A model-driven approach to machine learning and software modeling for the IoT

Software and Systems Modeling
https://doi.org/10.1007/s10270-021-00967-x

THEME SECTION PAPER

A model-driven approach to machine learning and software modeling for the IoT

Generating full source code for smart Internet of Things (IoT) services and cyber-physical systems (CPS)

Armin Moin[1] · Moharram Challenger[2] · Atta Badii[3] · Stephan Günnemann[4]

Received: 19 March 2021 / Revised: 1 December 2021 / Accepted: 7 December 2021

Abstract
Models are used in both Software Engineering (SE) and Artificial Intelligence (AI). SE models may specify the architecture at different levels of abstraction and for addressing different concerns at various stages of the software development life-cycle, from early conceptualization and design, to verification, implementation, testing and evolution. However, AI models may provide smart capabilities, such as prediction and decision-making support. For instance, in Machine Learning (ML), which is currently the most popular sub-discipline of AI, mathematical models may learn useful patterns in the observed data and can become capable of making predictions. The goal of this work is to create synergy by bringing models in the said communities together and proposing a holistic approach to model-driven software development for intelligent systems that require ML. We illustrate how software models can become capable of creating and dealing with ML models in a seamless manner. The main focus is on the domain of the Internet of Things (IoT), where both ML and model-driven SE play a key role. In the context of the need to take a Cyber-Physical System-of-Systems perspective of the targeted architecture, an integrated design environment for both SE and ML sub-systems would best support the optimization and overall efficiency of the implementation of the resulting system. In particular, we implement the proposed approach, called ML-Quadrat, based on ThingML, and validate it using a case study from the IoT domain, as well as through an empirical user evaluation. It transpires that the proposed approach is not only feasible, but may also contribute to the performance leap of software development for smart Cyber-Physical Systems (CPS) which are connected to the IoT, as well as an enhanced user experience of the practitioners who use the proposed modeling solution.

Keywords Model-driven software engineering · Domain-specific modeling · Analytics modeling · Machine learning · Internet of things · Cyber-physical systems

1 Introduction

As software and information/data-intensive systems, such as Cyber-Physical Systems (CPS), which are highly complex systems of systems [16], become smarter through incorporating Artificial Intelligence (AI), and more pervasive via the

Communicated by L. Burgueño, J. Cabot, M. Wimmer & S. Zschaler.

✉ Armin Moin
moin@in.tum.de

Moharram Challenger
moharram.challenger@uantwerpen.be

Atta Badii
atta.badii@reading.ac.uk

Stephan Günnemann
guennemann@in.tum.de

1 Department of Informatics, Technical University of Munich, Munich, Germany

2 Department of Computer Science, University of Antwerp & Flanders Make, Flanders, Belgium

3 Department of Computer Science, University of Reading, Reading, UK

4 Department of Informatics and Munich Data Science Institute, Technical University of Munich, Munich, Germany

Internet of Things (IoT) with billions of networked devices [2], we observe an increasing need for integration and liaison between the Software and Systems Engineering (SSE) community on the one side and the AI, including the Data Analytics and Machine Learning (DAML) community on the other side. To this aim, two research directions motivated by the following broad research questions are evolving simultaneously: (i) How to enhance SSE through AI (e.g., DAML)? For instance, the field of Mining Software Repositories (MSR), which deals with applying DAML methods and techniques to large amounts of data that are stored in various formats in the software source code and bug repositories, in order to make software development more efficient, serves as an example for this direction. (ii) How can AI, e.g., DAML benefit from SSE approaches and paradigms, such as Model-Driven Software Engineering (MDSE), also known as Model-Based Software Engineering (MBSE)? This work lies at the intersection of the said research directions since it aims to bring both communities together and contribute to each one.

Due to the abstraction and the automation that they can provide, software models in the context of the MDSE paradigm, especially the Domain-Specific Modeling (DSM) methodology [26] with full code generation play an important role in the highly complex and very large software systems of today. In particular, in the IoT domain, where distributed systems with heterogeneous hardware and software platforms, programming languages and communication protocols are the norm, one can better perceive the additional value of models and domain-specific MDSE [20,46]. Prior work in the literature, such as ThingML [14,20,34,49], HEADS [24,33] and μ-Kevoree [15] (see Sect. 3) concentrated on domain-specific MDSE for the IoT/CPS domain. However, the main shortcoming of these models is that they cannot support the ever-increasing DAML requirements of software systems, in particular, in the IoT/CPS domain where massive datasets and data streams are being generated by the sensors and other devices. We argue that the Domain-Specific Modeling Languages (DSML) for the IoT/CPS have to include DAML concepts and offer access to the APIs of libraries and frameworks for DAML on the modeling layer. Otherwise, the DAML functionalities of smart, data-driven IoT Services and CPS applications need to be implemented separately either in a manual way or using other 'silo' DSMLs. However, this would be in contrast to the DSM goal concerning being able to generate every artifact out of the abstract MDSE models, through model-to-code/model-to-text and model-to-model transformations, in an automated, integrated and seamless manner.

