity Components: The endpoint and mapping descriptions as explained in Section III-B. These descriptions are input to the component *Mapping Processor* via its CRUD REST API, which stores them in *Type Storage*.

The component *REST Communicator* provides the functionality for interpreting the endpoint and mapping descriptions to connect to the external data sources and map their responses back to the internal types defined in *Type Storage*.

Our solution currently supports HTTP/REST protocol for the communications with external data sources. Therefore, existing data sources should be extended to offer suitable RESTful APIs. The connectivity of existing tools to our platform is facilitated in the same way. Existing tools should be extended to import and export data from the platform. This

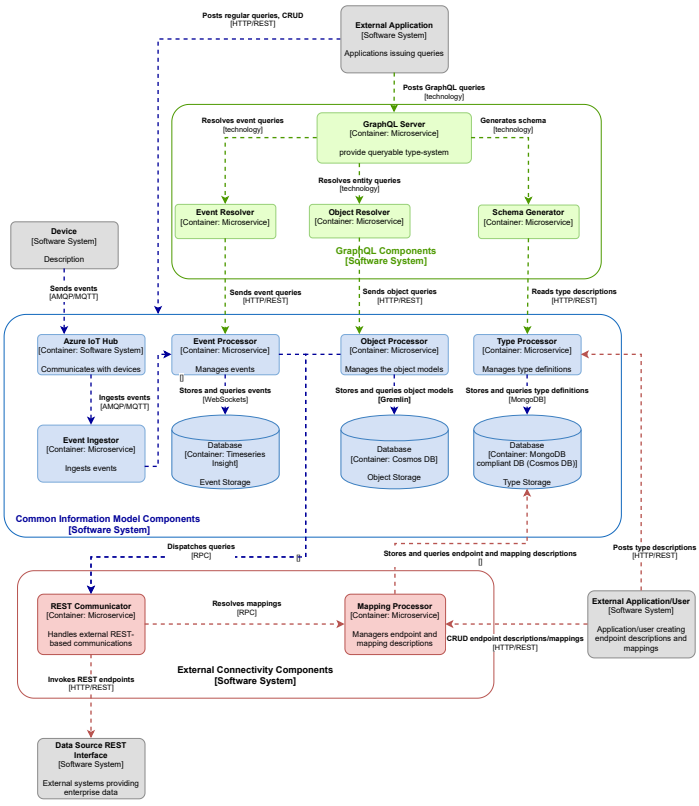

Fig. 7. Overall architecture of the digital twin-centric platform.

is facilitated via defining REST-based proxies for the tools, which must read the output of these tools, generate models based on our common information meta-model, and provide the models in the platform. Likewise, the proxies can enable importing data from the platform to the tools. The tools can also offer separate functionality to automatically generate type and mapping descriptions for each model.

Whether we adopt a pull-based or push-based approach to import/export models from/to the proxies depends on the nature of models and tools. In the push-based manner, the proxies use the APIs of our platform to upload the models in the platform. In the pull-based manner, the endpoint and mapping descriptions for the REST-based proxies must be provided, so that the platform can pull the models.

3) The Querying APIs: A key reason behind adopting digital twins is to make data access easier for applications. Our harmonized information meta-model is an enabler for offering harmonized APIs to access various lifecycle data. In our architecture, the components associated with different kinds of storage offer REST APIs to access the corresponding storage for the CRUD operations. Besides, the *Object Processor* and *Event Processor* components offer dedicated filtering APIs, which can respectively be used to query *Object Storage* and *Event Storage* for locally stored objects, as well as for fetching data that is external to the platform.

*a) **Querying Entity Data**:* The information stored in *Object Storage* do not have an event-based nature and can be queried using Boolean expressions defined on any property of the digital twin models. The properties that are used in the filter expression are decisive whether the query is resolved on the objects that are stored externally to the platform. To query the data that is stored externally, the filter expression must contain the type information in addition to the predicates on the desired property values. Consider the following example:

```
type = 'abb.MVDevice' and
(serialNumber = 'ASGHY4567' and
(code='MVT#155' or customer.name ='ABC'))
```

The type information is used by the *Object Processor* or *Event Processor* components to retrieve the corresponding endpoint and mapping descriptions, if any, and to dispatch the predicates further to the *REST Communicator* component. *REST Communicator* first based on the rules of Boolean algebra breaks the query to a set of conjunction (AND) queries, and then makes use of the mapping information to construct the following URLs. After the invocations to these URLs,the *REST Communicator* component unions the results of these queries and returns them to *Object Processor*.

```
GET https://server.net/example/device?serialN=ASGHY4567&customerName=ABC
GET https://server.net/example/device?serialN=ASGHY4567&code=MVT#155
```

