Bringing Clouds Down to Earth: Modeling Arrowhead Deployments via Eclipse Vorto

Géza Kulcsár
IncQuery Labs Ltd.
Budapest, Hungary
geza.kulcsar@incquerylabs.com

Sven Erik Jeroschewski
Bosch.IO GmbH
Berlin, Germany
svenerik.jeroschewski@bosch.io

Kevin Olotu
Bosch.IO GmbH
Berlin, Germany
kevin.olotu@bosch.io

Johannes Kristan
Bosch.IO GmbH
Berlin, Germany
johannes.kristan@bosch.io

***Abstract*—The design and development of interconnected industrial production facilities, which integrate aspects of the Internet of Things (IoT) or, more specifically, the Industrial IoT (IIoT), often deals with complex scenarios involving dynamic System of Systems (SoS), resulting in immense development and deployment efforts. The Arrowhead community aims at delivering mechanisms and technologies to cope with such complex scenarios. In particular, the concept of local clouds constitutes a service-oriented architecture (SOA) framework for IIoT. Here, a central challenge is the conceptual modeling of such use-cases. SysML is widely established as a standardized modeling language and framework for large-scale systems engineering and, thus, for Arrowhead local cloud designs. However, SysML and its Arrowhead profile lack a canonical way to support actual platform modeling and device involvement in heavily distributed IIoT scenarios. The Eclipse Vorto project is ideal for filling this gap: it provides a modeling language for IoT devices, a set of modeling tools, and already existing reusable templates of device models. In this paper, we propose an approach to integrating Eclipse Vorto models into Arrowhead SysML models. We illustrate the concept with a realistic yet comprehensible industrial scenario and also present a prototype to emphasize the benefits of our novel integration platform.**

***Index Terms*—System Modeling, SysML, Eclipse Vorto, Eclipse Arrowhead, IoT, IIoT**

I. Introduction

Many IoT and, especially, *industrial* IoT (IIoT) scenarios introduce high complexity in all phases of their life cycle. Reasons for this are, among others, the use of multiple hardware and software platforms or heterogeneous protocols and data formats. With the recent trends of the increasing volume and complexity of such scenarios, it becomes more difficult and more expensive to model, operate, and manage such complex Systems of Systems (SoS) [1]. The *Arrowhead* initiative aims at overcoming these issues using a holistic, comprehensive methodology and mindset. One of the central facets of Arrowhead, also being highly relevant for the present paper, is the application of the concepts of Service-Oriented Architectures (SOA). In turn, Arrowhead introduces *local clouds* for service-providing and service-consuming system resources that can be grouped logically or geographically.

In turn, a novel kind of architectural and design challenges arises in this context of SoS design combined with dynamic, service-oriented orchestration principles, calling for new methods to cope with them. However, established techniques within *model-based systems engineering* (MBSE) [2] serve as a convenient baseline for introducing a new level of dynamicity for system (of system) architectures. MBSE arguably provides a great compromise between domain-specific expectations and rigorous design on the one hand, and a flexible, accessible modeling approach on the other hand. Besides, SysML is an excellent base for formulating and validating the well-formedness of complex systems.

Consequently, the Arrowhead approach to IIoT modeling relies on SysML for specifying local cloud architectures. A major challenge of such a modeling scenario is to integrate the abstract architecture models created in design-time with the actual IoT deployments, more precisely, with their digital twin representations. Naturally, the device-specific and deployment-specific details of these representations are out of scope in abstract SoS (local cloud) models. The very metaphor in the title of the paper, *bringing clouds down to earth*, unites two key points of this conceptual hiatus: (1) while those platform- and hardware-independent local cloud plans lack a connection to their future embodiment, and thus, still have to be brought down to earth, (2) such a device-oriented addition allows the expression of those real communication channels which have to be established between the systems taking part in a given cloud architecture.

