Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Domain Model-Based Data Stream Validation for IoT Applications

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Simon Pizonka

geboren am: 26.07.1987

geboren in: Essen

Gutachter/innen: Prof. Dr. Timo Kehrer

Prof. Dr. Matthias Weidlich

eingereicht am: ..... verteidigt am: .....

Contents

List of Figures

List of Tables

List of Listings

Abstract

With ongoing digitalisation, more and more devices are connected to the Internet. A lot of these devices are online 24/7 and sending continuous data. Such devices can be in the field for multiple years and, over this period, they must work reliably. This includes not only the hardware but also the processing and storage of data. In order to achieve this, it is necessary to continuously monitor incoming data. This thesis presents an approach as to how a domain model can be used for stream data validation. The used model allows for the definition of properties and functions of a device. These include defining constraints e.g. limiting the range of a value. The developed prototype allows loading this model in a stream processing system and validating structure and constraints in an unbound data stream. In a case study, the presented approach is tested on its ability to detect errors in an existing data set.

Zusammenfassung

Mit der fortschreitenden Digitalisierung werden immer mehr Geräte mit dem Internet verbunden. Viele dieser Geräte sind rund um die Uhr online und senden fortlaufend Daten. Solche Geräte können mehrere Jahre im Einsatz sein und müssen über diesen Zeitraum zuverlässig funktionieren. Dies betrifft nicht nur die Hardware, sondern auch die Verarbeitung und Speicherung von Daten. Um dies zu erreichen, ist es notwendig, eingehende Daten kontinuierlich zu überwachen. Diese Arbeit stellt einen Ansatz vor, wie ein Domänenmodell für die Datenstromvalidierung verwendet werden kann. Das verwendete Modell ermöglicht die Definition von Eigenschaften und Funktionen eines Gerätes. Dazu gehört das Definieren von Beschränkungen, z.B. die Begrenzung von Wertebereichen. Der entwickelte Prototyp ermöglicht es, dieses Modell in ein Stream-Verarbeitungssystem zu laden und Struktur und Einschränkungen in einem konstanten Datenstrom zu validieren. In einer Fallstudie wird der vorgestellte Ansatz auf seine Fähigkeit getestet, Fehler in einem bestehenden Datensatz zu erkennen.

1 Introduction

The term Internet of Things (IoT) is omnipresent today and is an ongoing trend. There are several forecasts for how many IoT devices we will have in the future. One frequently cited source is the industry analyst company, Gartner Inc., and they expect that there will be around 20.4 billion IoT devices by 2020 [1].
All these devices will be connected to the internet and will send a continuous stream of data, which results in an increased amount of data to process and store. Often devices are in the field for multiple years and, over this period, their reliability and function must be guaranteed, not only for the hardware, but also for processing and storage. This is why it is necessary to continue to validate incoming streaming data and take necessary measures if something is wrong. To achieve this, a processing step can be included to send an alert if an error within the data is detected. To implement such a function, a deep understanding of the processed data is required, and the structure and format of the data must be clear. Often it is not the same person who designs and programs the hardware and then implements the cloud application. For this reason, it is important to agree in advance on the message structure and create documentation. With each change of the structure, it is necessary to update multiple things manually. Besides the code, this can also be the documentation of the system. To synchronise all this work can be a challenging task, which is quite usual for client/server applications. A popular framework for Application Programming Interfaces (APIs) is Swagger[1]. This framework helps to describe and generate code for RESTful web services. It achieves this by providing a documentation format decoupled from any programming language. This format can be later used, with the help of a generator, to generate code for the client and server.
Eclipse Vorto[2] tries to achieve this in a similar manner to Swagger but with a strong focus on IoT devices. Quite often, a device has a permanent open connection and, over this connection, a device can receive and send messages. With the help of Vorto, these messages are defined. Such a message can be something like a command or status. To achieve this, a Domain-Specific Language (DSL) is provided to describe the functions and capabilities of real-world devices. The project contains different generators to produce code for a device to connect to an IoT platform and transfer messages. Such a model not only provides the data structure of a message, it can also contain constraints and other meta-information about the device. Those constraints can be the measurement range of a sensor.

[1] `https://www.swagger.io/`
[2] `https://www.eclipse.org/vorto/`

This model could now be used in a stream processing system to autonomously validate an incoming data stream. The structure and constraints can be validated. Rules to identify violations of the constraints can be derived directly from the model. Such a model can serve several purposes in this case, without the requirement of having a deeper understanding of the different parts, as those parts are automatically generated. This model can easily be reused or extended. Furthermore, a deeper understanding of the stream processing is not required.
Besides boundaries, a model can contain further information. Typically, in a data sheet of a sensor, multiple characteristics are described. These can be boundaries, precision class, maximum error rate, etc. These can be transferred to a model, without much effort, to achieve a first level of data validation.

1.1 Motivation

Software development can be a complex and time-consuming process. With a rising amount of software developers, the speed does not necessarily proportionally rise, too. The more people are involved, the higher the requirement of coordination can be, especially for IoT applications where different domain skills are required. Here, a model-driven approach could be helpful. A domain model can be the basis for sharing common knowledge. Such a model also enables the processing of this knowledge in an automated manner.
IoT devices, for example, often produce continuous streams of data. The amount of data can be overwhelming and it is possible that problems are found too late. This is because the analysis and validation are often manual processes. One of the first steps is to clean the data of corrupt or inaccurate values. However, this knowledge is not always transferred back to the processing pipeline due to lack in awareness or communication. As a result, corrupt data would be collected and stored, resulting in continuous costs. With a domain model-based rule checking system, corrupt or incorrect data can be filtered before storage. They can be stored in a separate database for later analysis. If corrupt or invalid data can be detected at an early stage in the processing pipeline, it can save costs and potentially increase the reliability of the system.

1.2 Problem and Scope

The aim of this thesis is to evaluate how a domain model can be included in a stream processing system for data validation and what the benefits of this are. There are many

existing sophisticated approaches for detecting anomalies in data streams. Some of them use domain knowledge to achieve this, but they typically do not use domain models, especially in combination with a stream processing system.
There could be multiple advantages to this approach. A Model-Driven Development (MDD) approach can help to allow better testing of parameters and make potential errors visible. Additionally, they probably allow easier automation and can potentially reduce time to market and development costs. Furthermore, people with different domain knowledge are able to use models, as models are reusable and can be created with the help of a DSL.
The concepts presented in this thesis should be universally applicable. For the proof of concept, popular technologies are used, and there a several big companies offering a cloud platform. This thesis will focus on the Google Cloud Platform[3]. This platform is particularly interesting due to Google Cloud Dataflow[4]. Dataflow can be directly used without any setup. Additionally, the strong focus on cloud services saves time for many other setup tasks for a data processing pipeline. Also, all tests can be reproduced with the same services.
Protocols like MQTT[5] or AMQP[6] are often used as protocol for IoT applications. Nonetheless, they are only data protocols. They do not define semantics or how messages have to be structured. This leads to a state where everyone creates his or her own schema. Those schemas can vary in quality and are not compatible with each other.

1.3 Method

First, there will be an analysis of how data can be validated with the help of a domain model and how this can be adopted for streaming data. The goal is to validate the incoming structure and implement a simple outlier detection. With this approach, this thesis tries to create a foundation of how a domain model for IoT can be used in a stream environment. Further research can use this work to implement a more detailed validation approach with domain knowledge. This approach will be evaluated in a case study. For this study, data from real devices are used.

[3] https://cloud.google.com/
[4] https://cloud.google.com/dataflow/
[5] https://www.mqtt.org/
[6] https://www.amqp.org/

1.4 Use Cases

The proposed solution should be able to automatically detect constraint violations in a continuous data stream. The errors can be related to hardware or software issues. In the following, two potential use cases describe situations to which the approach of this thesis can be applied.

Testing during development

Software applications which involve hardware are often not easy to test. Mostly, there is a requirement to have the hardware onsite for testing. With a model-based approach, a model can first be used to describe the expected output of a device. This includes value types, min/max, and other constraints. During development, the model can be used to continuously validate the received data. It also facilitates development independently from the hardware.

Continuous validation

Often, it requires an additional effort to develop a system to validate data sent by a device. A domain model for IoT devices would have multiple purposes. It allows the generation of an adapter for the device, and it can also be used to validate incoming data on a server. This, in turn, can help to detect messages which are malformed or violating constraints.

1.5 Related Work

Stream processing is a vibrant research topic. Klein and Lehner (2009) have proposed a model for the propagation and processing of the quality of sensor data in streaming environments [2]. The definition of data quality in data streaming environments is based on the work of Wang and Strong (1996) [3]. They used the data quality categories *intrinsic* and *contextual* from Wang and Strong. As a result, five data quality dimensions are identified. These are *accuracy*, *confidence*, *completeness*, *data volume*, and *timeliness*. This thesis does not explore the term data quality further.
In 2016, Taleb et al. published 'Big Data Quality: A Quality Dimensions Evaluation' [4]. In this publication, they introduced a big data quality evaluation scheme. This evaluation scheme results in a data quality score, which is calculated with one or multiple data quality metric functions.
For Eclipse Vorto, there are currently two publications ([5], [6]) available. These describe

how Eclipse Vorto can be used, for example, to connect vehicles to the cloud. Besides these, there are a few publications ([7], [8]) on how a model-driven approach can be applied for IoT applications. These publications focus mostly on the generation of code for devices. How stream data is handled was out of their scope.
Cheng Xu et al. (2013) use another definition of model in the publication 'Model-based Validation of Streaming Data' [9]. They use mathematical models, which are described in a data stream query language. They describe two concepts: *model-and-validate* and *learn-and-validate*. For the first one, the idea is to define one or multiple formulas where the outcome can be validated. The second is to use a learning approach where a statistical reference model is created based on historical data which represents normal behaviour.
The dataset which will be used in the case study was analysed by Sharma et al. (2010) in 'Sensor Faults: Detection Methods and Prevalence in Real-World Datasets' [10]. They discuss different kinds of errors in sensor data. In their work, they compare rule-based and learning methods to detect abnormal data.