*b) **Querying Event-based Data**:* Since there might be many occurrences of events, the *Event Processor* component offers a dedicated API to query event-based data over a time range. Additional calculations such as computing min, max and sum of event values are also supported. Below is an example query for fetching maintenance service events over two years. Naturally, the endpoint description should accept date parameters as well as the other parameters that are used in the filter expression. Such filtering expressions are processed in the same way as explained above for querying entity data.

```
{
    "date": {
        "from": "2018-08-20T17:13:36Z",
        "to": "2020-08-20T17:14:36Z"
    },
    "filter": "type = 'abb.installedbase.MVDevice.ServiceEvent'
              AND serialNumber ='ASGHY4567'"
}
```

*c) **Querying using GraphQL Expressions**:* GraphQL is a query language for APIs and a runtime for processing queries with existing data. GraphQL is known to reduce the required effort and the number of requests to fetch data, because instead of multiple REST requests only one request is required also for custom information demands [14].

The component *GraphQL Server* in Figure 7 offers GraphQL APIs and schema to the applications for querying. Under the hood, the *Object Resolver* and *Event Resolver* components resolve the queries that must be answered by *Object Processor* and *Event Processor*, and provide the results back to *GraphQL Server*.

To be able to answer GraphQL queries, *GraphQL Server* should offer GraphQL schema to the applications. This schema is generated automatically by the *Schema Generator* component, which reads the information about available types from *Type Processor* and accordingly generates a GraphQL schema by translating each type to a GraphQL type as depicted below.

Here, the type *abb_installedbase_MVDevice* is generated from the type *abb.installedbase.MVDevice*

whose instance is depicted in Figure 6; the type *abb_installedbase_MVDevice_ServiceEvent* is generated from the respective type in Figure 6. Since in Figure 6 there is a link from *abb.installedbase.MVDevice* to *abb.installedbase.MVDevice.ServiceEvent*, this link is represented here via the field *events* within the GraphQL type *abb_installedbase_MVDevice*. The arguments *from* and *to* in the field *events* are defined because our *Event Processor* requires querying events based on a time frame, and these arguments will be passed to *Event Processor* in later stages of query processing.

```
type abb_installedbase_MVDevice
{
  serialNumber: String
  code: String
  ...
  events (from: String, to: String): [abb_installedbase_MVDevice_ServiceEvent]
}

type abb_installedbase_MVDevice_ServiceEvent
{
  serialNumber: String
  date: String
  ...
}
type abb_installedbase_MVDevice_Search {
  serialNumber: String
  code: String
  ...
}
type Query{
  MVDevices : [abb_installedbase_MVDevice]
  MVDevice (serialNumber:String!) : abb_installedbase_MVDevice
  MVDevice (where:abb_installedbase_MVDevice_Search!) : [abb_installedbase_MVDevice]
  ...
}
```

As depicted in the type *Query*, three kinds of GraphQL queries are supported for each type: a) one for querying the entire list of objects, b) one for querying an object based on its unique properties such as *serialNumber*, and c) one for querying objects based on an arbitrary list of properties, which is defined as a dedicated type in GraphQL.

Below is an example query to select an *MVDevice* with the serial number *ASGHY4567*. The query filters two properties *serialNumber* and *code*, as well as the service events that occur in a specific time frame. This GraphQL query is then resolved as two queries; one issued to *Object Processor*, and one to *Event Processor* to fetch the device data as well as its service events, respectively.

```
{
   MVDevice(serialNumber:"ASGHY4567"){
      serialNumber
      code
      events (from:"2018-08-20T17:13:36Z",to: "2020-08-20T17:14:36Z"){
         date
      }
   }
}
```

IV. Discussion

Our solution enables realizing the *plug and provide* and *plug and consume* concepts; new data sources can be plugged in and their data become available to applications via providing endpoint and mapping descriptions as well as defining corresponding type descriptions within our platform; likewise, new applications can be added by keeping them agnostic of underlying data sources and formats.