As for existing approaches to IoT deployment modeling, VORTOLANG from the Eclipse Vorto[1] project is one of the most relevant and prevalent examples. Other benefits of Eclipse Vorto are that it allows the reuse of existing models through shared repositories and provides plugins for code generation to integrate with other projects and platforms. As a consequence, it seems promising to use Eclipse Vorto to ease the modeling of local clouds by mapping those resources into SysML models. In this paper, we investigate this approach further. In particular, the main contribution of this work, with potential industrial relevance, is an integration concept for coping with the design complexity of industry-scale IoT installations through an integrated modeling approach. As an additional benefit, a baseline arises for a bilateral cross-fertilization: we extend Eclipse Vorto towards functional and architectural design, and systems modeling towards detailed device platform specifications.

Therefore, we first introduce the Eclipse Vorto project and

[1] https://www.eclipse.org/vorto/

its VORTOLANG in Sect. II and give an example use case in Sect. III to which we later apply our modeling approach. Sect. IV introduces the Arrowhead Framework while Sect. V presents the current model-based system engineering process in the Arrowhead Framework with SysML. In Sect. VI, we explain our approach for a mapping between SysML and Eclipse Vorto. This mapping is then illustrated in Sect. VII.

II. ECLIPSE VORTO

Eclipse Vorto [3] is an open-source project for modeling digital twins. The goal of the project is to provide platform-independent and vendor-neutral tools to model digital twins and to make use of the models by supplying plugins to ease the integration in existing IoT ecosystems. The project consists of 4 main components:

- VORTOLANG: a domain specific language (DSL) to describe digital twins
- Repository: a repository to create, manage and distribute models
- Plugins: transform Vorto models into source code, request templates or other representations
- Telemetry Payload Mapping: maps the telemetry data sent by a device using a mapping specification based on a Vorto model

A. VORTOLANG - *The Vorto Language*

VORTOLANG is the domain specific language used to describe digital twins. It consists of four different kinds of elements:

- Information Model (IM): describes a digital twin and its capabilities
- Function Block (FB): describes a set of capabilities that are implemented by the digital twin. Function Blocks can be designed hierarchically by extending other Function Blocks. The individual capabilities are grouped into the following property groups:
  - Status: contains properties of the digital twin that can only be read (read-only)
  - Configuration: contains properties of the digital twin that can be both read and set (read-write)
  - Event: contains events that can be emitted by the digital twin
  - Operation: contains functions that can be invoked on the digital twin
  - Fault: contains fault states that can occur on the digital twin
- Data Type (DT): describe complex data types or enumerations that can be assigned to Function Block properties
- Mapping: describes platform-specific or implementation-specific information that can be added to a generic Information Model or Function Block.

B. *Vorto Repository*

The committers of Eclipse Vorto host an official public instance of the repository[2]. The official repository is an offering for device manufacturers and IoT solution developers to develop and publish re-usable models of their devices / digital twins in a standardized way. However, it is also possible to self-host a Vorto repository (e.g. for on-premise solutions without internet access). The repository offers several features to interact with the Vorto models:

- UI and APIs to interact with the repository and the models
- A web editor to create and edit models
- A review and release process for models
- Different levels of visibility (private / public)
- Import and export functionality of models
- Direct integration with Vorto plugins
- Java client that can interact directly with the APIs of the repository