1.6 Outline

This thesis is structured in six sections. The following section gives a brief overview of models in software engineering and used techniques. Section 3 describes the used approach and the idea of this thesis. Section 4 shows a potential proof of concept which has been used in a case study. The thesis closes with a discussion of the results in Section 5 and a conclusion in Section 6.

2 Background

This chapter presents a short overview of how models are used in software development and techniques used later in Section 4.1. There are different concepts of how models can be used in software development. Depending on the point of view, they overlap in different areas. In the following, two different concepts are described.

2.1 Model

A model in software engineering provides an abstraction of a real-world system. It can represent the structure and function of a system. Also, different combinations are possible. The model contains only parts that are necessary for software development. This approach helps to focus on relevant aspects and prevents flooding in details. Between the models, relations can be established for complex coherences. There are various model concepts and notations available. These can be textual or visual. [11]
For example, a control software for a ship should be developed and, with models (see Figure 1), all relevant parts can be captured. This could include engine control, radar, light system, etc. The resulting models can now be used as a common base of knowledge. The models define the capabilities of the ship and its systems in context of the software. Furthermore, such models could be part of the software itself.

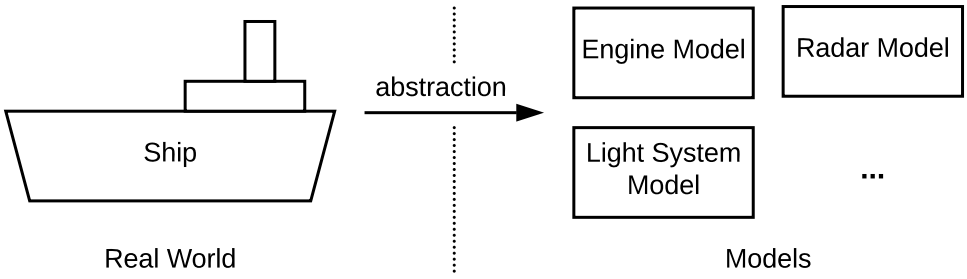

Figure 1: Abstraction of the characteristics of a ship into models

2.2 Model-Driven Software Development

There are two popular approaches for using models in software development. These are Model-Based Software Development (MBSD) and Model-Driven Software Development (MDSD). In the book 'Model-Driven Software Development' [12], Thomas Stahl et al. describe MBSD, as opposed to MDSD. In MBSD, models are mainly used before actual programming. With the help of a Unified Modelling Language (UML) different parts of

the software project can be described. The resulting models are later used to produce hand-written code. MDSD models can not only be used as part of documentation, but they are also part of the software itself. The models here have a specific representation in source code. Generally, there is a generator that transforms the model into code (see Figure 2). [12]

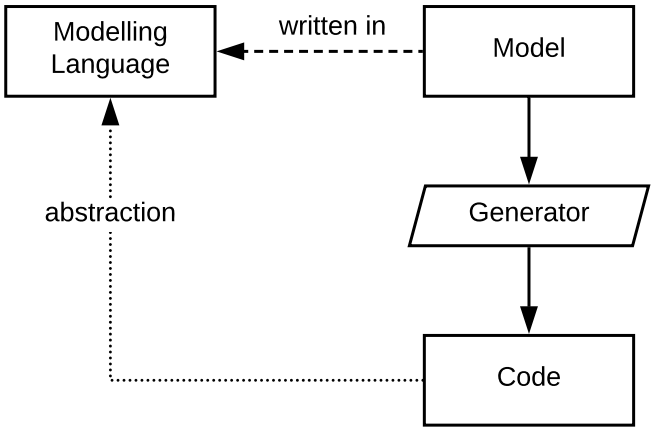

Figure 2: Model-Driven Software Development (excerpt from [13, p. 6])

Models mostly support the software development process. They can be useful in generating repetitive code and therefore speeding up development. The generated code is often combined with hand-written code.

2.3 Meta-Model

To create a model, a formal definition is required, including which concepts and properties are available. To define the concepts, another model is required. This model is called a meta-model. Each model is an instance of a meta-model. The meta-model is usually defined by the users of a modelling framework. An example of such a framework is given in Section 2.5. Such a framework also needs a model to allow for the creation of meta-models. This is called meta-meta-model. Like a meta-model, it defines all capabilities of the model at the next hierarchical level. This meta-meta-model is fixed and is part of the modelling framework itself. Figure 3 shows how the different model types build on each other. [12]

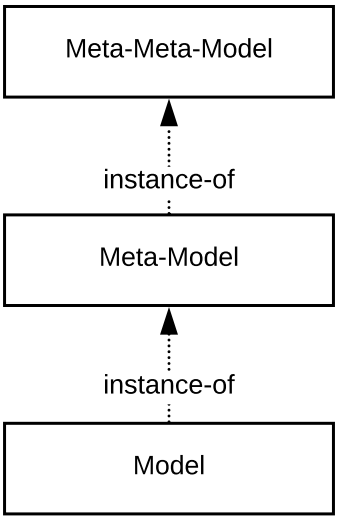

Figure 3: Meta-meta-model, meta-model and model instance

2.4 Domain-Specific Language

A DSL is a language that is specifically designed for a domain. The language allows the user to express the key aspects of a domain in a formal and machine-readable manner. It is human-readable and contains all the needed semantics from a domain for its purpose. An example of a widely used DSL is Structured Query Language (SQL). SQL was designed to create queries for a database system. There are also general-purpose languages which are not fixed to one domain (e.g. Java or C). The disadvantage of these is that they usually need far more language elements to model a part of a domain in contrast to a DSL. The DSL can have concrete language elements which represent parts of the domain. [12], [14]

Figure 4 gives an overview of the basic parts that a DSL consists of. A DSL contains a meta-model, semantics and a corresponding concrete syntax. The meta-model of a DSL is built on top of a meta-meta-model, which is usually provided by the used model framework. The concrete syntax is a realisation of an abstract syntax and defined by its structure. The concrete syntax is the textual expression, such as a programming language (e.g. Java). This text is used by a parser to load the model into the system. The model is represented by the abstract syntax and it defines the element of the resulting tree. [12]

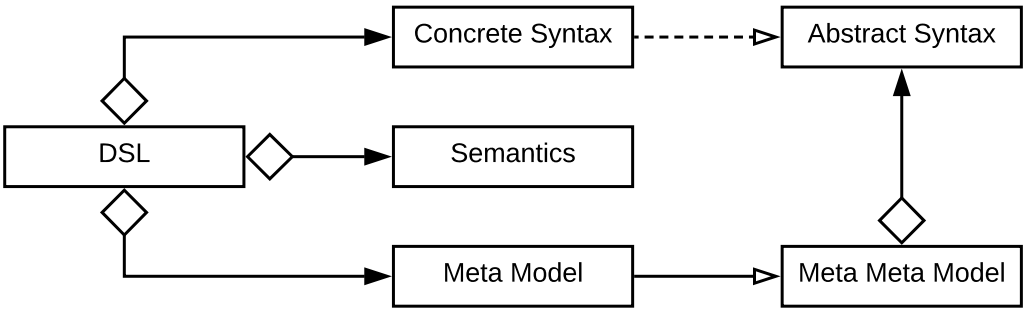

Figure 4: Relation between a DSL and related elements (excerpt from [12, p. 56])

2.5 Eclipse Modelling Framework & Xtext

The Eclipse Modelling Framework[7] (EMF) is an extension for the Eclipse Integrated Development Environment[8] (IDE). It allows the development of applications based on models. The framework provides different tools to create and use models. [15]
The starting point for modelling in EMF is the Ecore model. The Ecore model is a meta-meta-model which can be used to define meta-models. Figure 5 shows the view of the visual editor for a meta-model; the resulting model can be combined with Xtext[9] to implement a DSL.

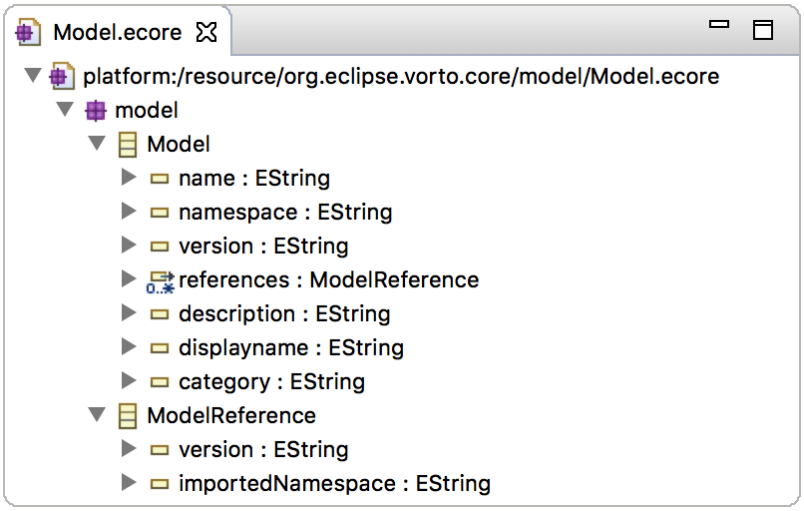

Figure 5: Ecore model of the Vorto DSL base model

Xtext is a framework which allows the user to implement programming languages and DSLs. It provides necessary parts for this purpose. These are a parser, interpreter and

[7] https://www.eclipse.org/modeling/emf/
[8] https://www.eclipse.org/ide/
[9] https://www.eclipse.org/Xtext/

code generator. Xtext integrates with the Eclipse IDE and allows the user to generate an editor with syntax highlighting, code completion, etc. for a self-designed DSL. [15]
Xtext allows one to connect the Ecore model with a corresponding syntax. Listing 1 shows an excerpt of an Xtext document, which uses the model shown in Figure 5. Each of the elements, like *namespace* and *version*, have a corresponding part in the Ecore model.
When an Xtext-based DSL is loaded, the syntax is validated and an Abstract Syntax Tree (AST) is created. The AST structure is given by the corresponding Xtext document. [15]

```
...
InformationModel:
  {InformationModel}
  'namespace' namespace = QualifiedName
  'version' version = VERSION
  'displayname' displayname=STRING
  ('description' description=STRING)?
  ('category' category=CATEGORY)?
  (references += ModelReference)*
  'infomodel' name=ID '{'
    ('functionblocks' '{'
      (properties += FunctionblockProperty)*
    '}')?
  '}';
...
```

Listing 1: Part of an Xtext document which relies on the Ecore model in Figure 5

2.6 Eclipse Vorto

Eclipse Vorto is an open source framework to support an MDD approach for IoT applications. Vorto provides a DSL which defines the terminology for specifying capabilities and functions of a device. Vorto provides different generators which allow the transformation of a model into adapter code. These adapters can be used for functions like connecting a device to a platform. An adapter can contain all the required parts to establish a connection and send messages. [16] The benefit of this approach is that these models can be reused and the adapters can be generated without much effort.