Although our information meta-model is proprietary within the organization, it already helps to increase the interoperability across the lifecycle phases, as different data models can on the fly be mapped to this model and be augmented via extra semantic tags. These features, in addition to the offered unified APIs, increase the reusability, maintainability and interoperability of the applications if the data sources, data models and/or APIs change.

In our solution, each domain must establish the connectivity of its data sources to our platform, must map its models in our common information model format, and augment it with necessary semantic information. This enables realizing the notion of 'Data-as-a-Product'.

As we experienced via various use cases in ABB, our solution paves the way to establish company-wide data pipelines and digital twin models for various use cases; hence, preventing the 'digital twin solution silos' to occur. For m applications and n data sources the maximum communication paths in our solution will be $m+n$ (see Figure 2), as opposed to the maximum $m*n$ (see Figure 1) communication paths that may occur if no common data pipeline is in place. This reduces the development and maintenance costs of the applications, with the additional cost of developing and maintaining our proposed platform. The concrete assessment of the cost-saving requires an empirical assessment of our solution in practice over multiple years, which was out of the scope of this paper.

V. Related Work

Multiple classifications and surveys exist, which summarize the state-of-the-art on the topic of digital twins [3], [4], [5], [2]. These studies identify that a significant body of work focuses on the simulation aspects of digital twins for specific use cases in a specific lifecycle phase of devices. In [3] and [2] a long list of enabling technologies for digital twins are surveyed. The authors in [4] indicate the need for common architectures and platforms that offer means for defining digital twins especially by incorporating early lifecycle data, and enabling interoperable interactions among digital twins. We also believe that developing solutions silo for various digital twin-based use cases will not scale, and the need for platform-based approaches becomes more prominent.

In [15], an architecture for intelligent digital twins in cyber-physical systems is proposed. However, the details of how data ingestion from various sources takes place, as well as various APIs that must be offered to applications are not discussed. Alam and Saddik describe C2PS [16], a "digital twin architecture reference model" for cloud-based cyber-physical systems. Their focus is on network communication aspects and controller design. Gabor et al. [17] present the definition of an architectural framework for digital twins with a dedicated focus on the simulation of real assets.

In [18], a service-oriented application for knowledge navigation is presented. The architecture of the application linking different data sources is outlined briefly without mentioning the approach of how to link those data sources together. In [19], a digital twin platform based on a data-centric middleware is defined, whose architecture mostly focuses on communication and data transfer between the physical assets and simulation and not the data silo problem. In [20], a solution is proposed for sensor data integration and information fusion

to build digital-twins for cyber-physical manufacturing. In contrary to this solution, our proposal is not limited to specific sensor data, and also covers earlier lifecycle data that come from various sources in a push-based or pull-based manner.

In our previous work [8] we have defined a four-layer architecture for constructing and managing digital twins. In the follow-up publication [7], we defined a cloud-based solution for digital twins. In both papers, dedicated microservices had to be provided to fetch specific data from various data sources. This leads to solutions silos because for each use case, a dedicated set of microservices and data pipelines should be provided. Our proposal in this paper overcomes this problem by offering a common information meta-model, an open-ended architecture via declarative endpoint descriptions as well as a rich set of APIs.

Asset Administration Shell (AAS) [12] has been proposed as a digital twin for manufacturing systems. Since the specification of AAS is still under development, we experienced the standardization deadlock [10] problem in adopting AAS due to the late availability of its specifications, and its insufficient expression power for our use cases. For example, the current specifications of AAS do not cover events, various query APIs, nor the strategies to ingest backend data and map it to the AAS . As a result, companies may opt for a proprietary information model like ours to at least reach company-wide interoperability. Complementary to this, we proposed the notion of 'interoperability on-demand' [6], which means it should be possible to flexibly translate a (proprietary) digital twin model to any future standard such as AAS upon its availability. One may still adopt AAS as the meta-model for digital twins, and adopt our proposed solution for ingesting and mapping backend data as well as querying data.

When it comes to addressing the data silo problem, one may consider (cloud-based) data warehouse solutions such as Azure SQL Data Warehouse and/or Data lakes for Big Data [21] as a potential solution. We observe the following limitations to these solutions compared to ours: a) These solutions are data-centric and are widely adopted for (business) reporting and analytics use cases. However, there are many different kinds of Industrie 4.0 use cases that require a device-centric view on data, need connectivity of data models to physical devices and also must deal with the data silo problem across the device lifecycle [22]. Adopting separate solutions for reporting/analytics use cases leads to the solution silo problem. b) These solutions usually keep a copy of the data from data sources; hence, the original data sources are not the source of truth and one must deal with data inconsistency issues.