[2]https://vorto.eclipse.org

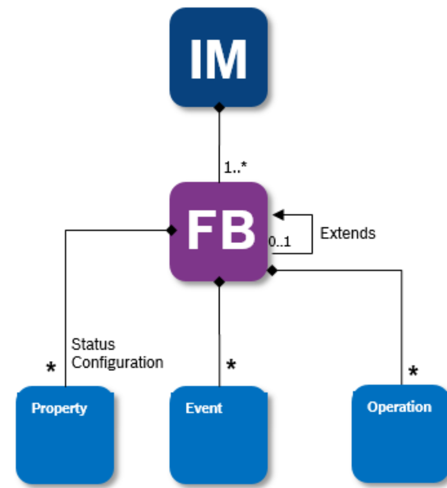

Fig. 1. A simplified model of VORTOLANG

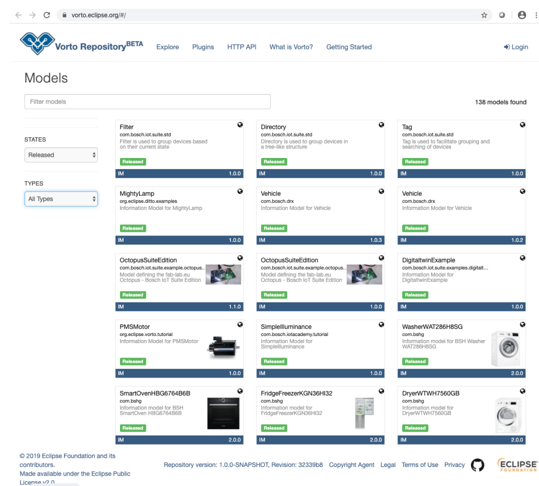

Fig. 2. Screenshot of the landing page of the public Eclipse Vorto repository (https://vorto.eclipse.org/#/, accessed 04.08.2020)

C. Vorto Plugins

Vorto Plugins can be used to process Vorto models to transform them into different formats, representations, source code, and request templates. Currently, there are five officially supported plugins:

- Eclipse Hono Plugin: transforms Vorto models into source code to connect devices to Eclipse Hono via MQTT
- Eclipse Ditto Plugin: transforms Vorto models into Eclipse Ditto Digital Twin representations (JSON or OpenAPI)
- JSON Schema Plugin: transforms Vorto models into a JSON Schema representation
- OpenAPI Plugin: transforms Vorto models into an OpenAPI YML representation
- Bosch IoT Suite Plugin: transforms Vorto models into source code to connect to the Bosch IoT Suite or into a request template to provision devices

In addition to the officially supported plugins, several experimental plugins offer other transformations. Experimental plugins are managed in a separate Github repository.[3] All plugins can be used either as local run applications or as AWS Lambda functions. The public Vorto repository is integrated with the official plugins as AWS Lambda functions and can thus be used directly from the Vorto Repository UI. One can also use the plugin API to develop custom plugins.

D. Vorto Telemetry Payload Mapping

The Vorto Telemetry Payload Mapping engine is a standalone application to map telemetry data that is sent by a device. To use the mapping application, one needs to create a payload mapping configuration, to understand the source format used by the device and the desired normalized target format. The payload mapping engine offers a canonical JSON target format and the Eclipse Ditto JSON format. The normalized payload data can then be used to build applications with normalized APIs based on the Vorto models.

III. Example Models

In the following, we define an artificial example use case to showcase the capabilities of the Eclipse Vorto models. Later in this paper, we use these models for an example mapping of the Vorto meta-model to an Arrowhead SysML Profile. Figure 3 gives an overview of the use case. In the depicted setup, we assume a production facility with several units like conveyors and an assembly robot. A server back end system allows the collection of production data by providing connectivity, a digital twin device abstraction, and storage e.g. through instances of Eclipse Hono[4], Eclipse Ditto[5] and a database. The back-end server also hosts Arrowhead core services to provide the infrastructure for a local cloud with the mentioned machines and software systems. The objective of the use case is to collect and centrally store data from the various units and to perform basic control operations between the machines.

[3]https://github.com/eclipse/vorto-examples/tree/master/vorto-generators

[4]https://www.eclipse.org/hono/

[5]https://www.eclipse.org/ditto/

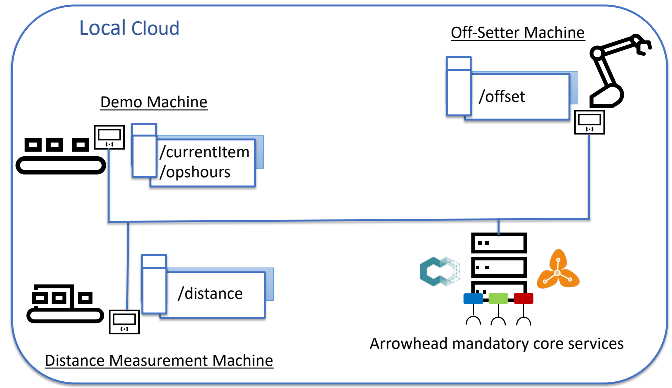

Fig. 3. Demo Setup

The local cloud has three machines. One of them is a demo machine for which we created a custom Vorto model. The starting point of the modeling approach is the information model in Listing 1.

