Supporting AI engineering on the IoT edge through model-driven TinyML

Supporting AI Engineering on the IoT Edge through Model-Driven TinyML

Armin Moin
Department of Informatics
Technical University of Munich, Germany
moin@in.tum.de

Moharram Challenger
Department of Computer Science
Univ. of Antwerp & Flanders Make, Belgium
moharram.challenger@uantwerpen.be

Atta Badii
Department of Computer Science
Univ. of Reading, United Kingdom
atta.badii@reading.ac.uk

Stephan Günnemann
Dep. of Informatics & Munich Data Science Institute
Technical University of Munich, Germany
guennemann@in.tum.de

***Abstract*—Software engineering of network-centric Artificial Intelligence (AI) and Internet of Things (IoT) enabled Cyber-Physical Systems (CPS) and services, involves complex design and validation challenges. In this paper, we propose a novel approach, based on the model-driven software engineering paradigm, in particular the domain-specific modeling methodology. We focus on a sub-discipline of AI, namely Machine Learning (ML) and propose the delegation of data analytics and ML to the IoT edge. This way, we may increase the service quality of ML, for example, its availability and performance, regardless of the network conditions, as well as maintaining the privacy, security and sustainability. We let practitioners assign ML tasks to heterogeneous edge devices, including highly resource-constrained embedded microcontrollers with main memories in the order of Kilobytes, and energy consumption in the order of milliwatts. This is known as TinyML. Furthermore, we show how software models with different levels of abstraction, namely platform-independent and platform-specific models can be used in the software development process. Finally, we validate the proposed approach using a case study addressing the predictive maintenance of a hydraulics system with various networked sensors and actuators.**

***Index Terms*—model-driven software engineering, domain-specific modeling, machine learning, tinyml, edge analytics, internet of things**

I. Introduction

Finding IT professionals, e.g., software developers and system engineers who are familiar with the diverse hardware and software platforms, programming languages, communication protocols and APIs that are involved in the Internet of Things (IoT) applications is very difficult. The technologies are diverse and the platforms have a broad spectrum, ranging from tiny sensors and microcontrollers with a few Kilobytes (KB) of main memory and very constrained power and processing resources to capable cloud servers with multiple GPUs and large in-memory databases. Consequently, their operating systems (if any), programming languages and communication protocols are also very different. For instance, a single IoT project might require familiarity with various operating systems, such as Linux, TinyOS, ContikiOS, ROS, Android, iOS as well as machine codes (machine languages) or assembly languages of multiple microcontrollers and computers with various architectures. Moreover, different programming languages will be used for different parts of the software systems that need to be deployed on distributed platforms. C, Java (J2SE and J2EE), Python, PHP, Go and Javascript are only a few examples of the common choices. In addition, communication protocols are diverse on different layers of the network stack. For example, on the application layer, CoAP (Constrained Application Protocol) and MQTT (Message Queuing Telemetry Transport) are more suitable protocols than HTTP (Hypertext Transfer Protocol) for resource-constrained devices. The former (CoAP) is suitable for one-to-one communications, whereas the latter (MQTT) is designed for many-to-many communications following the publish-subscribe pattern. Therefore, no matter how professional and skilled a software developer or system engineer is, they can neither master the entire technology spectrum and cross-domain functions of a complex IoT system nor can they efficiently communicate and collaborate with other experts working on the same project.

The Model-Driven Software Engineering (MDSE) paradigm, also known as Model-Based Software Engineering (MBSE), specifically the Domain-Specific Modeling (DSM) methodology with full code generation [1] offers abstraction and automation to deal with the above-mentioned complexity. Over the past decade, its applications have been expanded from the niche domains, such as embedded systems for safety critical applications, e.g., in the automotive industry, to the more complex domains, such as the Internet of Things (IoT) [2] with highly heterogeneous, distributed systems of systems, called Cyber-Physical Systems (CPS) [3], [4]. Examples of tools offering solutions for domain-specific MDSE of embedded systems comprise MATLAB Simulink [5] and AutoFOCUS [6]. Moreover, ThingML [2], [7]–[9], HEADS [10], [11] and μ-Kevoree [12] supported domain-specific MDSE for the IoT. These solutions enabled full source code generation in an automated manner.

However, today's software systems that support IoT services require more smart capabilities, as well as more dynamicity

at runtime. There is a trend towards data-driven software and systems design and modeling. This handles uncertainty at the software design-time. In other words, the modeler or designer of an IoT service that requires AI, or more specifically ML, may postpone certain design-time decisions and leave them to the predictions, recommendations or outputs of trained ML models at the runtime. Previous work exists in the literature, e.g., ML-Quadrat [13], [14] that introduced this. Nevertheless, their major drawback was that they did not support deploying ML models on the IoT edge devices. In contrast, for many IoT use cases today, the deployment of compact ML models that can function self-sufficiently on resource-constrained microcontrollers is necessary.

For instance, a modern smartphone that runs an iOS or Android operating system has a Digital Signal Processor (DSP) chip with power consumption in the order of only a few milliwatts (mW). This microprocessor chip has the task of continuously listening to the surrounding environment for a predefined statement in the form of a speech command, i.e., a so-called *wake word*, such as *'Hey Siri'* or *'OK Google'*. Note that this always-on DSP must be separated and independent of the main CPU so that the smartphone can save the battery by letting its main CPU that consumes considerably more energy stand by in an inactive mode as long as possible. The trained Artificial Neural Network (ANN) ML model that can efficiently perform the mentioned task on such a DSP has a size of about 14 KB. There are many further scenarios beyond the above-mentioned example. For instance, Park et al. [15] proposed an ANN model with the size of around 15 MB for enhanced, real-time, automatic speech recognition on smartphones and embedded devices. In contrast to the previous case, where speech recognition was only needed for a simple wake word, this model is much more capable in terms of speech recognition. However, the advanced capability of this model comes at a cost: Its size is more than one thousand times larger than the aforementioned wake word recognizer ANN model. Thus, it cannot fit into the memory of typical TinyML devices.

Furthermore, in other IoT use cases, such as predictive maintenance, the so-called *peel-and-stick sensors*, which require no battery change over their lifetime, or the *Everactive wireless sensors*, which are exclusively powered by low levels of harvested energy from the surrounding environment are increasingly becoming prevalent. These tiny IoT sensors are either already AI-enabled or are expected to become so in the future [16].