Furthermore, DAML models, such as Probabilistic Graphical Models (PGM) or Artificial Neural Networks (ANN) are not capable of acting as software models for the entire system, e.g., for modeling a complete smart IoT service or smart CPS application. Bishop [6] proposed Infer.NET [29], which was a DSML, based on the probabilistic programming paradigm, in order to treat PGMs as both ML models and software models in the sense of domain-specific MDSE, where the entire software solution is generated out of the model. However, the main drawback of such an approach is that PGMs and other families of ML models are not expressive enough to be capable of modeling the entire system for IoT and CPS use case scenarios.

In contrast, we enhance software models, in order to make them capable of creating, training, deploying and re-training ML models as necessary for IoT use cases. However, the proposed approach is not tied to any specific vertical problem (application) domain. This means, the proposed solution can be deployed in diverse vertical domains, such as smart healthcare and smart energy systems. This is in accordance with the nature of CPS, which are cross-domain by definition, and the IoT, that is an interconnection of all such cross-domain systems of systems [16,46].

The original idea was proposed previously in our position paper [31], as well as our poster/extended abstract [32]. In this work, we elaborate on the proposed approach more thoroughly, illustrate our implementation of the prototype that serves as the proof-of-concept, as well as the validation of the proposed approach. Hence, the contribution of this paper is twofold: (i) We validate the research hypothesis that software developers using the MDSE paradigm, particularly the DSM methodology, may have their software models enhanced with the capability to automatically produce and train ML models, and deal with them. Simultaneously, we maintain the feasibility of full source code generation in an automated way. The said ML models may affect the behavioral models of software systems. This is validated using a case study. (ii) In addition to the feasibility of the proposed approach, we validate the hypothesis that it contributes to the performance leap of software development in the IoT domain and leads to a higher level of satisfaction regarding the user experience of the practitioners (i.e., software developers, data scientists, etc.) who use the proposed approach. This is validated through an empirical evaluation by a number of external experts.

We provide our open-source prototype, called ML-Quadrat, with sufficient documentation and samples to facilitate using this as a platform to let both software developers and ML practitioners support new IoT platforms and ML libraries. This shall lead to open innovations and generate synergies in both the SSE and AI communities. Using the proposed approach, the SSE community is empowered with the state-of-the-art ML methods and techniques out-of-the-box, while the ML community can obtain access to the scalable, robust and efficient Software Engineering (SE) solutions, based on best practice. The integration of the said models from SE and ML is conducted in a seamless manner that does not require any knowledge and skills in the particular APIs of the underlying platforms and libraries. For instance, to generate Python code for ML, based on the APIs of different libraries and frameworks, one does not need to be familiar with their specific APIs. Our DSML abstracts from those platform-specific APIs, hence offering a higher layer of abstraction, i.e., the modeling layer. Different model-to-code transformations, also known as code generators can generate the entire source code for various DAML libraries and frameworks, e.g., Scikit-Learn [40] and Keras [9] with the TensorFlow [1] backend in a fully automated manner.

Moreover, since our work is built based on the open-source ThingML [49] project, we also inherit their code generators (‘compilers’) for various platforms, programming languages and protocols. ThingML [49] can generate code in Java, C (Posix, Teensy, Arduino), C++, Javascript and Go. Further, they support not only the Hypertext Transfer Protocol (HTTP), but also the more suitable application layer communication protocols for resource-constrained IoT-devices, namely the Constrained Application Protocol (CoAP) for one-to-one communications and the Message Queuing Telemetry Transport (MQTT) protocol for many-to-many communications following the publish-subscribe pattern. In this work, we extend their approach, including the meta-model, as well as the code generation framework to enable generating Python code for supporting the required DAML functionalities.