VI. Conclusion and Future Work

As the adoption of digital twins in industry increases, we believe that the need for software architectures and platform-centric solutions for digital twins to avoid the solution silo problem becomes even more prominent. This paper aimed to fill the observed research gap in this area by proposing a common information meta-model for digital twins, along with an open architecture and platform for creating and managing digital twins and ingesting backend data in them.

As future work, we would like to extend our solution further with various technologies listed in [2] for data collection, incorporating simulation and machine learning models, etc. for different use cases.

References

[1] Industrial Internet Consortium, "Digital Twins for Industrial Applications," 2020.

[2] Q. Qi, F. Tao, T. Hu, N. Anwer, A. Liu, Y. Wei, L. Wang, and A. Nee, "Enabling technologies and tools for digital twin," *Journal of Manufacturing Systems*, vol. 58, pp. 3–21, 2021.

[3] M. Liu, S. Fang, H. Dong, and C. Xu, "Review of digital twin about concepts, technologies, and industrial applications," *Journal of Manufacturing Systems*, vol. 58, pp. 346–361, 2021.

[4] 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.

[5] E. Negri, L. Fumagalli, and M. Macchi, *Procedia Manufacturing*, vol. 11, pp. 939 – 948, 2017.

[6] M. Platenius-Mohr, S. Malakuti, S. Grüner, J. Schmitt, and T. Goldschmidt, "File- and API-based interoperability of digital twins by model transformation: An iiot case study using asset administration shell," *Future Generation Computer Systems*, vol. 113, pp. 94 – 105, 2020.

[7] J. Schmitt, S. Malakuti, and S. Grüner, "Digital twins in practice: Cloud-based integrated lifecycle management," in *Automatisierungstechnik*, 2020.

[8] S. Malakuti, J. Schmitt, M. Platenius-Mohr, S. Grüner, R. Gitzel, and P. Bihani, "A four-layer architecture pattern for constructing and managing digital twins," in *Software Architecture*, 2019, pp. 231–246.

[9] T. G. Foundation, "GraphQL," https://graphql.org/.

[10] R. Drath and M. Barth, "Concept for managing multiple semantics with AutomationML – maturity level concept of semantic standardization," in *ETFA*. IEEE, 2012, pp. 1–8.

[11] R. Stark and T. Damerau, *Digital Twin*. Springer Berlin Heidelberg, 2019.

[12] Plattform Industrie 4.0, "Details of the asset administration shell – part 1," 2020.

[13] Int. Electrotechnical Commission, "IEC 62541-1: OPC Unified Architecture – Part 1: Overview and concepts," 2016.

[14] G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A practical assessment," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 140–150.

[15] B. A. Talkhestani, T. Jung, B. Lindemann, N. Sahlab, N. Jazdi, W. Schloegl, and M. Weyrich, "An architecture of an intelligent digital twin in a cyber-physical production system," *at - Automatisierungstechnik*, vol. 67, pp. 762 – 782, 2019.

[16] K. M. Alam and A. El Saddik, "C2PS: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE Access*, vol. 5, 2017.

[17] T. Gabor, L. Belzner, M. Kiermeier, M. T. Beck, and A. Neitz, "A simulation-based architecture for smart cyber-physical systems," in *ICAC*. IEEE, 2016, pp. 374–379.

[18] A. Padovano, F. Longo, L. Nicoletti, and G. Mirabelli, "A digital twin based service oriented application for a 4.0 knowledge navigation in the smart factory," *IFAC-PapersOnLine*, vol. 51, pp. 631–636, 2018.

[19] S. Yun, J. Park, and W. Kim, "Data-centric middleware based digital twin platform for dependable cyber-physical systems," in *Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2017.

[20] Y. Cai, B. Starly, P. Cohen, and Y.-S. Lee, "Sensor data and information fusion to construct digital-twins virtual machine tools for cyber-physical manufacturing," *Procedia Manufacturing*, vol. 10, pp. 1031–1042, 2017.

[21] A. Gorelik, *The Enterprise Big Data Lake: Delivering the Promise of Big Data and Data Science*. O'Reilly Media, 2019.

[22] Q. Qi and F. Tao, "Digital twin and big data towards smart manufacturing and industry 4.0: 360 degree comparison," *IEEE Access*, pp. 3585–3593, 2018.