2.6.1 Architecture

Figure 6 shows a potential architecture of how Vorto can be used in an IoT scenario. This was created by the lead developer Alexander Edelmann. Vorto is still under development

and should be seen as target architecture for the future. Not all concepts are implemented and publicly available.

The Vorto Eclipse Toolset is based on EMF and Xtext. It allows the creation of Information Models. Such a model can be used to generate a platform adapter. The adapter consists of a Vorto interface, a serialiser and a protocol client. The interface of the adapter can be written in different programming languages and it exposes all functions and properties which can and must be implemented. The serialiser takes the data from the interface and transfers it to the protocol client. This client manages the connection to the cloud. Different protocols, like MQTT, are possible. On the cloud side, there should be a Vorto message parser. This converts the format to a standard Vorto format, where it can be validated and processed further.

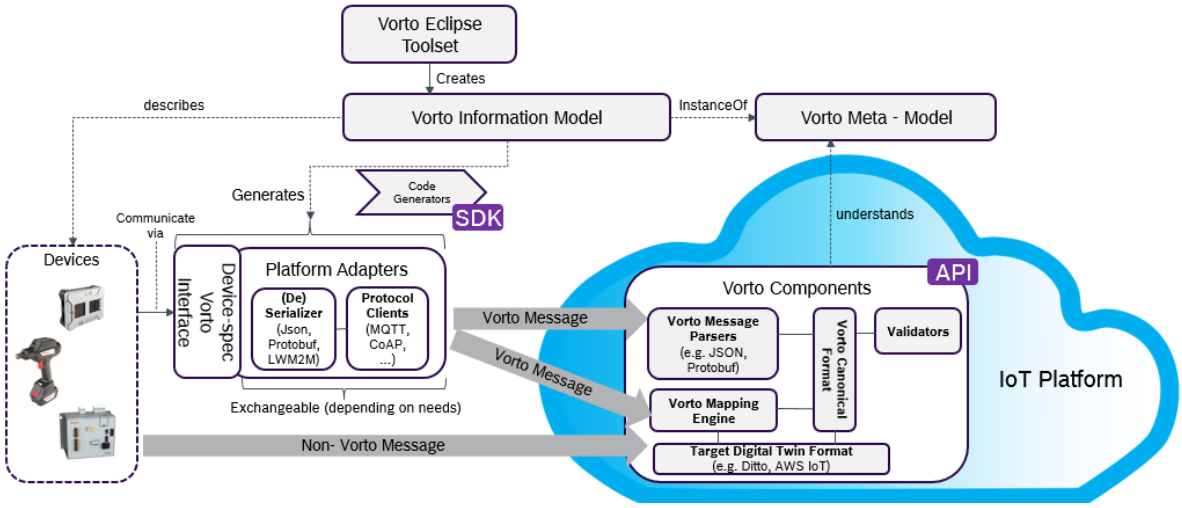

Figure 6: Vorto architecture concept [17]

2.6.2 Vorto DSL

As already mentioned, the Vorto DSL allows the abstraction of a real-world device into a model. Vorto provides, for this purpose, a strongly typed language. A basic overview of available models is shown in Figure 7. Each model needs a unique identifier, which consists of a *name*, *namespace* and *version*. In the following, the content of each model is explained in detail.

Information Model A device is represented as an Information Model and provides the base for all other parts. The model can contain one or multiple Function Blocks.

Function Block A Function Block represents the functions and properties of a device. For this purpose, it is split in four sections: *configuration*, *status*, *fault* and *operations*. The status section provides all properties which can be read, e.g. sensor values. The configuration section can be used for device configuration, e.g. the identity of a device. In the operations section it is possible to define method signatures. Each method can have multiple parameters and a return type. In faults, potential errors can be defined like a sensor measurement that is out-of-range.

Data Type A Data Type is a complex data type which can contain other primitive and complex data types. It can only be referenced by one or multiple Function Blocks. In the Vorto DSL for a Data Type the notation Entity is used.

Enum An Enum is known from other programming languages as enumeration of different values. An element of this Enum cannot store another value like in the Java programming language.

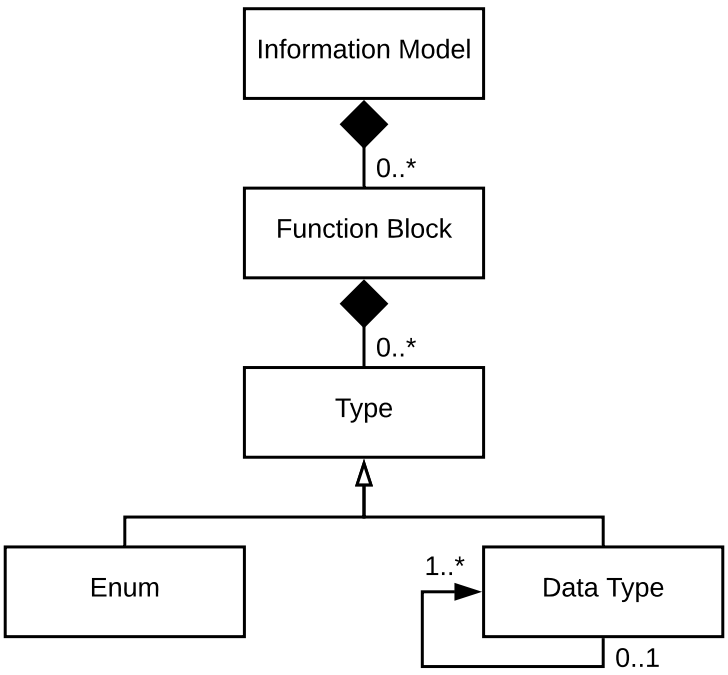

Figure 7: Simplified ER-diagram of the Vorto meta-model

The Vorto DSL also provides a mapping concept, e.g. map a property to a protocol specific property. For each model shown in Figure 7, it is possible to define a mapping.

2.6.3 Model Repository

Vorto provides a central repository where models can be stored. All models described in the previous section can be uploaded individually. Everyone can share their models or make use of the models of others. The repository can be accessed by a web interface or API. In the Vorto editor, the repository can be searched directly and models can be included in new projects.

2.6.4 Constraints

The Vorto DSL allows the user to define multiple constraints for each property. An overview is presented in Table 1. Not all constraints can be applied to all data types. For example, the *STRLEN* constraint can only be applied to a string. The Vorto editor shows an error if the constraint does not match the data type.

Constraint	**Apply to**	**Description**
MIN	dateTime, double, float, int, long, short	minimum value for the property
MAX	dateTime, double, float, int, long, short	maximum value for the property
STRLEN	string	length of a string
REGEX	string	regular expression the property has to match
MIMETYPE	base64Binary	defines media type of the binary data
SCALING	dateTime, double, float, int, long, short	scaling factor for the property
DEFAULT	boolean, dateTime, double, int, float, long, short, string	default value for the property
NULLABLE	all types	defines if a property can be *null*

Table 1: Constraints provided in the Vorto DSL

Listing 2 shows an example of how constraints can be applied to a property. The temperature property here is marked as mandatory and restricted to a value between -55°C and 128°C. Each property can have additional attributes. In this example, the feature is used to define Celsius as measurement unit.

```
1  entity HeatingTemperature {
2    mandatory temperature as float with {
3      measurementUnit: Temperature.Celsius
4    } <MIN -55, MAX 128> "Temperature range in celsius"
5  }
```

Listing 2: Entity with a temperature property

2.7 JSON & XML

JavaScript Object Notation (JSON) and Extensible Markup Language (XML) are both popular human readable data formats. Most programming languages support these standardised formats with a built-in or external library. One of the reasons these are so popular is that everyone can define their own message format and they are easy to use. For both data formats, additional standards exist which allow the user to define a schema and restrict specific parameters, e.g. range for a number.

2.8 Google Cloud Dataflow & Apache Beam

Dataflow is a big data analytics service which is offered in the Google Cloud Platform. It was first introduced in 2014 [18]. With a unified programming model, it facilitates the creation of pipelines for data analytics which are independent from the underlying processing engine. Figure 8 illustrates how these parts connect. As input modes, streaming and batch processing is supported. How data should be transformed is described in a dataflow model and the model can run on different runtimes like Apache Flink[10] with the provided Software Development Kit (SDK).
In 2016, Google donated the Dataflow framework, to create pipelines, to the Apache Foundation [19]. There, it is now developed under the name Apache Beam[11]. To create pipelines, an SDK for Java and Python is available. The advantage of this split between pipeline and runner is that there is no need to rewrite a pipeline if the runner is changed, e.g. using Apache Flink as runtime instead of Google Cloud Dataflow.