In this paper, we propose a novel approach to domain-specific MDSE of smart services that will run on heterogeneous and distributed IoT devices. In particular, we advocate edge computing (fog computing), specifically edge analytics and TinyML. As mentioned, the latter involves delegating ML tasks to highly resource-constrained microcontrollers with ultra-low energy consumption levels. This lets the data remain at the edge of the network (i.e., on the users' side) and be processed there instead of being transferred to the other nodes of the distributed system, e.g., to the cloud. One of the main drivers for this paradigm shift at the present time is of privacy concerns and the need to ensure legal compliance assurance by design, e.g., concerning the General Data Protection Regulation (GDPR) in the European Union (EU) and the California Consumer Privacy Act (CCPA) in the United States (US) [17]. In addition, the device energy efficiency, the network throughput optimization and the availability of the service or certain functionalities irrespective of the network conditions will be other benefits of the said transition regarding the execution of data analytics and ML on the edge devices.

Our main Research Questions (RQ) are the following: **RQ1.** Can we enable automated full code generation out of the software models of smart IoT services that will deploy trained ML models on highly resource-constrained IoT edge platforms (i.e., realizing TinyML)? **RQ2.** Can we have a higher level of abstraction for the Platform-Independent Models (PIM) that will abstract from the details and constraints of the underlying IoT platforms, and simultaneously a lower level of abstraction for the Platform-Specific Models (PSM) out of which the full implementation must be generated? Note that PIM and PSM here refer to the software and system models, not the data analytics and ML models. Hence, the contribution of this paper is two-fold: (i) It assesses RQ1 and enables TinyML in the domain-specific MDSE of smart IoT services. (ii) It assesses RQ2 and provides the PIM and the PSM layers for the software models of smart IoT services.

The rest of this paper is structured as follows: Section II offers some background information. Then, we review the state of the art briefly in Section III. We propose our novel approach in Section IV. Further, Section V implements and validates the proposed approach. Finally, we conclude and suggest future work in Section VI.

II. Background

In this section, we provide some required background information on MDSE and TinyML.

A. MDSE

There exist different approaches to the MDSE paradigm. The Model-Driven Architecture (MDA) standard of the Object Management Group (OMG) [18], which was initially issued in 2001 and then updated in 2014, serves as a key reference for MDSE. According to MDA, three default architecture viewpoints are defined for every system: computation-independent, platform-independent and platform-specific. Computation-Independent Models (CIM) are business or problem-domain models. They use the vocabulary that is familiar to the subject matter experts in the respective domains. However, Platform-Independent Models (PIM) are solution-domain models, namely models that are related to the computational concepts. Nevertheless, they abstract from the details of any specific platform. Further, Platform-Specific Models (PSM) augment PIMs with the details that are specific to particular platforms. In MDA, the requirements specified in a CIM must be traceable to the constructs in the PIM and the PSMs that implement them (and vice-versa) [19].

However, efficient full code generation that does not require any further manual development is often not feasible with MDA. MDA is rather generic and broad. Moreover, the round-tripping processes (i.e., the model transformations from the CIM to the PIM and then the PSMs and vice-versa) result in many model artifacts that need to be managed and might not be consistent over the time. In contrast, the Domain-Specific Modeling (DSM) methodology [1] that is adopted and adapted in this work promotes narrowing the domain of interest down and also avoiding round-tripping. Models are very specific to a particular use case and the full implementation of the solution is generated out of the models, namely PSMs in the MDA terminology. In this paper, we use both PIM and PSM, but avoid round-tripping. A PSM in our software development methodology is simply an extension of a PIM with the platform-specific *annotations* and *configurations* that are necessary for the code generation for a certain target IoT platform out of the PSM. By *a particular platform*, we mean the combination of the hardware architecture, the operating system if applicable, the programming language, the libraries and APIs, as well as the communication protocols.

B. TinyML

Running Machine Learning (ML) tasks, such as making predictions using ML models on embedded devices is a young field. In the context of the IoT, i.e., for the networked embedded devices, this is inline with the trend towards assigning more computational tasks to the edge of the network as opposed to the cloud. The trend is known as edge computing or fog computing. The IoT edge devices reside on the users' side and can range from desktop PCs, laptops, tablets and smartphones to embedded single-board computers, such as Raspberry Pi and embedded microcontrollers. In the case of deploying pre-trained ML models on the resource-constrained microcontrollers with ultra-low power consumption in the range of 1 mW, we are dealing with TinyML. These microcontrollers possess main memories (RAM) in the order of tens to hundreds of kilobytes. In addition, their persistent flash memories can be in the order of kilobytes to megabytes. Moreover, their CPU clock speeds might be as low as just tens of MHz. Hence, they are small and highly energy efficient. Last but not least, they are relatively inexpensive and can be ordered in large quantities.

In this work, we use an *Arduino Nano 33 BLE Sense* microcontroller board [20] with an ARM® Cortex®-M4 32-bit processor with a clock speed of 64 MHz, 256 KB RAM, 1 MB flash memory and various on-board sensors, e.g., for the temperature, humidity, pressure, brightness, vibration, etc. for the TinyML platform. In contrast, another target platform for code generation will be an embedded single-board computer, namely a *Raspberry Pi 3 B+* board [21] with an ARM® Cortex-A53 (ARMv8) 64-bit SoC (System on Chip) that has a clock speed of 1.4 GHZ, as well as 1 GB of main memory. Although this does not fall under the category of the TinyML platforms, it will be used to demonstrate the heterogeneity of the target platforms for the fully-automated code generation out of the software models.

III. State of the Art

As set out in Section I, ThingML [2], [7]–[9] and HEADS [10], [11] supported the MDSE paradigm, specifically the DSM methodology [1] for full code generation in the IoT/CPS domain. They were based on the Eclipse Modeling Framework (EMF) [22] and the Xtext framework [23]. While they mainly focused on the design-time of software systems, other approaches, such as μ-Kevoree [12] concentrated on *Models@Runtime*, thus conflating the two distinct phases of design (modeling) and execution of IoT services. The major shortcoming of all the said approaches was the lack of DAML support at the modeling level. In other words, the users of those DSMLs might not deploy the APIs of DAML libraries and frameworks in their software models. Hence, there was no seamless integration between the software models and the DAML models.

GreyCat [24] by Hartmann et al. [25]–[27] and ML-Quadrat [14] by Moin et. al [13], [28], [29] filled this gap in. However, they fell short of supporting edge analytics and TinyML on resource-constrained IoT devices. The former, which was based on the Kevoree Modeling Framework (KMF) [30], [31] and μ-Kevoree [12] could generate Java and Javascript/Typescript code. This was not sufficient for many IoT use case scenarios that involved resource-constrained devices that were incapable of executing the Java Virtual Machine (JVM) for the backend. Finally, the latter offered code generation for a range of platforms, including C code generation for various resource-constrained microcontrollers. However, the analytics and ML part had to run in the cloud or any node in the distributed system that was not resource-constrained. Table I shows a comparison of the proposed approach with the related work in the literature.