Listing 1. Information model for demo machine

```
1  vortolang 1.0
2
3  namespace org.arrowhead.demo
4  version 1.0.0
5  displayname "demo-machine"
6  using org.arrowhead.demo.CurrentItem ; 1.0.0
7  using org.arrowhead.demo.OpsHours ; 1.0.0
8
9  infomodel DemoMachine {
10
11   functionblocks {
12     currentItem as CurrentItem
13     opsHours as OpsHours
14   }
15 }
```

In our case, this machine tracks its operation hours and which item it currently processes. For both aspects, we defined separate function blocks. Listing 2 shows the function block for the operational hours. Here the assumption is that the machine tracks it operational hours and produces events when it reaches a maintenance window (line 19 to 21) or requires that it gets moved (line 15 to 18) depending on the operation time. We assume that one needs to move the demo machine to avoid it from wearing out when operating in the same position for too long. The operator can further set the duration of the maintenance window and the window until the next movement, which we model with the configuration block in lines 9 to 12.

Listing 2. Function block for operation hours

```
1  vortolang 1.0
2
3  namespace com.eclipse.arrowhead
4  version 1.0.0
5  displayname "Operation Hours"
6  description "Operating hours"
7
8  functionblock OpsHours {
9    configuration {
```

```
10      turnWindow as long
11      mandatory largeMaintanceWindows as long
12    }

14    events {
15      moveTimeReached {
16        mandatory timestamp as long
17        mandatory offset as long
18      }
19      maintenanceReached {
20        mandatory timestamp as long
21      }
22    }
23  }
```

Then there is also a function block for the currently processed item depicted in Listing 3. Here the item is identified by an id. Since one can only read but not write this value from the item we modeled this as status.

Listing 3. Function block for the currently processed item

```
1  vortolang 1.0

3  namespace com.eclipse.arrowhead
4  version 1.0.0
5  displayname "Current Item"

7  functionblock CurrentItem {
8    status {
9      mandatory id as string
10   }
11 }
```

Another machine can move the first machine to a given offset. The capability of this off-setter machine can be modeled as an operation in a function block corresponding to Listing 4. We do not show the information model for the second machine because it has strong similarities with the information model of the demo machine.

Listing 4. Model for offset movement

```
1  vortolang 1.0

3  namespace com.eclipse.arrowhead
4  version 1.0.0
5  displayname "Offset Movement"

7  functionblock Offset {
8    operations {
9      moveToOffset(offset as long)
10   }
11 }
```

One had to describe the already mentioned machines in particular models. However, as shown in Figure 6 the modeled Arrowhead local cloud shall also contain a third machine, which measures distances e.g. between objects on a conveyor. For this machine, it is possible to reuse the distance sensor model obtainable from the public Vorto repository [4] and thus integrate the distance measurement machine into the model of the local cloud with minimal additional effort. This integration also highlights the benefit of having a central source for more or less generic models to foster reuse and adoption of existing models to decrease overall engineering overhead.

IV. Eclipse Arrowhead

Eclipse Arrowhead [6] is a newly founded open-source project in incubation at the Eclipse Foundation, which offers methods and tools to bring concepts of service orientation to the Industrial Internet of Things. The *Arrowhead* initiative has a strong focus on fostering interoperability via service and interface descriptions between systems and components [5]. The Arrowhead community originated from a joint European effort of more than 80 industrial and academical partners to bridge the interoperability gaps for applications and tools in IoT-based automated industrial scenarios. Currently, there are multiple projects following up on the promising results, the two most important, large-scale consortial endeavors being Productive 4.0 and Arrowhead Tools. Productive 4.0 explicitly aims at putting Arrowhead concepts into industrial production in the context of Industry 4.0 [6]. In contrast, the goal of the *Arrowhead Tools* project, started in 2019, is to establish a mature software and tooling landscape around the Arrowhead core to foster even broader and more efficient adoption of Arrowhead technology in the industry. It is in the context of Arrowhead Tools that the initially proposed Arrowhead Framework has started its journey in the Eclipse universe.