[10] https://flink.apache.org/
[11] https://beam.apache.org/

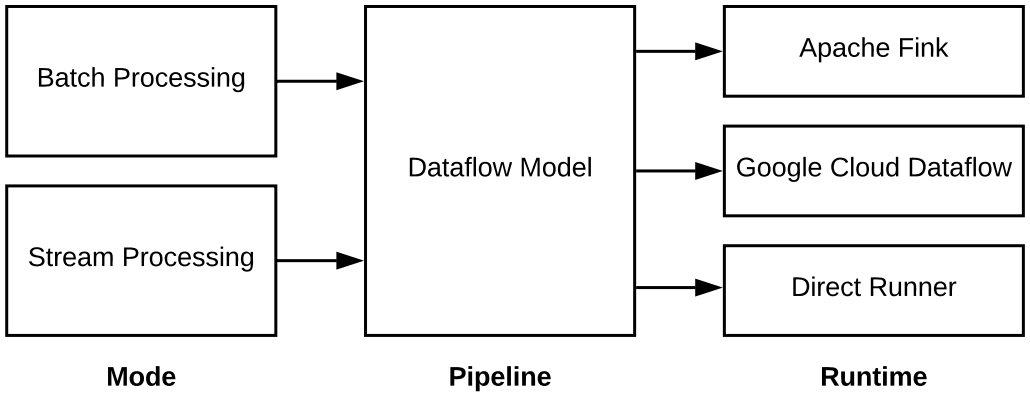

Figure 8: One pipeline and different runtimes [19]

A linear pipeline of Apache Beam consists of an input, one or multiple transform functions, and an output. It is also possible to have branches with multiple outputs. Figure 9 shows a pipeline which first transforms the output to a new *PCollection*. A *PCollection* is used as output for each transform function and can also be used as input. After that, depending on the input, the data gets transformed by function A or B. Then the data is given to the relevant output. There are a lot of variation possibilities to create a pipeline which fits for various applications. [20]

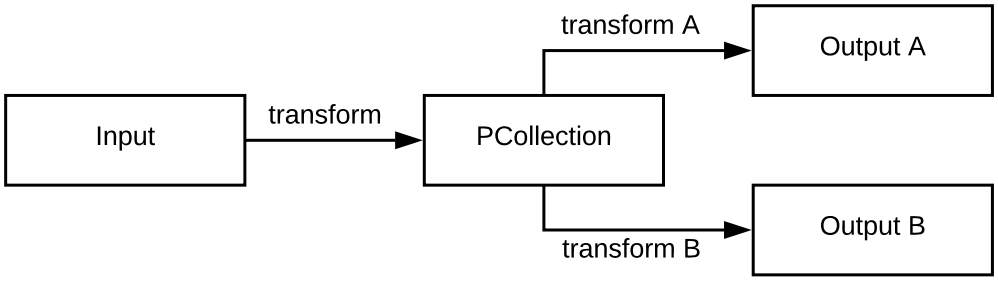

Figure 9: Apache Beam pipeline with a branch

One of the core operations of Apache Beam SDK is *ParDo*. With *ParDo*, each element to be processed will be passed to a user-defined function (*DoFn*). This function can yield zero or more elements. With *ParDo*, the parallel processing of an unbound data stream is not a problem. The function only gets one element passed and does not need knowledge of other elements in the stream. [21]

3 Approach

As already mentioned in Section 1.2, the goal of this thesis is to make use of a domain model to validate incoming data of real-world devices. The domain model used for this case is the Vorto model. There are not many DSLs for IoT available. Vorto was selected because it seems to be the one with the most active development (as seen on GitHub[12]). The Vorto DSL allows users to define individual constraints for each property. These constraints should be automatically transformed to rules which can be applied to incoming data. The rules should mark messages which violate one or multiple constraints. Additionally, some information about the reason should be added. Besides this, the structure should be validated, as should the presence of all mandatory properties. If attributes are present that are not defined, this should result in an error.
The first idea was to generate SQL-like queries for real-time analytics systems like Amazon Kinesis Data Analytics[13]. Conceptually, it is possible to generate SQL queries with a new Vorto generator. After some tests, the possibility of loading and evaluating a Vorto Information Model directly in a stream processing system was realised. The advantage of this is that it is not necessary to generate and deploy new code every time the model changes. It allows multiple models to be used for validation without much effort.
To use the domain model in a stream environment, Apache Beam was chosen. As described in Section 2.8, Apache Beam currently allows a flexible use of different popular engines. This brings portability to the presented approach without additional effort.
The Vorto DSL is based on EMF and Xtext. Existing Vorto models can be loaded as an Ecore model and all properties can be easily accessed within Java. The Vorto project is divided into multiple projects which are published as separate dependencies. This means that only the required parts for the Vorto DSL can be included.
Figure 10 illustrates the data flow and how a model can be used in an IoT scenario. The sensors can measure the environment and create sensor readings. With hand-written code, the sensor data is given to the platform adapter. This platform adapter was created by a Vorto generator. This generator uses the model which describes the capabilities and function of the IoT device. Additionally, the model provides constraints about the sensors like temperature range. The platform adapter is capable of transforming the incoming data to a format which the IoT platform can process. This platform receives the incoming data stream and forwards it for data validation. Here, the model is used to validate the data structure and all constraints which are defined.

[12] https://github.com/eclipse/vorto
[13] https://aws.amazon.com/kinesis/data-analytics/

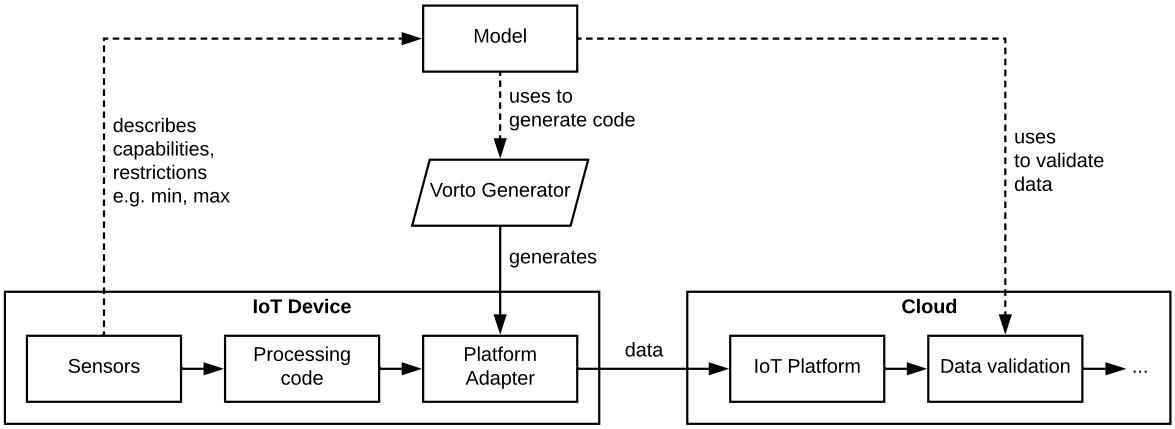

Figure 10: Dataflow and use of a model in an IoT scenario

3.1 Stream and Batch Data Processing

The system should be able to process a stream of incoming real-time data and give instant feedback. It should also validate existing data. For this purpose, the data should be loaded and processed in a batch. This can be helpful for multiple reasons. First of all, data that already exists can be validated and a Vorto model can be created afterwards. Secondly, it allows the user to re-evaluate data if the model has changed.

3.2 Rule-Based Data Validation

The data validation must be deterministic and reproducible. The Vorto DSL allows the user to define constraints for each property. Furthermore, the structure, types, etc., are also given by a model. Constraints and structure should be validated. The constraints could be checked with simple rules. A rule would be, in this case, an if-statement. If the check fails, the message is marked and additional details about the error should be added.

The validation process should contain the following steps:

1. Receive a message
2. Parse JSON string to a Java object
3. Compare the object with the corresponding Vorto model
4. Return *null* or create an object with errors marked
5. Serialise the object to a JSON string

Besides the constraints, the Vorto DSL allows the user to define types for each property. These can be a primitive type like *int*, *long*, etc., or a complex type. A complex type can contain further complex types and primitive types. For all primitive types, it should be checked if they can be parsed by the corresponding type in Java.

3.3 Comparison to JSON-Schema

Currently, JSON-Schema has many more features to validate a JSON document compared to the Vorto DSL. In the following, there is a limited comparison of the current features of the Vorto DSL and the last draft of the JSON-Schema Validation standard [22] of the Internet Engineering Task Force.
The section 'Validation Keywords' of the JSON-Schema Validation standard is divided into seven subsections. The first five define keywords, which apply to different data types. The other two sections define keywords for conditions and Boolean logic.
In Table 2, there is a comparison between JSON-Schema and Vorto. It contains a list of constraints which apply to any instance. JSON-Schema permits the user to define the *type* of a property. Vorto requires the user to define a type for each property. JSON-Schema permits only the definition of primitive types here. In Vorto, these can also be of a complex type. In JSON-Schema, an enum is defined as an array which can contain any value. In Vorto, this is limited to strings. With *const*, JSON-Schema can define constant values. Vorto does not allow this currently.

JSON-Schema	**Vorto**
type	*PrimitiveType*
enum	*Enum*
const	-

Table 2: JSON-Schema validation keywords for any instance type compared to Vorto

For numbers, the keywords listed in Table 3 are available. The *multipleOf* keyword allows the user to check if a number divided by the defined number is an integer. Vorto does not support this. JSON-Schema allows the user to define if the *minimum* and *maximum* for numbers is exclusive. If *exclusiveMinimum* is *true*, then only a value bigger than the given *minimum* is allowed. For example, if the minimum is 5, then 5 would not be allowed but 5.01 is valid. The reason for this is that it is not necessary to define a float

with many decimal places as minimum. Vorto has the same minimum and maximum feature, but currently has no option to set this to exclusive.