TABLE I
Relating the proposed approach to the previous work

Approach	**Basis**	**ML-enabled**	**Resource-constrained edge**	**TinyML**
ThingML [9] / HEADS [11]	EMF [22] & Xtext [23]	-	✓	-
μ-Kevoree [12]	KMF [31]	-	✓	-
GreyCat [24]	μ-Kevoree [12]	✓	-	-
ML-Quadrat [14]	ThingML [9]	✓	✓	-
The proposed approach	ML-Quadrat [14]	✓	✓	✓

Finally, the proprietary software tooling provided by MathWorks® Inc., which comprises MATLAB®, Simulink®, ThingSpeak™ and other MATLAB® add-ons, is widely used in the industry. However, the technology stack is based on the MATLAB® ecosystem with full code generation in several

programming languages, such as C/C++, HDL, .NET and Java, and with integrated APIs for relational and non-relational databases, as well as several communication protocols, e.g., REST, MQTT and OPC Unified Architecture (OPC UA) for both offline data (i.e., batch processing) and online data (i.e., stream processing). Note that the proprietary solution is expensive, thus a potential barrier for open innovations. In contrast, the solutions listed in Table I are open source with permissive licenses that enable their cost-effective deployment and possible future extensions. For instance, μ-Kevoree [12] was the basis for GreyCat [24], ThingML [9] and HEADS [11] were the basis of ML-Quadrat [14], and the present work builds on top of ML-Quadrat [14]. It is clear from the table that none of the prior work has addressed the TinyML platforms in its target IoT platforms for code generation.

IV. Proposed Approach

We propose a novel approach to software and AI engineering for smart IoT services that enable their data analytics and ML parts to run in part or completely on the IoT edge devices that might be highly constrained in terms of their power and computational resources. The proposed approach is based on the DSM methodology [1] of the MDSE paradigm for software development. In particular, it builds on the prior work ML-Quadrat [13], [14], thus integrates model-driven software and AI (specifically ML) engineering. As shown in Figure 1, we generate code for heterogeneous IoT platforms out of the Platform-Specific Models (PSMs). However, we also offer generic models that are similar to the PIM viewpoint models in MDA [18], [19]. A PIM here can be considered as the *common denominator* of the PSMs for a particular IoT service that must run on several heterogeneous IoT platforms. PIMs are at a higher level of abstraction than PSMs and specify the business logic of the IoT services regardless of the platform-specific details. Hence, the practitioner may concentrate on the overall, platform-independent structural and behavioral design of the software system architecture without any concerns about the possible lack of knowledge and skills in the diverse hardware, software, network and AI technologies that are used in the heterogeneous IoT edge devices and in the cloud.

Formally, we define a platform-independent software model for a smart IoT service as follows (see Equation 1):

$$\text{PIM} = (\Psi, M_{\text{ML}}, B) \tag{1}$$

In Equation 1, Ψ represents the structural elements of the software architecture model. Thus, it can be denoted, for example, by a component diagram. Figure 2 illustrates a sample UML component diagram for a predictive maintenance service. In addition, M_{ML} is the ML model that brings AI to the IoT service. For instance, it can be an Artificial Neural Network (ANN), a Support Vector Machine (SVM) or a random forest ML model. Also, B represents the behavioral elements of the software model[1]. Hence, it can be denoted, for example, by a state machine diagram. Figure 3 demonstrates a sample state machine or statechart for the behavioral model of an IoT sensor that can also conduct anomaly detection via ML as its *TinyML service*, in addition to its measurement service.

[1]Note that M_{ML} might affect B.

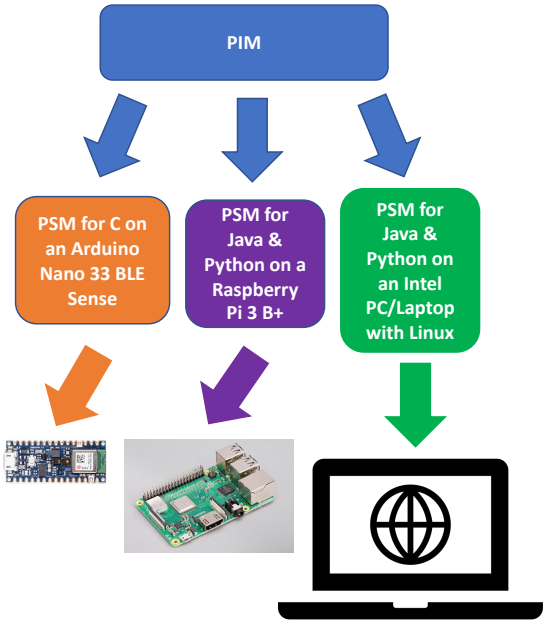

Fig. 1. From PIM to PSMs and full code generation. The images of Arduino and Raspberry Pi are from [20] and [21], respectively.

Furthermore, we define a platform-specific software model for a smart IoT service as in Equation 2 below:

$$\text{PSM} = (\text{PIM}, A, C) \tag{2}$$

Here, A and C stand for the platform-specific annotations and configurations, respectively. *Annotations* can be attached to various elements of model instances to add platform-specific details. For instance, they may help model-to-code transformations (code generators) in mapping the data types to the right ones in the target platforms. This means, they can, for example, specify whether an Integer data type in the software model instance must be mapped to the *int*, *short* or *long* type in the generated Java code. Furthermore, *configurations* are required in order to make model instances ready for code generation out of them. Configurations must include object instances of the *thing* classes and specify their interconnections for message-passing. In addition, they may include annotations that specify the target platform for code generation. In order to create a PSM out of a PIM, one must insert annotations and append configurations to the PIM for the respective target platform for code generation. We show this in Section V. Note that currently both PIM and PSM instances conform to the same meta-model that is adopted from ML-Quadrat [13], [14]. Figure 4 illustrates part of this meta-model.

Moreover, we provide model-to-code transformations (i.e., code generators) that can produce the entire software solution for the desired smart IoT services (see the orange, purple and green arrows in Figure 1). In the present work, we

implement the code generator for the TinyML part that can generate the APIs of *TensorFlow Lite* [32] and *TensorFlow Lite for Microcontrollers* [33]. For the validation case study in Section V, we deploy the former on a *Raspberry Pi 3 B+* [21] and the latter on an *Arduino Nano 33 BLE Sense* microcontroller [20]. In addition, other code generators from prior work, e.g., ML-Quadrat [13], [14] can be used to generate code for a Linux PC/laptop with an Intel x86 CPU. In this case, the Keras [34] API for TensorFlow [35] will be generated and used for the ML part.