As for its principles, the whole Arrowhead ecosystem bases on a *service-oriented architecture* (SOA) [7]. However, in contrast to classical SOA, the Arrowhead Framework does not explicitly employ an enterprise service bus (ESB) as a central messaging point. Instead, it uses service-to-service communication as proposed for micro-services instead. The Arrowhead Framework introduces the concept of local clouds, which encapsulate geographically connected processes, such as production facilities. Delsing et al. [5] define five substantial requirements, which local clouds have to meet: (i) Low latency guarantee for real-time use cases; (ii) a high degree of scalability of automation systems; (iii) multi-stakeholder integration and operations agility; (iv) security and safety measures; and (v) ease of application engineering. As long as the listed criteria are met, the Arrowhead local cloud concept does not define the underlying architecture. However, especially latency and security requirements, depending on how important they are for the respective use-case, might require a complete IoT setup involving *edge deployments* [5]. The local clouds contain all necessary components to operate on their own. Generally, each local cloud consists of three kinds of entities:

1) *Devices* are the hardware foundation of each local cloud and are hosts to one or multiple systems. A device is not bound to a specific performance threshold. Hence, small and constrained hardware can be part of the local cloud, as well as more powerful machines.
2) *Systems* are the software artifacts executed on the underlying devices, forming a logical unit of semantically coherent tasks. These systems autonomously register themselves and their provided services at a service registry. Besides service provision, a system is also capable of consuming services of other systems.

[6] https://projects.eclipse.org/projects/iot.arrowhead

3) *Services* are functional representations of systems towards the outside world. They are the primary artifacts in connecting services according to SOA principles: *provided* services get *consumed* by other systems (which might, e.g., depend on specific inputs for their operation). There is no technical specification concerning the choice of protocol or payload format. Since this is part of the interface definition of each service and beyond the scope of the Arrowhead specification. For instance, the system might employ web technology or broker-based communication patterns.

In the following section, we turn ourselves to a founded approach of capturing such local cloud architectures via systems modeling techniques.

V. Modeling a Cloud: Arrowhead Tools and Systems Modeling

The notion of model-based systems engineering (MBSE) plays an important, even crucial role in holistic engineering workflows involving large-scale, complex, dynamic systems [2]. However, MBSE and its primary modeling language, SysML [8] are arguably recognized for capturing monolithic systems with a more fixed (though probably complex) architecture. Recently, there has been a growing interest around the best ways to employ systems modeling in modern, dynamic, even cloud-based scenarios.

As for modeling Arrowhead local clouds, we rely on the aforementioned established systems modeling language and methodology, SysML. SysML is a dialect of the well-known Unified Modeling Language (UML), tailored to meet the specific needs of systems engineering activities. In turn, modeling Arrowhead local clouds requires a custom-tailored approach with a considerable amount of flexibility to adequately capture the diverse set of entities as introduced above. SysML is the canonical language of choice for such endeavors. Also, SysML excels at language extensibility (being a primary concern) and has mature, feature-rich tooling and an active community. The language provides several different diagram types to represent the facets of the system to be modeled, from requirements to static structures to communication protocols. For further general details, there is a large variety of textbooks available — e.g., for practical information of SysML, refer to the comprehensive book of Friedenthal et al. [8].

Recently, there has been a proposal for a concrete, Arrowhead-centered approach for modeling service-oriented applications, i.e., Arrowhead *Systems of Systems* [9], extending and refining the Arrowhead documentation approach proposed earlier [10]. The solution is a standard UML/SysML mechanism to enlarge and tailor the modeling language for a specific domain. In short, a profile is an organized collection of *stereotypes*, i.e., domain-oriented specializations of generic SysML language concepts. We refer the interested reader to [9]; here, we focus on those parts which have direct relevance to the present integration approach:

The so-called *interface design descriptions* (IDD) can be conceived as the realization blueprints for certain services on certain systems (cf. Sect. IV). In particular, IDDs contain operation signatures representing service functionality. A single IDD can be used as a service "type" for modeling both provider and consumer behavior.