JSON-Schema	**Vorto**
multipleOf	-
exclusiveMaximum	-
minimum	MIN
exclusiveMinimum	-
maximum	MAX

Table 3: JSON-Schema validation keywords for numeric instances compared to Vorto

Table 4 lists the JSON-Schema keywords which can be used for strings. Vorto currently only allows setting the maximum length of a string with *STRLEN*. It is not possible to define a minimum. The reason for this could be that the constraint was included for another purpose. With *STRLEN* is could be possible to give constraint devices information on how much memory they need to allocate for a certain value. A *pattern* is defined as regular expression. The same can be achieved with the *REGEX* constraint in Vorto.

JSON-Schema	**Vorto**
maxLength	STRLEN
minLength	-
pattern	REGEX

Table 4: JSON-Schema validation keywords for strings compared to Vorto

Also, for arrays, JSON-Schema has multiple keywords to restrict the content (see Table 5). For arrays, the *items* keyword defines a schema for each array item at the exact position or one schema for all elements. The *additionalItems* keyword allows the validation of all elements at a position greater than the size of *items* if it contains an array. With *maxItems* and *minItems* the size of an array can be limited. If the *uniqueItems* keyword is set to true, each element of the array needs to be unique. With the *contains* keyword a JSON-Schema can be defined where at least one item needs to exist where the schema matches.
Vorto currently does not support any of these array validations; it uses the *multiple*

keyword to define if a property is a list. All items stored in such a property are required to have the same type.

JSON-Schema	**Vorto**
items	-
additionalItems	-
maxItems	-
minItems	-
uniqueItems	-
contains	-

Table 5: JSON-Schema validation keywords for arrays compared to Vorto

To define the content of an object JSON-Schema has multiple available keywords. They are listed in Table 6. Similar to numbers and strings, the count, here, can also be restricted with *maxProperties* and *minProperties*. Because of the structure of a Function Block and data type, this is implicit. Vorto requires the user to define each used property. Therefore, this boundary is not required for Vorto. With *required* in JSON-Schema, it can be defined that an object must contain the listed properties. A similar behaviour can be achieved with the *mandatory* modifier in Vorto. This modifier can be added to each property. The *properties* keyword allows the definitions of regular expressions that the properties of an object need to match. Keywords like *properties* and *patternProperties* are not applicable to Vorto. As mentioned, Vorto requires the explicit definition of all properties and references. For the *dict* type in Vorto this feature could be implemented.

JSON-Schema	**Vorto**
maxProperties	*not required*
minProperties	*not required*
required	mandatory
properties	*implicit*
patternProperties	*not required*
additionalProperties	*not required*
dependencies	*not required*
propertyNames	*not required*

Table 6: JSON-Schema validation keywords for objects compared to Vorto

Besides introducing the keywords, JSON-Schema can define if-then-else statements. Conditions are currently not supported by the Vorto DSL. There is much more which JSON-Schema can define, but the preceding content should be enough to give an impression of how it compares to Vorto.
As shown, the same basic checks can also be achieved with JSON-Schema. It can be argued that a generator for such a schema would have the same benefits. This is true, but it would take additional effort to limit the schema to the IoT domain. This would result in something similar to the Vorto DSL. However, the benefits of EMF, like a development IDE, would not apply. The desired approach should be applicable for different message formats and not limited to JSON.
The benefit of using the Vorto DSL compared to JSON-Schema is the given structure and semantics. In Vorto, for example, each property can have additional attributes defined as shown in Section 2.6.4. This provides additional value for each property definition. Vorto can be seen as kind of a standard with a given structure. With JSON, one would have to start from scratch. The DSL is optimised for the domain of describing real-world devices in a machine-readable manner. Each developer that needs to interact with the data can get the same understanding of the data.

4 Evaluation

To evaluate the benefits of the approach in the previous section, a prototype was developed. First, the features and implementation of the prototype are explained in detail. Second, the prototype was used in a case study to validate an existing dataset.

4.1 Proof of Concept

The inspiration of this thesis was how rules can be generated from models to continuously validate constraints in a data stream. To test the approach, a prototype called VortoFlow was developed. VortoFlow is a Java library which can be included in an Apache Beam project. The library provides a class which can be added to an Apache Beam pipeline. The idea is that besides the model validation, further steps to transform the data can be included in the final project. This could save costs because the messages are already loaded. The processing function in VortoFlow is stateless and can be included without much effort.
In the current state, VortoFlow is not optimised for performance. The project currently has only a small set of supported features. The focus was to implement all necessary functions for the case study described in Section 4.2.

4.1.1 Loading a Vorto Model

To load an existing Vorto model with all related models, the class *VortoModelLoader* was created. The class allows the system to load a model from three different locations. The model can be included as a resource in the jar of the project, it can be stored in the local file system, or Google Cloud Storage[14] can be used. For Google Cloud Storage, all files have to be stored as a single ZIP file. Loading a model from the Vorto repository is currently not supported but can be implemented into a later version.
The Vorto model is loaded outside of an Eclipse environment. This requires using the standalone mode. A few things, like the Google Guice Injector[15] and the required information about the model need to be configured. [15] For all four models (Function Block, Information Model, Data Type, and Mapping) the *...StandaloneSetup.doSetup()* method is called. This creates the injector and registers the model within EMF.
The *VortoModelLoader* has a *loadModel* method. This can be used to load a model. If the model is not loaded, it calls the *loadInformationModel* function. This method creates

[14] `https://cloud.google.com/storage/`
[15] `https://github.com/google/guice`

a *XtextResourceSet* and loads all resources with the appropriate method. Afterwards, all resources are checked for grammar errors. If an error exists, the method returns *null*. Otherwise, a new *VortoModel* class is returned. This class contains the loaded Information Model. The class allows the system to access all properties, including all loaded dependencies.

4.1.2 Validation

Each incoming message is processed by the *processElement* method of the *ValidateDataFn* class. The class extends the *DoFn* class of the Apache Beam Framework. This facilitates the implementation of user defined functions. The DoFn object has to fulfil some requirements defined by Apache Beam. The class must be serialisable and thread-compatible. The documentation of Apache Beam [23] also contains a recommendation to make the function object idempotent. The *ValidateDataFn* class fulfils all those requirements.
As input, the function requires a JSON object as string. In the *processElement*, the JSON is loaded with the help of the *ObjectMapper* from the library jackson-databind[16]. Afterwards, the object is passed to the method *validateInformationModel* of the *VortoModelValidator* class. For each input, the function creates a deterministic output. The Vorto DSL allows the user to define different constraints for properties. Table 7 contains all constraints currently supported by VortoFlow.

Constraint	**Implemented**
MIN	X
MAX	X
STRLEN	X
REGEX	X
MIMETYPE	-
DEFAULT	-
NULLABLE	X

Table 7: List of implemented constraints in VortoFlow

For *MIN* and *MAX*, the constraint value and the incoming value are parsed as double in Java. The defined data type of a property in the Vorto model is validated in another

[16] `https://github.com/FasterXML/jackson-databind`

function. Then, both values are compared with the according Boolean operators <= or >=. For example, if the incoming value is 10 and the maximum is 9, an error would be raised.
To allow the validation of the dataset discussed in Section 4.2.1, the Vorto DSL was extended with the *NULLABLE* constraint. With this, a property can be marked as containing *null* or requiring a value other than *null*. The Vorto DSL allows the user to mark properties with a modifier as *mandatory* or *optional*. If a mandatory property is missing, an error message is created.
Schema validation can be quite computation-intensive. The standards for JSON and XML have no limit for nesting, but most parsers have a hard-limit for nesting or working as long as enough memory is available to parse a JSON object. Currently, VortoFlow does not limit nesting actively.

4.1.3 Error Handling

When VortoFlow detects an error a new JSON object is created and all details about the error are added to this. The error object contains a Boolean value if an error was detected. The structure of the object follows the structure of the Function Block. It contains the status and all properties for which an error was detected. Each attribute contains a list of constraints which are violated. A constraint error gives information about the type of error along with a short description. The original payload is always included in the object.
Listing 3 shows an example output. In this example, a wrong sensor reading of humidity was detected. Humidity is defined as *float* in the Function Block with a range of 0 to 100. The property can be *null*. The payload attributes hold the original input where the constraints were violated. Here, it is shown that the humidity value was -4. This is out of the defined range of the model.

```
{
  "error": true,
  "weathersensor": {
    "error": true,
    "properties": {
      "error": true,
      "status": {
        "humidity": {
          "error": true,
          "constraints": [{
            "type": "MIN",
            "error": true,
            "msg": "Value is smaller than min (0)."
  }]}}}},
  "payload": {
    "weathersensor": {
      "properties": {
        "status": {
          "moteid": 19,
          "date": "2004-03-20",
          "time": "09:04:54.844218",
          "epoch": 61453,
          "temperature": 122.15299999999999,
          "humidity": -4,
          "light": 596.16,
          "voltage": 2.31097
  }}}}
}
```

Listing 3: VortoFlow error message

4.2 Case Study

To evaluate the VortoFlow prototype as described in the previous section, an existing dataset was used. The idea was to use data from a real installation with real problems that occurred. One of the main challenges was finding raw data that contains sensor readings with errors. The errors should be obvious to find. This helps to validate the outcome of the prototype. In this section, the used dataset and the findings are explained in detail.

4.2.1 Intel Lab Data

In the Intel Berkeley Research Lab, 54 Mica2Dot Mote[17] boards equipped with weather boards were deployed. A Mote is a small embedded board based on an Atmel ATmega 128L. This board was extended with a weather board which provides multiple environment

[17]https://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2dot.pdf

sensors. With this board, the Motes were capable of measuring the temperature, humidity and light. The installation was running between February 28th and April 5th 2004. [24] The dataset contains a lot of obvious errors which makes it ideal for testing VortoFlow. In a data analysis of Sharma et al. they found that there is a correlation between sensor failures and the voltage. All devices started to report incorrect values when the battery level was too low. [10] Figure 11 shows the humidity of the Mote with id 1. The humidity started to fall far below zero after 26th March 2004. In the graph, it is clearly visible where an error occurred. All other Motes have shown similar behaviour. The idea is to use this dataset to detect all values out of range with the help of VortoFlow.