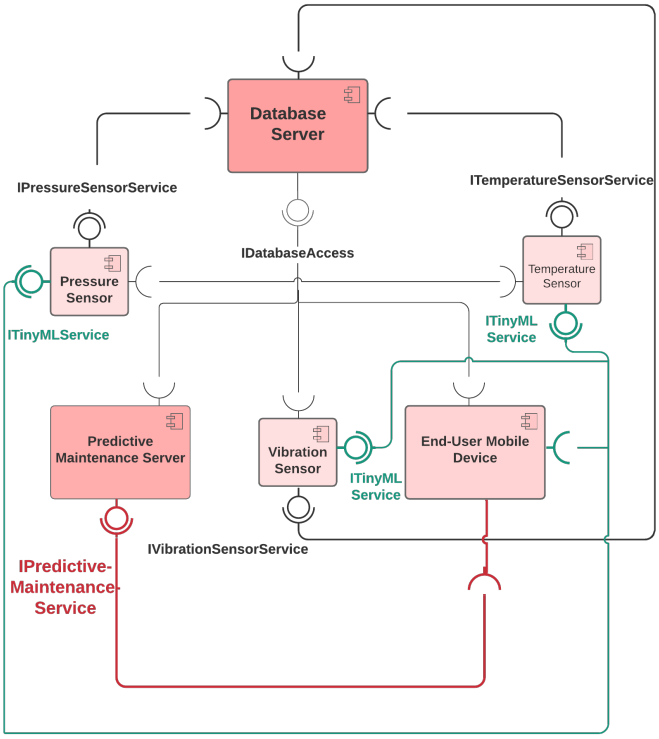

Fig. 2. The UML component diagram illustrating the structural architecture model of a sample IoT service for predictive maintenance.

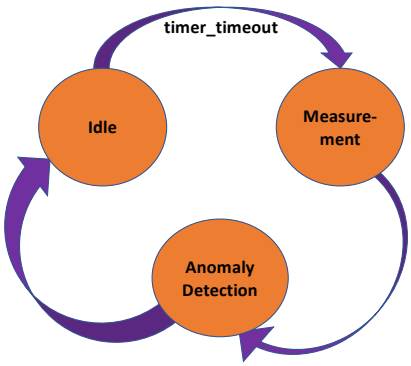

Fig. 3. The state machine diagram illustrating the behavioral architecture model of the sensors in the sample IoT service of Figure 2.

The proposed approach enables deploying pre-trained ML models on various distributed edge devices without sharing

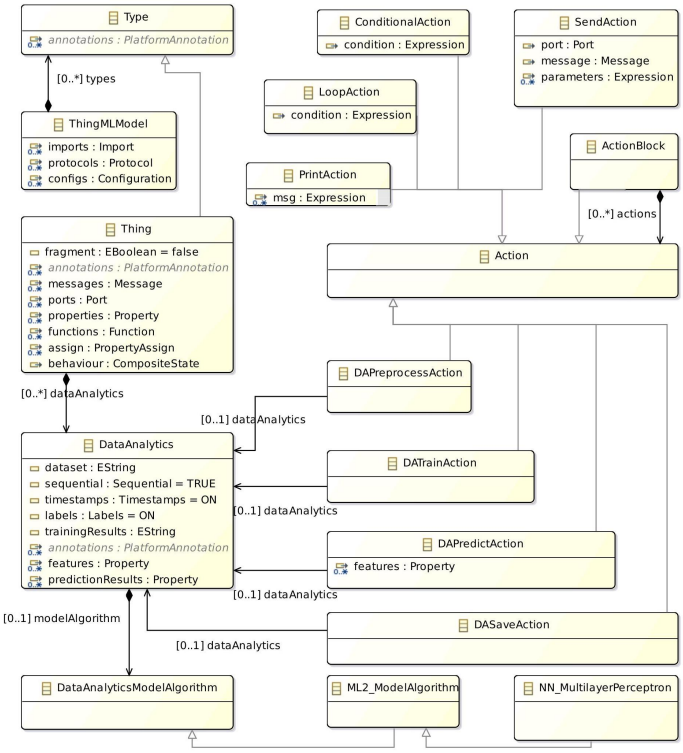

Fig. 4. Part of the meta-model that is adopted from ML-Quadrat [13].

data. This means, ML models, such as ANNs can be created and trained with existing data on a server, e.g., on-premises or in the cloud. Then, depending on the size of the trained ML models, the requirements and the capabilities of the IoT devices, some of the ML models may be deployed on TinyML platforms, and some other ML models may be deployed on other edge devices, such as smartphones, smart home appliances and gateways, or in the cloud. One key benefit of bringing the data analytics and ML models to the edge of the network is the ability to respect the possible privacy and security concerns or regulations. Instead of asking the users to upload their new data to the cloud and making predictions there, we let the users keep their new data on their side and enable predictions there.

In the context of the predictive maintenance use case that is illustrated in Figure 2 and Figure 3, one might think of $m + 1$ ML models to be deployed on m sensors and on one server or controller. The latter will be more capable in terms of the computational power, thus can run a more advanced ML model and take more data into account, whereas the former will be restricted ML models for carrying out simpler tasks on a local level. In other words, each sensor might, for example, process the data that come from its own measurements and possibly its neighbors in the sensor network, whereas the server or controller node that must be more capable conducts a more extensive condition-based monitoring of the entire system. This means, the predictive maintenance operations can be executed on the local, i.e., sensor and global levels. These are shown via the green and the red lines in Figure 2,

respectively. This *federated* design[2] is expected to increase the availability of the services, e.g., condition-based monitoring, and contribute to the fault tolerance and the overall resilience of the system since multiple nodes will perform the ML task independently and with different qualities according to their resources. If a network or power outage occurs in one part of the system, other nodes can still deliver some level of service.

In addition, the network throughput might be reduced by letting individual sensors perform some basic analytics tasks locally, thus reducing the frequency and the amount of the data that needs to be sent to the database and/or the server/controller. Further, in certain applications, the TinyML operations on the resource-constrained nodes might suffice, thus resulting in a much lower level of energy consumption for the data analytics and ML tasks. This should lead to environmental care and sustainability in the long term. Finally, by deploying ML models on the TinyML devices with ultra-low power consumption, AI/ML can become more affordable, and can also be brought to the situations where Internet connectivity is not possible and/or to the extreme conditions where sensors that will be mounted somewhere, e.g., under the ground, in the oceans or on very large structures must run on their limited battery powers for a relatively long time and are not physically accessible in a cost-effective manner after their initial installation.