A central Arrowhead notion is that of *devices*; thus, there is a corresponding stereotype, serving as a mere placeholder (better said, a canonical integration point) in the original version of the profile. The present paper is, in fact, an actualization of such an integration, which results in filling that stereotype with life and details.

A local cloud configuration is modeled via *deployed entitites*, represented as specialized SysML *part properties*. This part configuration is the place where platform modeling gets realized on a (conceptual) deployment level—the next section demonstrates this through examples.

This profile and modeling approach is referred to as SoSysML in the rest of the paper. Figure 4 shows an overview of the SoSysML representation of our example local cloud (cf. Sect. III); in particular, the upper row of the diagram consists of *system design descriptions* (SysDDs), representing design templates for the three system kinds involved in the use-case. SysDDs also play a significant role as the hosts for actual interfaces. This is materialized by their *ports* (the small rectangles at the edge of the SysDD boxes). The essential logical structure of SoSysML lies here: SysDDs are brought together with IDDs (abstract service and interface descriptions not explicitly depicted here) by *using IDDs as types of ports on SysDDs*. Thus, a SysDD represents an object (a system) while its ports express its behavior (via the typing IDDs). Details on IDDs are out of scope here—for the present paper, it suffices to conceive of them as operation collections.

The bottom row, in turn, contains the SysML representations of those *device kinds*, whose *instances* the actual system *instances* will be allocated to. These device templates come in two fashions: in some cases like the distance sensor in our running example, an already modeled device can be readily used and, thus, directly imported from the Eclipse Vorto repository, while in other cases, the design process might necessitate the modeling of new devices. The next section covers details of this instantiation and allocation.

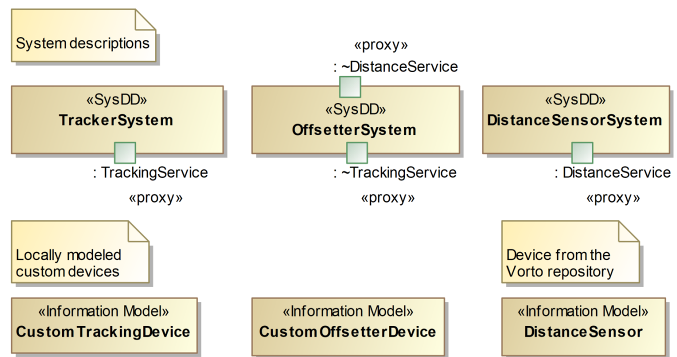

Fig. 4. SoSysML Overview: The Smart Assembly Use-Case (Sect. III)

Note that the above figure is an excerpt from an actual

model realization as available in MagicDraw, an industrially established and widely used systems modeling tool. Thus, the integration presented as the essential contribution of this paper in the following Section VI also reveals an industrial potential w.r.t. base technologies involved here: it seems that while IoT device/deployment models are addressed in the Eclipse ecosystems, abstract systems modeling and platform-independent design have a home in the NoMagic[7] (MagicDraw, Cameo Systems Modeler, Teamwork Cloud, ...) tool infrastructure. The prototype serving as the practical baseline for the present contribution constitutes an important step towards integrating these two platforms/ecosystems (having some conceptual touching points which we can rely on) and, thus, we find that this "mismatch" is more a benefit than an impediment. Moreover, the underlying implementation of our MagicDraw plugin is based on VIATRA, an established Eclipse model transformation engine.[8] We will continue to build upon this combined ground in the future. As for the future potential of the current approach, we also remark that this approach is very likely to play a leading role in the future of systems modeling as well. Currently, the upcoming new major release of the systems modeling standard, SysML v2, considers the Arrowhead SoS profile mentioned as a candidate for SOA modeling standardization.

VI. Vorto, SysML, Arrowhead: The Integration Approach

As a culmination and summary of the ideas introduced above, we turn ourselves to the primary concept of this paper: the integration of Eclipse Vorto device models with SoSysML, i.e., our Arrowhead-specific SysML-based design approach to device-independent SoS (local cloud) modeling.