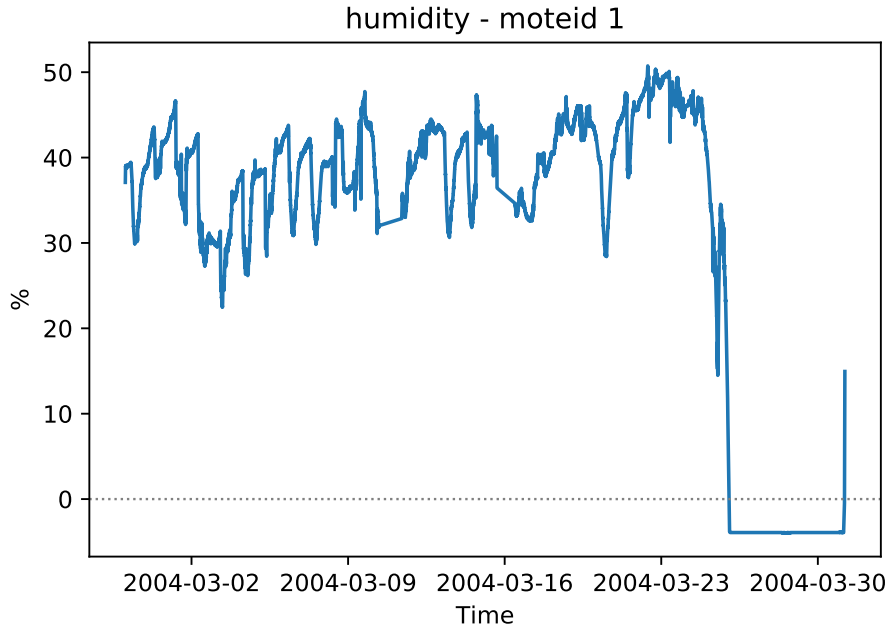

Figure 11: Humidity graph of Mote with id 1

4.2.2 Vorto Model for Intel Lab Data

This section describes how a Vorto model for the Intel dataset can be created. The idea is to use information from the data sheets of the used sensors. This information is already available before any program code is written. The model should give an indication that VortoFlow works and general errors can be detected without much effort. The model could be refined later with knowledge about the physical boundaries of the environment; for example, it should be impossible that a room has a temperature of 100°C.
The Mica2Dot weather board is an extension for the Mica2Dot micro controller. An Information Model for this setup could be defined as shown in Listing 4. This model references the Function Block for the weather board as *weathersensor*.

```
using de.hu_berlin.intellabsensor.fb.MICA2DOTWeatherSensor ; 1.0.0

infomodel IntelLabSensor {
  functionblocks {
    weathersensor as MICA2DOTWeatherSensor
  }
}
```

Listing 4: Information Model for Intel Lab Data

Listing 5 shows a Function Block for the weather board. The model properties match the existing data. From the description of the data, it is known that 54 Motes were in the field. This results in a constraint that a *moteid* can only be a value between 1 and 54. For time and date, the format was derived from the existing data. This resulted in two regular expressions to validate those values. The *epoch* starts when the device is turned on. It is a value bigger that zero. For the upper bound, no information was found. This could be the limit of the data type.

```
functionblock MICA2DOTWeatherSensor {
  status {
    // yyyy-mm-dd
    mandatory date as string <REGEX "[0-9]{4}-[0-9]{2}-[0-9]{2}">
    // hh:mm:ss.xxxxxx
    mandatory time as string <REGEX "[0-9]{2}:[0-9]{2}:[0-9]{2}.[0-9]{1,6}">
    mandatory epoch as int <MIN 0>
    mandatory moteid as int <MIN 1, MAX 54>
    mandatory temperature as float <MIN -40, MAX 123.8, NULLABLE true>
    mandatory humidity as float <MIN 0, MAX 100, NULLABLE true>
    mandatory light as float <MIN 0, MAX 1847.36, NULLABLE true>
    mandatory voltage as float <MIN 0, MAX 3.2, NULLABLE true>
  }
}
```

Listing 5: Function Block for Mica2Dot weather board

The board contains multiple sensors. For the trial, it seems that the Sensirion SHT11 for temperature and humidity and a TAOS TSL2250 for ambient light were used [25]. For the SHT11, the minimum and maximum range for the measurement can be directly obtained from the data sheet. The SHT11 has a defined measurement range from -40°C to 123.8°C for the temperature and 0% to 100% for humidity [26]. The range can vary with the chosen binary format.
The TSL2250 sensor has two separate photodiodes. One measures visible and infrared light and the other only infrared light. Both values are used to calculate the lux value.

Because the values are transferred digitally without using float, there is a formula to calculate the resulting float value. In one byte, the chord number (C) and step value (S) are encoded. Each sensor has its own channel for which a count can be calculated. Both count values are used to calculate the final lux level. The formula is provided by the manufacturer of the sensor. [27]
It seems that the light sensor stopped working when the battery level was low and no values out of range were recorded. When using the formula from the data sheet, a maximum lux level of 1846.9 can be calculated. In the dataset the maximum measured value is 1847.36. This could be due to the float precision of the Mica2Dot. For the data validation, 1847.36 will be used as maximum value and not detected as error. For the voltage level a maximum voltage level of 3.2 Volt is used. This value is also obtained from the data. It looks like the used coin cells have variated in their voltage level.

4.2.3 Test Program

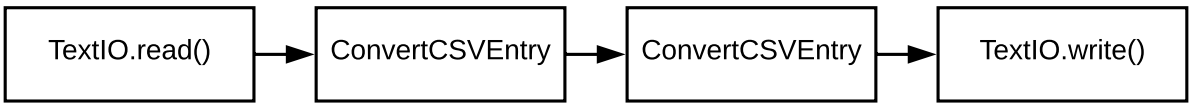

Figure 12: Pipeline for the Intel dataset

To validate the Intel dataset with VortoFlow, a test program was written which makes use of the *ValidateDataFn* (see Section 4.1.2). This program defines and runs an Apache Beam pipeline to process the dataset. The resulting pipeline is shown in Figure 16. First, the dataset is loaded as a ZIP file with the help of *TextIO.read()* from a Google Cloud Storage Bucket. The *read()* method of *TextIO* allows the user to load and process text files line by line in a pipeline. The next step in the pipeline is the class *ConvertCSVEntry*. This class transforms a line of the Comma-Separated Values (CSV) input to a JSON object. The JSON object is structured suitably for the model described in Listing 5. The JSON object is finally serialised and returned as string. This JSON string is now processed by the *ValidateDataFn* class. Last is the *TextIO.write()* step. Here, all messages are written to a text file on a Google Cloud Storage Bucket which are tagged as errors. Each line contains a JSON object which describes the error. The error format is described in Section 4.1.3.

4.2.4 Detected Errors

This section describes how detected errors are visualised and compared to manual findings in the dataset. For this purpose, the Python libraries pandas[18] and matplotlib[19] were used. Pandas can be used to load and manipulate the dataset. With matplotlib, the data can be visualised. The figures in this section were created with these libraries.
Multiple violations of the humidity constraints were detected. Figure 13 shows an example of such a violation. Here, starting at 26th March 2004 00:30:05, the humidity dropped below zero. All values below zero are marked with the dotted line. These are values which violate the constraint of *MIN* for humidity.

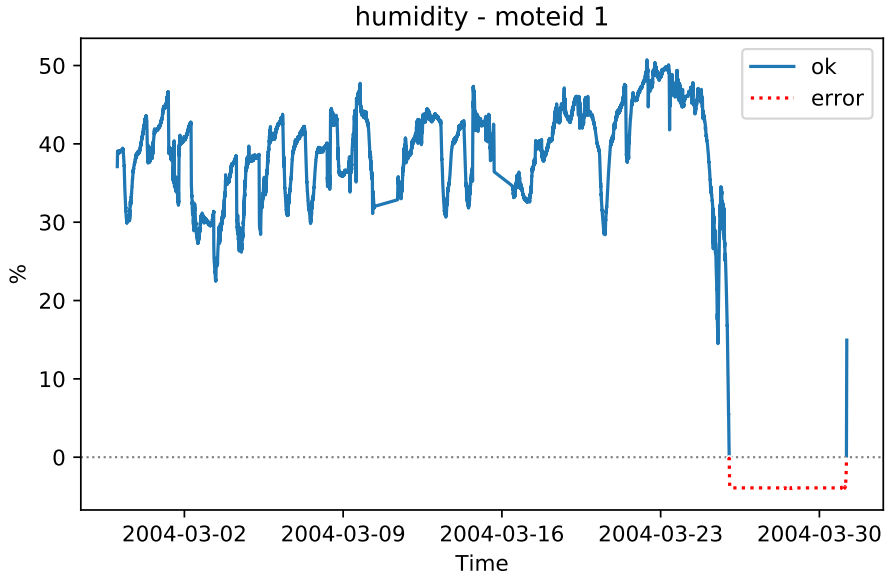

Figure 13: Errors detected in humidity readings of Mote with id 1

The model for the temperature sensor defined the maximum temperature at 123.8°C. However, the maximum value for a temperature recorded was only 122.153°C. This value could be the result of the chosen calculation method or used data type. In the graph, in Figure 14, it is obvious that there is something odd about the data. But VortoFlow was not able to detect the error because the range was too widely defined.