V. Implementation & Validation

We implement the proposed approach by extending the ML-Quadrat [14] project. First, we adopt the Xtext-based meta-model (grammar) of ML-Quadrat [14]. Second, to support the new target platforms, namely Raspberry Pi 3 B+ [21] and Arduino Nano 33 BLE Sense [20], we introduce new annotation types for *configurations* that enable practitioners to choose the said platforms as target platforms for code generation. We extend the model-to-code transformations, i.e., the code generators of ML-Quadrat [14] to support full code generation for the said platforms. To this aim, we deploy the APIs of the TensorFlow Lite [32] and TensorFlow Lite for microcontrollers [33] libraries. In the former case, the generated code is in Python, whereas in the latter case it is C code for Arduino. The code generation process also includes the conversion of ML models to the proper formats that are acceptable by the mentioned libraries. In the latter case, namely the microcontroller, this format is a hexadecimal dump of a C array that is stored in a C source file. Further, the code generators themselves are implemented in Java.

In the following, we validate the proposed approach through a case study. There exists a hydraulic test rig that is deployed in Saarbrücken, Germany [36]. A test rig or test station is used to test and assess the capability and performance of components for industrial use [37]. The hydraulic test rig is equipped with multiple sensors and the sensor data, as well as the data about the working conditions and status of the system is provided by the ZeMA gGmbH research center for Mechatronics and automation technologies as open data [38]. We use the data from the following 3 sensors in order to predict any possible internal leakage of the main pump: (i) The vibration sensor (VS1) of the main pump. Its readings are recorded in the mm/s unit and at the frequency of 1 Hz (i.e., once a second). (ii) The Electrical Power Sensor (EPS1) of the main pump. Its readings are recorded in Watts and at the frequency of 100 Hz. (iii) The System Efficiency (SE) factor that is not a real (i.e., physical) sensor, but a virtual sensor. Its values are determined by combining different directly measured values [36]. Moreover, it is a percentage and has the frequency or sampling rate of 1 Hz. Since the hydraulic test rig repeats periodic constant load cycles of 60 seconds, we require the sensor data for one cycle in order to predict whether the main pump is prone to any internal leakage or not. We let an Artificial Neural Network (ANN) model perform this prediction. The ML features that are used to train this model are the above-mentioned sensor values, namely VS1, EPS1 and SE. As the sampling rates or frequencies are not identical, one could, for example, down-sample EPS1 that has a frequency of 100 Hz to 1 Hz. However, for the current use case, we keep it as it is. Therefore, there exist 60 features for VS1, 6,000 features for EPS1 and another 60 features for SE per system cycle. We assign one Boolean/Binary class label to each cycle that is either True (i.e., leakage positive) or False (i.e., leakage negative). The dataset contains 2,205 data instances, i.e., system cycles. Out of these 2,205 instances, 1,221 instances/cycles (55%) correspond to no leakage and the rest corresponds to leakage.

Since we are dealing with time series data in which the order of the data instances matters, we avoid shuffling the data samples/instances. We separate the dataset into two parts. We dedicate 80% of the available data to the training and validation dataset and the rest to the test dataset for the evaluation. The latter must remain unseen by the ML model to make a fair evaluation possible. The choice of 80% vs. 20% is a common practice in ML and also aligns with the Pareto principle that is widely used in science and engineering.

We standardize the numeric training data using Z-Scores and train an Artificial Neural Network (ANN) model using the data. It transpires that a *Multi-Layer Perceptron (MLP)* with three layers (input, hidden and output) accomplishes the prediction task with high accuracy, precision and recall. The hidden layer is a *Dense* layer with 32 units in the first experiment and 8 units in the second experiment below, as well as the *Relu* activation function. Further, the output layer has 2 units and the *Sigmoid* activation function. Moreover, we use the *Adam* optimizer, the *Binary Crossentropy* loss function, a learning-rate of *1e-5*, a batch size of 100, 200 training epochs, as well as early-stopping with a patience level of 3. Figure 5 depicts the changes of the loss function and the binary accuracy during the training of the ML model.

We could deploy more complex and advanced ANN architectures, e.g., Recurrent Neural Networks (RNN), such as Long-Short Term Memory (LSTM) layers. However, since the

[2]Here, federated should not be confused with the notion of *federated learning* [17] in which an ML model is built collaboratively in a distributed system without sharing data among the participants.

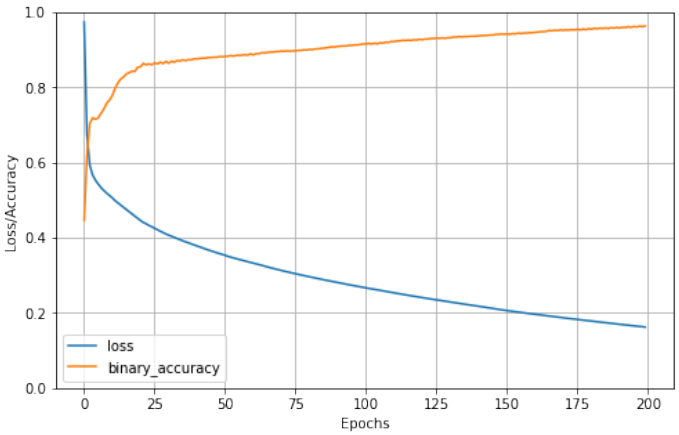

Fig. 5. The values of the loss function and the binary accuracy during the training of the ML model.

mentioned architecture already performs well and we prefer a more compact ML model, we leave it like this. According to the experimental results on the test dataset that are illustrated in Table II, the accuracy, precision and recall were 97%, 97% and 97%, respectively, for the first experiment (i.e., with 32 units in the hidden layer of the above-mentioned ML model), and 80%, 86% and 80%, respectively, for the second experiment (i.e., with 8 units in the hidden layer of the above-mentioned ML model) on an Intel x86 platform with the Linux Operating System (OS) and Python code that deploys the TensorFlow [35] library. This Linux server has 45GB of main memory (RAM) and 10 Intel Xeon 2.3 GHz Processors. The respective rows in Table II are colored in gray. As shown, the more compact ML model (namely the latter experiment) performs faster. Thus, it requires only 86 milliseconds for the entire test dataset instead of 119 milliseconds in the first experiment (i.e., 38% time reduction). However, the accuracy and recall have been reduced by 17.5% each, and the precision has fallen by 11.3% in the second experiment with the more compact ML model that is 74.5% smaller in size.