On a conceptual level, Table I summarizes the essence of the integration approach. As it has already been hinted at in the previous section, the main observation behind the integration is a direct correspondence, i.e., a mapping, between concepts from SoSysML and Vortolang. In particular, *Device* in SoSysML serves as a topmost container for any device descriptions, thus corresponding to the topmost Vortolang elements, *Information Models*. The IDDs correspond to the actual functional specifications, the *Function Blocks* of Vortolang. This concise table also indicates that below this level, the two modeling languages become equivalent: their *Operation* concept represents the same abstraction level.

TABLE I
SoSysML to Vortolang concept mapping

SoSysML	**Vortolang**
IDD	Function Block
Device	Information Model
Operation	Operation

The essence of our contribution, i.e., the actual integration of SoSysML local cloud models and Vorto digital twins is illustrated through our example in Fig. 5. The upper region above the horizontal dashed line (*Deployed systems*) is the way to organize non-allocated, abstract, platform-independent system instances into a local cloud in SoSysML: each box represents a system instance to be deployed, where the string before the colon is its proper name, while the string after the colon indicates its type, which is, in turn, a SysDD (cf. Fig. 4). This typing provides each system with an interface (port) structure. The structure is turned into an abstract, design-time local cloud representation by the connectors (solid lines) attaching provided interfaces to consumed interfaces (the latter denoted by the same type name depicted with a tilde prefix). Notice the resemblance of this part of the figure to the intuitive scenario depiction in Fig. 3.

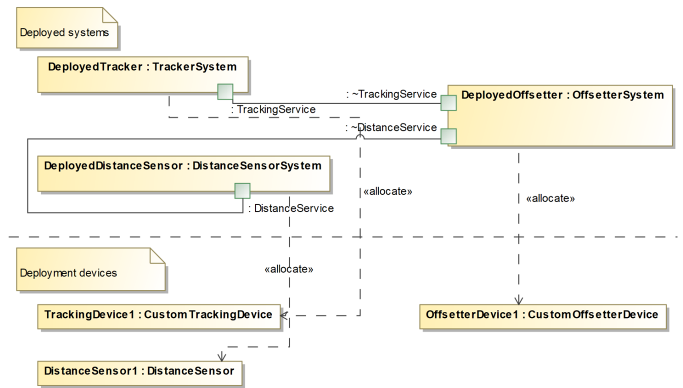

Fig. 5. Example Use-Case: SoSysML-Vorto Integration

The extended, integrated local cloud representation (including the lower region of Fig. 5) now contains Vorto devices as well. These are represented, again, by boxes with typed instances (using the same label convention)—but here, the types come from a set of Vorto device descriptions (represented as *Information Models*, cf. Table I). In turn, these information models might originate from two different sources (cf. also the bottom line of Fig. 4), both considered by our *MagicDraw Vorto Importer* module:

they either directly come from the Vorto repository itself; to this end, the importer browses the online catalog and the user can automatically create a SysML representation of such information models, if she or he thinks that one of them would fit their design (the case of the DistanceSensor in our example); or

if the desired functionality is not yet represented by any "stock" solution, the user can choose to write his or her own Vorto specification with any preferred local workflow conforming to Vortolang. For example, they can use the Vorto IDE or the web modeler, and might even reuse already predefined function blocks.

Based on the Vorto models users may also use one of the various code generators and plugins to generate integration code for different back-end systems. We implemented both of the aforementioned model input sources in our implementation of the *MagicDraw Vorto Importer*, a plugin of the established

[7]https://www.nomagic.com/
[8]https://www.eclipse.org/viatra/

systems modeling tool MagicDraw[9] (or its SysML-equipped distribution Cameo Systems Modeler, the difference being immaterial for our purposes). The prototype features a user-friendly manner, where the built-in *Import* menu is extended by two further options one for browsing the Vorto repository and choosing information models to import, the others to browse locally for user-created Vorto model packages. From this point on, the usage of the imported information models is the same regardless of the source, as detailed above.

VII. Applying the Integration Approach

In the previous section, we presented our integration approach. Now we demonstrate its use in an engineering workflow using our running example from Sect. III. As our metaphor in the title suggests, our integration approach fills SoSysML models of Arrowhead local clouds with device details, i.e., relates the cloud model with the bare metal devices (or their abstractions).