[18] https://pandas.pydata.org/
[19] https://www.matplotlib.org/

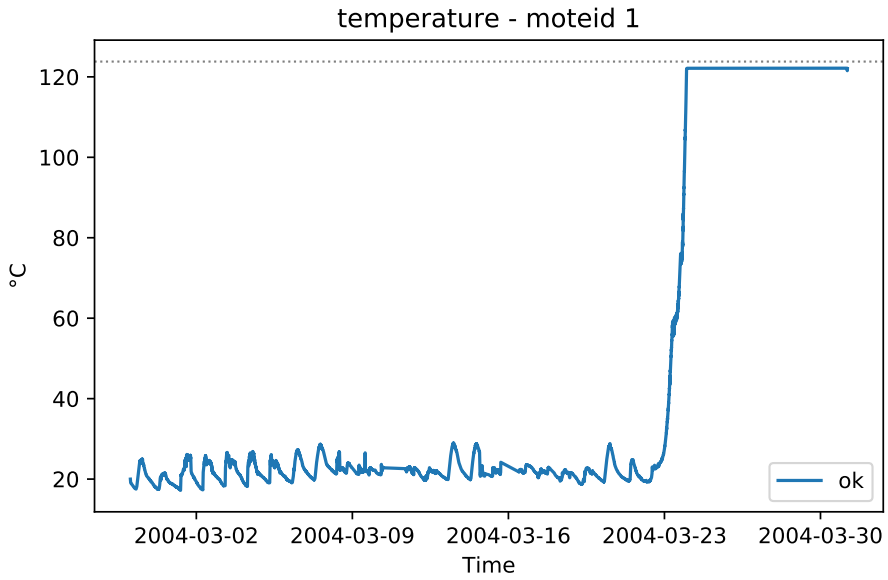

Figure 14: Temperature readings of Mote with id 1

This first example is showing that the general approach works and errors could be detected. The second example shows that if the constraints are too wide, VortoFlow is not able to detect errors. For this example, it would have been better to define a maximum according to physical constraints of the environment. Under normal circumstances, it is not possible that the Mote measures a temperature around 100°C. The normal room temperature would be around 23°C. A maximum of, for example, 60°C would fit. Figure 15 shows how VortoFlow could detect errors in temperature readings if the *MAX* constraint was set to 60°C.

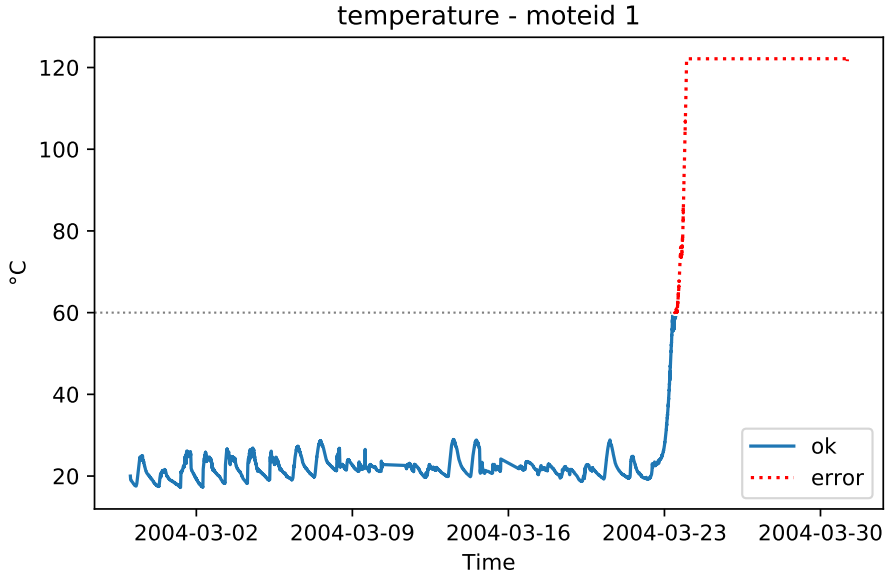

Figure 15: Errors detected in temperature readings of Mote with id 1

Checking only the range of values can be seen as the first indication for errors. The chosen constraints should be as restrictive as possible. Further checks are recommended and the best results could be achieved with a combination of multiple approaches.

4.2.5 Performance

VortoFlow is not optimised for performance at the moment. Nonetheless, the tests with the Intel dataset have shown that processing of data can be done in a reasonable time. In total, it takes around five minutes for one worker to process the complete dataset, including the setup and shutdown of a worker. For the test, the latest version of the Apache Beam SDK (2.4.0) for Java was used. Figure 16 shows the result of the batch job in detail. Each box represents one computing step as described in Section 4.2.3. Additionally, each box contains information on whether the step was successful and the wall time it required. Wall time represents the approximate time that it has taken from initialisation to termination. As expected, the validation step needs the most time with around 1 min 22 sec. The dataset contains 2,313,682 elements which means, per run, around 28,216 elements were validated per second. The test was repeated multiple times with similar results.

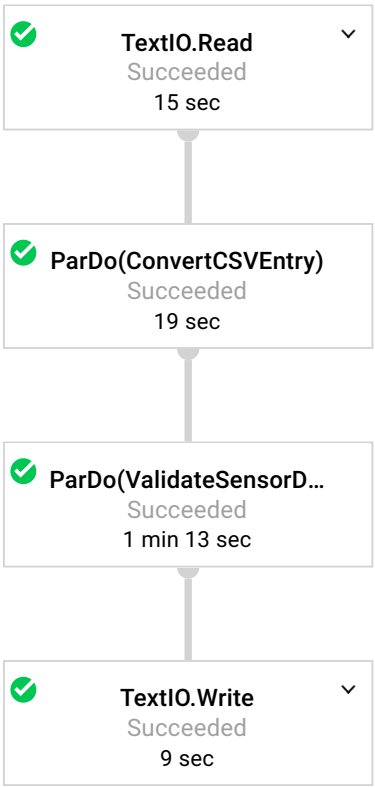

Figure 16: Batch job for the Intel dataset

Table 8 lists the results of three independent test runs. For each step the wall time is given along with the average over all runs, where calculated. There are multiple reasons why the results are varying each time. The read and write tasks require the system to access the network. Here, the available bandwidth may vary. Furthermore, the processing of the data requires memory and CPU time, as well as the fact that the hardware may be shared with other users. This can potentially explain the changing timings for the different tasks.

TextIO.Read	**ConvertCSVEntry**	**ValidateSensorData**	**TextIO.Write**
19 sec.	26 sec.	1 min. 31 sec.	12 sec.
17 sec.	24 sec.	1 min. 14 sec.	10 sec.
14 sec.	26 sec.	1 min. 22 sec.	10 sec.
Average:			
~17 sec.	~25 sec.	~1 min 22 sec.	~11 sec.

Table 8: Runtime of three independent runs with Google Cloud Dataflow

The internal structure is not optimised for a quick access of all properties. This could be improved in the future. Currently, each time a validation of the *REGEX* constraint is executed the regular expression is recompiled. A better approach would be to cache the compiled expressions.

4.2.6 Results

With VortoFlow, it was possible to detect elements that violate the constraints of the created model. It was not very difficult to create such a model for the given dataset. The goal to perform validation of an existing dataset could be met, there is some space for improvement. In particular, the chosen approach to use the boundaries of the sensor data sheets. The dataset that was used is not very complex and does not require any nesting in the Vorto model. However, the results also show that the quality of the error detection depends on the chosen constraints. If the constraints are too broad, errors can remain undetected. A lot of other research already exists on how anomalies in data can be detected. Further research could be done on how the results can be transferred to the Vorto DSL.

5 Discussion

As shown with the domain model of Vorto, it is possible to validate stream data. Without much effort, this approach can help to detect errors in a data stream. The implementation of the device software can be tested and there is also direct feedback if the assumptions in the beginning are met by real data. No hand-written code is required for such checks. When the system is later in use there can be a continuous loop between the analysis of the collected data and the used model which describes the data. Some boundaries can be unclear or some assumptions wrong. The gained knowledge can be used to update the used model. In the following, different arguments for a validation system like VortoFlow are discussed.

Alert frequency

Like all alerting systems, special tuning is needed in order to not overwhelm someone with alerts. If the range of a value is often out-of-range, but this is a usual behaviour, then there is something wrong with the used model. The approach which is presented in this thesis only shows how a basic validation concept can be implemented. Depending on the use case, it could be helpful to allow certain rule violations. Also, the system which receives the error messages from VortoFlow can support the handling of errors. The system can group and prioritise errors.

Attack surface reduction

In the history of software engineering, a lot of possible attacks were due to missing validations of incoming data. Stack overflows are one common example. With VortoFlow, the attack surface can be reduced. At the start of a processing chain, it drops all incoming messages with invalid formation and unexpected fields. This can help to increase security. The downside of this is that additional computation time is required to run the validation checks for each message. This requires the user to find the right balance between security and additional work.

Filter incorrect data

For predictive maintenance machine learning can be used. Such an approach can, for example, learn from historical data and maybe detect potential anomalies in future data streams. Someone can argue that VortoFlow filters such data and prevents the use of

such a system. However, this does not significantly depend on the used architecture for the data processing pipeline. VortoFlow's main goal is to detect errors in the beginning of a processing pipeline. This data could cause errors in a later stage. Maybe this approach has the potential to detect all obvious errors and save costs. The raw data or messages with errors can still be collected for later use.

VortoFlow vs. JSON-Schema

In Section 4, it was highlighted that Vorto currently only has a subset of the functions of JSON-Schema. For example, it is not currently possible to set the minimum or maximum length of a string. This is a disadvantage of Vorto. It should be possible to extend the Vorto DSL with all available constraints that JSON-Schema validation offers. It requires only minor changes to Xtext and Ecore model. Implementing these constraints in VortoFlow could be done in a manageable time.

Test and validate software

When starting to develop an IoT solution, often no deeper knowledge about the behaviour of the software and hardware exists. To validate as much as possible, unit tests and models can be used. However, some cases are first discovered when multiple devices are in the field. It can be argued that an analysis of the collected data is necessary, but with the help of VortoFlow the general data cleaning steps may be easier. Also, the structure and some semantics are already provided by Vorto and could support the data analysis and testing of the software.

Creating rules

It can be faster to deploy manual checks if an error was detected after release. In this case, it is necessary that all team members are aligned and follow the MDD approach. No code for data validation should be written if the task can also be achieved with VortoFlow. Updating the model should have a higher priority than writing code. This can prevent historical legacy issues. Often such hot-fixes are not documented and are only known by a limited group of people.