Further, the second and the sixth rows in Table II demonstrate the experimental results for the first and the second experiment on a Raspberry Pi (RPI) platform, respectively. As mentioned, this is a Raspberry Pi 3 B+ [21] board with the Raspberry Pi OS (formerly known as Raspbian) and Python code that deploys the TensorFlow Lite [32] library. This library provides an API for an ML model converter that enables generating an optimized ML model in the FlatBuffers [39] serialization format and the *.tflite* file extension [32]. As we can see in the table, this conversion results in 67% and 68% ML model size reduction in the first and the second experiments, respectively, without compromising the ML model performance in terms of its accuracy, precision and recall. Nevertheless, it is clear that predictions on the RPI platform need more time, due to the limitations of the computational resources, compared to the said Linux server. The increases in the prediction time for the entire test dataset are 1,100% and 694% for the first and the second experiments, respectively.

In addition, we apply a technique, called *post-training quantization* [32] in both experiments and illustrate the results in the third and the seventh rows in Table II. Hence, with a negligible compromise in the ML model performance in terms of accuracy, precision and recall, we reduce the ML model size considerably and speed up its predictions too. For instance, in the case of our first experiment on the RPI platform, we do not face any reduction in the accuracy, precision or recall. Also, in the second experiment, the accuracy and recall remain the same, while the precision is reduced by only 1%. However, the quantization technique results in an ML model size reduction of 75% and 74% in the first and the second experiments, respectively. Note that this quantized ML model makes predictions 60% and 29% faster in the first and the second experiments, respectively. In this case, quantization leads to converting all of the *float32* weights of the ML model to *int8* values.

While both the non-quantized and the quantized variants of the above-mentioned ML model fit into the main memory of the RPI board, for the TinyML platform, namely the Arduino Nano 33 BLE Sense microcontroller [20], the situation is different. To deploy the ML model on this platform, we use the *xxd* Unix/Linux command to generate a hexadecimal dump of the mentioned FlatBuffers model as a C Byte Array. We store the resulting ML model in a C++ source file with the *.cc* extension. This can be used via the TensorFlow Lite for microcontrollers [33] library on the Arduino microcontroller. However, the main issue is that the hexadecimal dump requires more space on the disk than the efficient FlatBuffers serialization. Note that in the case of the first experiment, the C Byte Array has a size of 198 KB (i.e., with 198 thousands elements). However, its hexadecimal dump requires 1.2 MB disk space. This is 506% larger. Since the main memory of the microcontroller has only 1 MB, which is even large compared to many TinyML platforms, we cannot deploy this ML model on the Arduino platform. In fact, this is the reason that we conduct the second experiment with a more compact ML model. Here, we have a C Byte Array of 51 KB (i.e., with 51 thousands elements). Again, the hexadecimal dump requires more space. In this case, it takes 316 KB on the disk. Therefore, it can fit into the main memory of the TinyML platform. The results of both experiments on Arduino are shown on the fourth and the eights rows of Table II.

Recall from Section I that we have 2 Research Questions (RQ): **RQ1.** Can we enable automated full code generation out of the software models of smart IoT services that will deploy trained ML models on highly resource-constrained IoT edge platforms (i.e., realizing TinyML)? **RQ2.** Can we have a higher level of abstraction for the Platform-Independent Models (PIM) that will abstract from the details and constraints of the underlying IoT platforms, and simultaneously a lower level of abstraction for the Platform-Specific Models (PSM) out of which the full implementation must be generated?

To assess and validate the research questions, we realize the above-mentioned use case with our textual Domain-Specific

Modeling Language (DSML) that is based on ML-Quadrat [14] and let the full source code become generated automatically out of the software model instances. The generated code can create, train and deploy the said ML models. To this aim, we implement both PIMs and PSMs. In fact, we need two PIMs and two PSMs for training the ML model on the x86 Linux machine. The two PIMs are responsible for creating and training the ML models of the two experiments, respectively (namely, rows 1-4 and rows 5-8 in Table II). Here, we illustrate the PIM for the first experiment. Figure 6 and Figure 8 show part of the platform-independent and platform-specific software model instances for the training on the x86 Linux server. If we wanted to train the ML model on another platform, such as a Raspberry Pi, we would take the same PIM and import it in another PSM that would have been specific to that platform. The choice of the target platform for code generation is specified through the *@compiler* annotation of the *configurations* (see Figure 8). For instance, *@compiler python_java* generates Python and Java code for the default platform, namely an X86 Linux machine. The other PIM is also similar. However, the value of the parameter *hidden_layer_sizes* is 8 rather than 32.

In addition, we require two PIMs for making predictions using the ML models. Again, the two PIMs correspond to the two experiments (namely, rows 1-4 and rows 5-8 in Table II). These two PIMs for prediction are imported in the PSMs for prediction. In principle, we need one PSM for each of the target platforms. Since we have three target platforms (x86, RPI and Arduino) and two experiments, we have in total 6 PSMs for prediction. We depict part of one of the PSMs for prediction on the x86 platform in Figure 11. The other ones are also similar to this one. However, their *@compiler* annotations in their *configurations* have the values *rpi_3b+_python*, *rpi_3b+_python_quantized* and *arduino_nano_33_ble_sense_cpp* for each of the respective target platforms. The code generators not only produce the source code that has the APIs of these platforms, but also automatically converts the ML model to the right format for each of the specific platforms.

To make it more comprehensible, we show the behavioral part of the model instances of Figure 6 and Figure 9 in the graphical form in Figure 7 and 10, respectively. The implementation of the state machines is carried out through the *statechart* section of the textual model instances (see the last lines in Figure 6 and Figure 9).

For more information about the syntax of ML-Quadrat, please refer to its documentation [14] and prior work [13].

VI. Conclusion & Future Work

In this paper, we proposed a novel approach to model-driven development of smart IoT services that can be deployed on a wide variety of distributed platforms, including the highly resource-constrained, ultra low-power microcontrollers. We enabled TinyML, thus supported the deployment of compact ML models on the said microcontrollers. To this aim, we used

```
thing Predictive_Maintenance_Training_Server includes Hydraulic_Rig_Msgs {
    provided port training_service {
        sends training_done
        receives training_request
    }
    property vs1_value: Double
    property eps1_value: Double
    property se_value: Double
    property leakage: Boolean
    data_analytics predictive_maintenance {
        labels ON
        features vs1_value, eps1_value, se_value, leakage
        prediction_results leakage
        dataset "data/hydraulic_rig.csv"
        sequential TRUE
        timestamps ON
        model_algorithm nn_multilayer_perceptron my_nn_mlp
        (hidden_layer_sizes (32), activation relu, batch_size 100,
            learning_rate_init "1e-5", epochs 200
        )
        training_results "data/training_predictive_maintenance.txt"
    }
    statechart Predictive_Maintenance_Training_Server_Behavior init Preprocess {
```

Fig. 6. Part of the PIM for training the ML model.