To realize the scenario described in Sect. III, the following steps have to be carried out.

a) Model System of Systems Model: Here the initial Model of the SoS scenario, i.e. the Arrowhead local cloud, has to be modeled. In the example case depicted in Fig. 6, one has to create a SysML model for the three Systems and their relationships, as shown in Fig. 4. This step is not unique to our mapping approach but is more a prerequisite.

b) Select Devices in Repository: With the SoSysML model available, the modeler can turn to fill the modeled abstract devices with life. As the Vorto repository already contains a considerable number of different models, the first step should be to browse it and check if an information model of the required device already exists. In our example, this is the case for the distance sensor modeled for machine three.

c) Model Device: If a model does not already exist in the repository, the modeler can still use VortoLang to model the device capabilities. In our example, this is done with the devices described in Listings 1–4. Those models can be imported into the SoSysML as well. Using VortoLang has the advantage that the whole infrastructure provided by the Eclipse Vorto project can still be applied. For example, one can integrate existing function blocks into an information model. It is then possible to publish the models later on in the Vorto repository if they are stable and of broader use.

d) Import Device Models: With the device models at hand, the modeler can now import the Information models into the SoSysML models (upper part of Fig 6). The device models provide the SoSysML model with the concrete capabilities of the devices, which in turn allows for fine-grained modeling of processes and procedures. An example is the conveyor maintenance procedure described in Sec. III.

With that step, the actual integration of Eclipse Vorto and SysML is complete but there are some further steps possible.

e) Generate Connectivity Code: As the device models are available as Vorto information models also the code generators of Eclipse Vorto can be used to generate the connectivity code (lower part of Fig 6). In our example, the machines connect to an Eclipse Hono instance. So we need code for the connectivity with Eclipse Hono. The Vorto generator plugin for Hono directly generates connectivity code stubs in C (Arduino), Java, and Python that can be integrated into the connectivity stack of the machine or an attached gateway.

f) Perform Arrowhead Wiring: As the SoSysML model has all the required data, the actual Arrowhead wiring, i.e., registration of the services at the Service Registry and configuration of the Arrowhead Orchestrator can happen semi-automatically utilizing another plugin.[10]

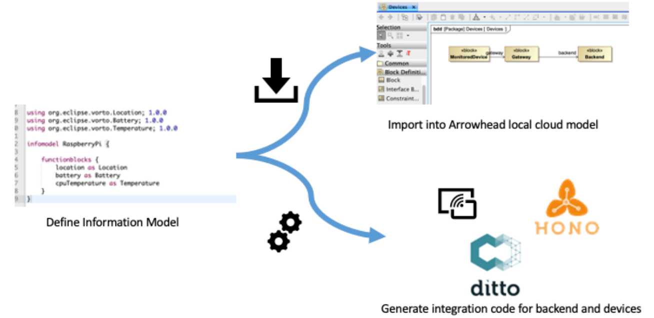

Fig. 6. Applying Vorto models in an engineering process

VIII. Related Work

The integration approach pursued in the paper touches on various industrial concepts and frameworks of high relevance; however, due to the rising interest around such topics, it is not typical in IIoT and Industry 4.0 to have widely established industrial solutions. We perceive an abundance of often non-public domain-specific architectural solutions. The EU has started to establish an industrial reference architecture framework, RAMI4.0 [11], which represents a much higher abstraction than our present contribution, and a direct comparison is, therefore, out of scope here. We mention some related approaches in the most relevant fields in the following.

First, service-oriented architectures already have a standard modeling language called *SoaML* [12] which is also a dialect of UML (just as SysML). The concepts which SoaML relies on are more orthogonal to the SysML-based design pursued here. However, our flexible setup allows an integration of various further diagram and representation types, even SoaML. SoaML also lacks a device modeling aspect and one therefore would have to refactor the Vorto integration in that case too. For an overview of other SOA standards, refer to [13].

The topic of *platform modeling* and the clear distinction between a platform-independent and a platform-specific model (PIM/PSM) is a fundamental concept in the OMG standard

[9]https://www.nomagic.com/products/magicdraw

[10]https://github.com/IncQueryLabs/arrowhead-tools