6 Conclusion

This section gives a short summary of the results of this thesis and which goals where achieved. The section closes with an outlook on the potential future development of VortoFlow.

6.1 Summary of Contributions

This thesis presents an approach to combining a domain model with a stream processing system to validate data streams in IoT applications. The chosen approach with Apache Beam has shown that it is possible to include a domain model based on EMF directly in an Apache Beam pipeline. With help of the Vorto DSL, a structure and constraints can be defined for a message. Each processed message can be validated and, for this purpose, a prototype was developed. This prototype was used in a case study where it is shown that the approach could be successfully applied to real data. The case study has also shown that the outcome depends on how boundaries are defined. It is recommended that the approach is combined with further checks.

6.2 Conclusion

Data validation can be a challenging task. With a high number of devices in the field, the data streams cannot be overseen anymore by a human. It requires the user to filter and combine the data to get a picture of the data. It is near-impossible to validate data by hand. An automated process is indispensable for validating incoming data.
The presented approach shows that it is possible to make use of a domain model for this task. The model can be used alongside code generation for stream data validation. The Vorto DSL allows the user to oversee multiple data-points with a relatively compact syntax. All constraints are directly defined alongside each property. This can result in a complexity reduction.
VortoFlow can be used at the beginning of software development for a device. The time needed to set up the system is minimal. This is an advantage over approaches where the structure of a message is self-defined and rules need to be manually programmed. The resulting data validation is currently not very complex but is already able to detect different types of errors. This sets a base for further extension. The results of this thesis could potentially raise the development speed and maintainability of IoT applications.

6.3 Future Work

There is a lot of potential in using a domain model for IoT application in stream processing. This thesis has only focused on the general possibility and how the validation of incoming sensor data could be done. For this purpose, basic constraints and object structure checks were implemented. Currently, VortoFlow only supports JSON as an input format. In the future, further constraints and relations to the Vorto DSL could be introduced.
Besides the presented approach, the Vorto model could be used for other purposes. For example, a database schema could be derived from the model. This could result in a better maintainability through redundancy avoidance. Also, interoperability could be enhanced. Instead of writing code individually for different parts of the processing pipeline of an IoT application, Vorto could be used as a basis. In the following, there are ideas as to how the prototype could be extended and Vorto could be used.

Dependencies between Information Models

Currently, an Information Model represents a real-world device. Such a device can be a smart bulb or a complex device with multiple sub devices. The Vorto DSL does not currently allow the user to define relations between Information Models. Such a relation could be useful in creating logical connections between devices and validating them. A system could have a power meter and a bulb connected to this meter. With a simple mathematical model, it could be validated so that, if the bulb is turned on, the power consumption needs to be bigger than zero. Depending on the extension of the model, more complex relations of properties could be validated.

Include a model from the Vorto model repository

Currently, the Vorto model repository (see Section 2.6.3) is not used by VortoFlow. Generally, it would be useful if VortoFlow only required the user to configure the identifier and the version of a model. VortoFlow could automatically fetch the model from the Vorto repository. A message could also provide an identifier to dynamically load the according model from the repository, but this could have drawbacks in performance.

Programming logic and mathematical expressions

For more sophisticated validation tasks, it is useful to define constraints as mathematical expressions or use logic operators like *if* and *else*. With the help of these methods, relations between different properties could be established. A property could be marked

as optional, but this property depends on other properties. These other properties are also optional. If the property is used all depending properties are then marked as mandatory.

Simulation

The Vorto DSL provides a well-defined, machine-readable description for real-world devices. This allows the user to describe the potential functions and data output of a device. With this knowledge, it would be possible to emulate a device. The model can be used to generate a random output for each model. This could help to test and develop other parts of the data pipeline without the real hardware. Such an emulation could make use of existing datasets and replay those.

Message compression

For devices with restricted bandwidth it is necessary to keep messages as small as possible. Sometimes the user is additionally restricted to use JSON as a message format. With Vorto, an adapter could be created that compresses the JSON. All property names from the model are replaced with the shortest names possible without collision. On the Cloud side, VortoFlow could be extended to restore all original property names. Further compression techniques could also be used. The benefit is that the compression would be transparent for the user.

Nomenclature

API	Application Programming Interface
AST	Abstract Syntax Tree
CSV	Comma-Separated Values
DSL	Domain-Specific Language
EMF	Eclipse Modelling Framework
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
MBSD	Model-Based Software Development
MDD	Model-Driven Development
MDSD	Model-Driven Software Development
SDK	Software Development Kit
SQL	Structured Query Language
UML	Unified Modelling Language
XML	Extensible Markup Language

References

[1] Gartner, Inc. (2017). Gartner says 8.4 billion connected 'Things' will be in use in 2017, up 31 percent from 2016, [Online]. Available: `http://www.gartner.com/newsroom/id/3598917` (visited on 05/31/2018).

[2] A. Klein and W. Lehner, 'Representing data quality in sensor data streaming environments', *J. Data and Information Quality*, vol. 1, no. 2, 10:1–10:28, 2009.

[3] R. Y. Wang and D. M. Strong, 'Beyond accuracy: What data quality means to data consumers', *J. of Management Information Systems*, vol. 12, no. 4, pp. 5–33, 1996.

[4] I. Taleb, H. T. E. Kassabi, M. A. Serhani, R. Dssouli, and C. Bouhaddioui, 'Big data quality: A quality dimensions evaluation', in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, July 18-21, 2016*, 2016, pp. 759–765.

[5] M. Wagner, J. Laverman, D. Grewe, and S. Schildt, 'Introducing a harmonized and generic cross-platform interface between a vehicle and the cloud', in *17th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks, WoWMoM 2016, Coimbra, Portugal, June 21-24, 2016*, 2016, pp. 1–6.

[6] J. Laverman, D. Grewe, O. Weinmann, M. Wagner, and S. Schildt, 'Integrating vehicular data into smart home IoT systems using Eclipse Vorto', in *IEEE 84th Vehicular Technology Conference, VTC Fall 2016, Montreal, QC, Canada, September 18-21, 2016*, 2016, pp. 1–5.

[7] P. Patel, B. Morin, and S. Chaudhary, 'A model-driven development framework for developing sense-compute-control applications', in *1st International Workshop on Modern Software Engineering Methods for Industrial Automation, MoSEMInA 2014, Hyderabad, India, May 31, 2014*, 2014, pp. 52–61.

[8] D. Cassou, B. Bertran, N. Loriant, and C. Consel, 'A generative programming approach to developing pervasive computing systems', in *Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings*, 2009, pp. 137–146.

[9] C. Xu, D. Wedlund, M. Helgoson, and T. Risch, 'Model-based validation of streaming data: (industry article)', in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, 2013, pp. 107–114.

[10] A. B. Sharma, L. Golubchik, and R. Govindan, 'Sensor faults: Detection methods and prevalence in real-world datasets', *TOSN*, vol. 6, no. 3, 23:1–23:39, 2010.

[11] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*. Springer, 2005, ISBN: 978-3-540-25613-7.

[12] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development - technology, engineering, management*. Pitman, 2006, ISBN: 978-0-470-02570-3.

[13] H. Lochmann, 'Hybridmdsd: Multi domain engineering with model-driven software development using ontological foundations', PhD thesis, Dresden University of Technology, 2009.

[14] D. Spinellis, 'Notable design patterns for domain-specific languages', *Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, 2001.

[15] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Birmingham: Packt Publishing Ltd, 2016, ISBN: 978-1-78646-496-5.

[16] Eclipse Foundation. (2015). Eclipse project proposal for Vorto, [Online]. Available: `https://projects.eclipse.org/proposals/vorto` (visited on 05/31/2018).

[17] A. Edelmann *et al.* (2018). Vorto Readme, [Online]. Available: `https://github.com/eclipse/vorto/blob/development/Readme.md` (visited on 05/31/2018).

[18] G. DeMichillie. (2014). Cloud platform at Google I/O - new big data, mobile and monitoring products, [Online]. Available: `https://developers.googleblog.com/2014/06/cloud-platform-at-google-io-new-big.html` (visited on 05/31/2018).

[19] F. Perry and J. Malone. (2016). Dataflow and open source - proposal to join the Apache Incubator, [Online]. Available: `https://cloudplatform.googleblog.com/2016/01/Dataflow-and-open-source-proposal-to-join-the-Apache-Incubator.html` (visited on 05/31/2018).

[20] The Apache Software Foundation. (2018). Apache Beam - design your pipeline, [Online]. Available: `https://beam.apache.org/documentation/pipelines/design-your-pipeline/` (visited on 05/31/2018).

[21] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, 'The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing', *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[22] A. Wright, H. Andrews, and G. Luff, 'JSON schema validation: A vocabulary for structural validation of JSON', IETF Secretariat, Internet-Draft draft-handrews-json-schema-validation-01, Mar. 2018, https://tools.ietf.org/html/draft-handrews-json-schema-validation-01.

[23] T. A. S. Foundation. (2018). Apache Beam programming guide, [Online]. Available: https://beam.apache.org/documentation/programming-guide/ (visited on 05/31/2018).

[24] S. Madden *et al.* (2004). Intel lab data, [Online]. Available: http://db.csail.mit.edu/labdata/labdata.html (visited on 05/31/2018).

[25] Crossbow Technology, Inc. (2004). Mote hardware session, [Online]. Available: https://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/PresentationOverheads/Day1_Sect03_Hardware.pdf (visited on 05/31/2018).

[26] Sensirion Inc. (2011). Datasheet SHT1x (SHT10, SHT11, SHT15) humidity and temperature sensor IC, [Online]. Available: https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/0_Datasheets/Humidity/Sensirion_Humidity_Sensors_SHT1x_Datasheet.pdf (visited on 05/31/2018).

[27] TAOS Inc. (2004). TSL2550 ambient light sensor with SMBus interface, [Online]. Available: https://upverter.com/datasheet/57011d7e45248fad7bad9be4e94bd6ea20b8d575.pdf (visited on 05/31/2018).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 14. Juni 2018 .....