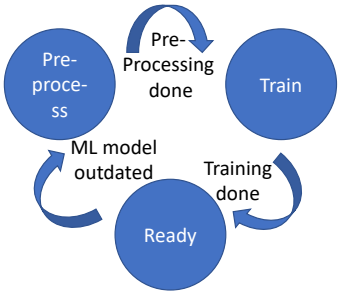

Fig. 7. The behavioral model of the PIM for training the ML model.

```
import "Hydraulic_Rig_Training_PIM.thingml"

configuration Hydraulic_Rig_Training_Cfg @compiler "python_java" {
    instance dB_Server : DB_Server
    instance vS1 : VS1
    instance ePS1 : EPS1
    instance sE : SE
    instance predictive_Maintenance_Training_Server :
    Predictive_Maintenance_Training_Server

    connector dB_Server.sensor_service => vS1.sensor_service
    connector dB_Server.sensor_service => ePS1.sensor_service
    connector dB_Server.sensor_service => sE.sensor_service
    connector dB_Server.predictive_maintenance_service =>
    predictive_Maintenance_Training_Server.training_service
}
```

Fig. 8. Part of the PSM for training the ML model on an x86 Linux platform. The PIM of Figure 6 is imported here.

TABLE II
Experimental Results

	Experiment & Platform	Prediction time (s)	Accuracy	Precision	Recall	ML Model size
1	1, x86 Linux	0.119	97%	97%	97%	2.4 MB
2	1, RPI	1.43	97%	97%	97%	785 KB
3	1, RPI, Q.	0.572	97%	97%	97%	198 KB
4	1, Ard., Q.	n/a	n/a	n/a	n/a	1.2 MB
5	2, x86 Linux	0.086	80%	86%	80%	613 KB
6	2, RPI	0.683	80%	86%	80%	197 KB
7	2, RPI, Q.	0.482	80%	85%	80%	51 KB
8	2, Ard., Q.	218	80%	85%	80%	316 KB

```
thing Predictive_Maintenance_Prediction_Server includes Hydraulic_Rig_Msgs {
    provided port prediction_service {
        sends prediction_done
        receives prediction_request
    }
    property vs1_value: Double
    property eps1_value: Double
    property se_value: Double
    property leakage: Boolean

    data_analytics predictive_maintenance {
        labels ON
        features vs1_value, eps1_value, se_value, leakage
        prediction_results leakage
        dataset "data/hydraulic_rig.csv"
        sequential TRUE
        timestamps ON
        model_algorithm nn_multilayer_perceptron my_nn_mlp
        (hidden_layer_sizes (32), activation relu, batch_size 100,
            learning_rate_init "1e-5", epochs 200
        )
        training_results "data/training_predictive_maintenance.txt"
    }
    statechart Predictive_Maintenance_Prediction_Server_Behavior init Ready {
```

Fig. 9. Part of the PIM for predictions using the ML model.

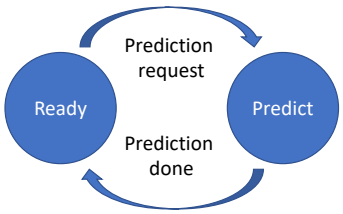

Fig. 10. The behavioral model of the PIM for predictions using the ML model.

```
import "Hydraulic_Rig_Prediction_PIM.thingml"

configuration Hydraulic_Rig_Prediction_Cfg @compiler "python_java" {
    instance dB_Server : DB_Server
    instance vS1 : VS1
    instance ePS1 : EPS1
    instance sE : SE
    instance predictive_Maintenance_Prediction_Server :
    Predictive_Maintenance_Prediction_Server

    connector dB_Server.sensor_service => vS1.sensor_service
    connector dB_Server.sensor_service => ePS1.sensor_service
    connector dB_Server.sensor_service => sE.sensor_service
    connector dB_Server.predictive_maintenance_service =>
    predictive_Maintenance_Prediction_Server.prediction_service
}
```

Fig. 11. Part of the PSM for predictions using the ML model on an x86 Linux platform. The PIM of Figure 9 is imported here.

the APIs of the TensorFlow Lite [32] and the TensorFlow Lite for microcontrollers [33] libraries.

First, we validated RQ1 by showing the feasibility of full source code generation in an automated manner. In addition to generating the source code for the different target platforms, we also automatically converted the ML models to the right formats for each of them. Second, we validated RQ2 concerning the different levels of abstraction on the modeling layer: PIMs vs. PSMs. We support importing a PIM that abstrcats from the underlying platform-specific details in multiple PSMs, such that the PSMs can add the APIs of the target platforms and enable full code generation out of them.

The validation was performed through a case study. While this is a common empirical research method, we acknowledge that this single case study and use case scenario might not be representative enough for the entire IoT domain. Therefore, future research work is required to assess and validate the proposed approach for multiple other scenarios and cases. In addition, we used the open data of a hydraulic test rig for the validation. Concerning the reported evaluation results, we must note that any dataset typically contains a certain degree of noise and often has multiple missing values. In this work, we did not handle any imputation of missing values since the provided dataset did not contain any. We assume that the data have already been cleaned before being released publicly.

Furthermore, future work can extend the proposed approach to enable federated learning by embedded platforms and TinyML devices such that an ML model can be built collaboratively without sharing raw data between the devices. In contrast, in the present work, we trained the ML model on one platform (x86 Linux) and deployed it on three different platforms, including a TinyML platform.

APPENDIX

In the following, we briefly explain some of the keywords of the textual concrete syntax of the DSML of ML-Quadrat [14] that is adopted in this work:

Labels: This is a binary parameter that can have the values ON, OFF or SEMI for supervised, unsupervised and semi-supervised ML, respectively.

Features: This is a list of the properties (i.e., local variables) of the *thing* that must be considered as ML features (attributes). The local variables might include the messages or parameters of the messages.

Prediction_results: This parameter determines the property of the *thing* in which the prediction result of the ML model must be stored.

Sequential: A Boolean parameter that indicates whether the input data are sequential, e.g., time series, where the order of data instances matters. In this case, shuffling and cross-validation should be avoided.

Timestamps: A binary parameter that states if the data instances have timestamps.

Model_algorithm: Here, one can specify the particular ML model architecture that must be deployed, e.g., the Multi-Layer Perceptron (MLP) Neural Networks (NN). Additionally, the hyperparameters, e.g., the choice of the error/loss function, the learning/optimization algorithm, the learning rate, etc. might be given in the parenthesis. Each family of ML models may have a different set of possible hyperparameters.

Training_results: This is the path of the text file in which the log of training must be stored.

ACKNOWLEDGMENTS

This work is partially funded by the German Federal Ministry for Education & Research (BMBF) through the Software Campus initiative (project ML-Quadrat).

References

[1] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, 1st ed. Wiley, 2008.

[2] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A language and code generation framework for heterogeneous targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16, 2016.

[3] E. Geisberger and M. Broy, Eds., *Living in a networked world. Integrated research agenda Cyber-Physical Systems (agendaCPS)*, ser. acatech STUDY. Munich, Germany: Herbert Utz Verlag, 2014.

[4] B. Schaetz, "The role of models in engineering of cyber-physical systems – challenges and possibilities," in *CPS20: CPS 20 years from now - visions and challenges*, ser. CPS Week, 2014.

[5] S. Documentation, "Simulation and model-based design," 2020. [Online]. Available: https://www.mathworks.com/products/simulink.html

[6] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz, "Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems," in *Joint Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-physical and Embedded Systems and 1st International Workshop on UML Consistency Rules (ACES-MB 2015 & WUCOR 2015) co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, ser. CEUR Workshop Proceedings, I. Dragomir, S. Graf, G. Karsai, F. Noyrit, I. Ober, D. Torre, Y. Labiche, M. Genero, and M. Elaasar, Eds., vol. 1508. CEUR-WS.org, 2015, pp. 19–26. [Online]. Available: http://ceur-ws.org/Vol-1508/paper4.pdf

[7] B. Morin, N. Harrand, and F. Fleurey, "Model-based software engineering to tame the iot jungle," *IEEE Software*, vol. 34, no. 1, pp. 30–36, 2017.

[8] F. Fleurey, B. Morin, A. Solberg, and O. Barais, "Mde to manage communications with and between resource-constrained systems," in *Model Driven Engineering Languages and Systems*, J. Whittle, T. Clark, and T. Kühne, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 349–363.

[9] "ThingML," https://github.com/TelluIoT/ThingML, 2015, accessed: 2020-04-29.

[10] B. Morin, F. Fleurey, K.-E. Husa, and O. Barais, "A generative middleware for heterogeneous and distributed services," in *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, 2016, pp. 107–116.

[11] "Heterogeneous and Distributed Services for the Future Computing Continuum," https://cordis.europa.eu/project/id/611337, 2015, accessed: 2021-09-01.

[12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, ser. CBSE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 135–144. [Online]. Available: https://doi.org/10.1145/2304736.2304759

[13] A. Moin, M. Challenger, A. Badii, and S. Günnemann, "A model-driven approach to machine learning and software modeling for the IoT," *Software and Systems Modeling (SoSyM)*, 2022. [Online]. Available: https://doi.org/10.1007/s10270-021-00967-x

[14] "ML2," https://github.com/arminmoin/ML-Quadrat, 2020, accessed: 2020-09-12.

[15] J. Park, Y. Boo, I. Choi, S. Shin, and W. Sung, "Fully neural network based speech recognition on mobile and embedded devices," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 10642–10653.

[16] P. Warden and D. Situnayake, *TinyML*. USA: O'Reilly Media, Inc., 2019.

[17] Q. Li, Z. Wen, Z. Wu, S. Hu, N. Wang, Y. Li, X. Liu, and B. He, "A survey on federated learning systems: Vision, hype and reality for data privacy and protection," 2021.

[18] "Model Driven Architecture (MDA), MDA Guide rev. 2.0," Object Management Group, Boston, MA, USA, Standard OMG Document ormsc/14-06-01, 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[19] F. Truyen, "The Fast Guide to Model Driven Architecture," Cephas Consulting Corp, Tech. Rep., 01 2006, https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf.

[20] "Arduino Nano 33 BLE Sense," https://store.arduino.cc/products/arduino-nano-33-ble-sense, accessed: 2021-12-07.

[21] "Raspberry Pi 3 B+," https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/, accessed: 2022-01-13.

[22] "The Eclipse Modeling Framework (EMF)," https://www.eclipse.org/modeling/emf/, accessed: 2021-12-15.

[23] "Xtext," https://www.eclipse.org/Xtext/, accessed: 2021-12-15.

[24] "Next-Gen Live Analytics using Temporal Graph," https://github.com/datathings/greycat, 2018, accessed: 2021-09-02.

[25] T. Hartmann, A. Moawad, F. Fouquet, and Y. Le Traon, "The next evolution of mde: A seamless integration of machine learning into domain modeling," in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017, pp. 180–180.

[26] T. Hartmann, F. Fouquet, A. Moawad, R. Rouvoy, and Y. L. Traon, "Greycat: Efficient what-if analytics for data in motion at scale," 2018.

[27] T. Hartmann, A. Moawad, F. Fouquet, and Y. Le Traon, "The next evolution of mde: a seamless integration of machine learning into domain modeling," *Software and System Modeling (SoSyM)*, vol. 18, p. 1285–1304, May 2019.

[28] A. Moin, S. Rössler, M. Sayih, and S. Günnemann, "From things' modeling language (thingml) to things' machine learning (thingml2)," in *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, E. Guerra and L. Iovino, Eds. ACM, 2020, pp. 19:1–19:2.

[29] A. Moin, S. Rössler, and S. Günnemann, "Thingml+: Augmenting model-driven software engineering for the internet of things with machine learning," in *Proceedings of MODELS 2018 Workshops, co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, ser. CEUR Workshop Proceedings, R. Hebig and T. Berger, Eds., vol. 2245. CEUR-WS.org, 2018, pp. 521–523. [Online]. Available: http://ceur-ws.org/Vol-2245/mde4iot_paper_5.pdf

[30] F. Francois, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "Kevoree modeling framework (kmf): Efficient modeling techniques for runtime use," 2014.

[31] "The Kevoree Modeling Framework (KMF)," https://github.com/kevoree-modeling/framework, accessed: 2021-12-15.

[32] "TensorFlow Lite," https://www.tensorflow.org/lite/guide, accessed: 2021-12-10.

[33] "TensorFlow Lite for Microcontrollers," https://www.tensorflow.org/lite/microcontrollers, accessed: 2021-12-10.

[34] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[35] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[36] N. Helwig, E. Pignanelli, and A. Schütze, "Condition monitoring of a complex hydraulic system using multivariate statistics," in *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings*, 2015, pp. 210–215.

[37] "Hydrotechnik," https://www.hydrotechnik.co.uk/hydraulic-and-hydrostatic-test-rigs, accessed: 2022-01-14.

[38] "Condition monitoring of hydraulic systems," https://www.kaggle.com/mayank1897/condition-monitoring-of-hydraulic-systems?select=description.txt, accessed: 2022-01-14.

[39] "FlatBuffers," https://github.com/google/flatbuffers, accessed: 2022-01-25.