The rest of this paper is structured as follows: Sect. 2 provides the required background on the IoT/CPS and the preliminaries on analytics modeling, as well as software modeling. Moreover, Sect. 3 reviews the state of the art and points out the gap in the literature that is being addressed by the present work. We propose our novel approach in Sect. 4 that is followed by presenting the open-source prototype in Sect. 5. Further, we validate the above-mentioned research hypotheses in Sect. 6. Finally, we conclude and suggest future work in Sect. 7.

2 Background

2.1 The Internet of Things (IoT) and cyber-physical systems (CPS)

The original *World Wide Web (WWW)* was developed in 1989 to enable automated information-sharing between scientists in universities and institutes around the globe [7]. The term *Web 2.0* was introduced in 1999 [12] as user-generated content on the web gained more attention. In 2001, *Web 3.0* or the *Semantic Web* was introduced [3]. This was an extension of the web to support machine-readable multi-media content development on the web, i.e., *semantic* data that could be processed and *understood* by computers, such that they can conduct reasoning supported by the semantic markup. To this aim, the World Wide Web Consortium (W3C) promoted a set of standards, such as the Resource Description Framework (RDF) that could enable data from heterogeneous sources to be shared and reused across applications, websites and mobile apps. These semantic technologies let concepts, objects and their relationships be formally represented through meta-data, e.g., via ontologies. Today, the fourth generation of the web, i.e., *Web 4.0*, which is better known as the *Internet of Things (IoT)*, is being gradually formed. The IoT is an expansion of the Internet into new domains, devices and objects (i.e., *things*), such as Radio-Frequency Identification (RFID) tags, sensors, actuators, mobile phones, etc. which through unique addressing schemes are able to interact and perhaps also cooperate with each other to reach common goals [2].

Another related but slightly different notion is Cyber-Physical Systems (CPS). Similar to the IoT systems, CPS, which are highly complex systems of systems that possess both physical and virtual (cyber) components [16], consist of heterogeneous and distributed platforms, such as various embedded micro-controllers. As more CPS are being connected to the Internet (IoT), we no longer need to distinguish between the two notions of CPS and IoT. Nevertheless, there is no consensus on the exact definition of CPS and its borders and/or possible overlaps with the IoT. The US National Institute of Standards and Technology (NIST) Special Publication on CPS and the IoT [17] also highlighted this fact and pointed out that CPS and the IoT have ‘distinct origins but overlapping definitions, with both referring to trends in integrating digital capabilities, including network connectivity and computational capability, with physical devices and systems’.

CPS have by nature special capabilities, known as the so-called *cross-**, *live-** and *self-** capabilities. The *cross-** capabilities, include cross-domain, cross-technology, cross-organization and cross-functional. Moreover, the *live-** capabilities comprise live-re-configuration, live-re-deployment, live-update, live-enhancement and live-extension. Further, the *self-** capabilities are self- documenting, self-monitoring /diagnosis, self-optimizing, self- healing and self-adapting /training [46].

Since CPS involve both the physical and the virtual (cyber/digital) worlds, modeling them is quite challenging. For instance, in the physical world, the dynamics of the system is captured by a set of variables that change their values *continuously* over time. The dependencies between these variables are captured by *continuous functions* that are expressed by differential calculus and integration theory, where time is represented by real numbers. By contrast, digital systems can be modeled as discrete event systems with a number of states. They can be modeled, e.g., via state machines or Petri-Nets. Thus, in digital systems, time is discrete. Furthermore, in such systems, the notion of causality, i.e., the logical dependencies between the events, might be more sophisticated than the notion of time [16,50]. Finally, Papatheocharous et al. [38] proposed a closely related and similar concept to CPS in their position paper, called Federated Embedded Systems (FES).

In this work, we focus on modeling IoT services that require smart capabilities through Machine Learning (ML). As a motivating example, let us consider a condition-based monitoring system of a hydraulics system in an industrial facility. The goal is to conduct predictive maintenance via ML models that are trained on the data that are acquired from a number of various sensors (e.g., multiple pressure and temperature sensors), thus enabling the prediction of possible future faults of the system by the ML models. Moreover, there exist a number of *virtual sensors*, whose values are not directly measured by any physical sensor device, but they are calculated based on other sensor measurements. One example is the cooling efficiency. There is no sensor to measure this quantity explicitly, but it is calculated according to the oil temperature at the cooler, one of the temperature sensors, as well as the ambient temperature. Helwig et al. [25] elaborated on this condition-based monitoring system that is deployed in Germany.

In line with the above-mentioned vision of the IoT, we assume that this system will be connected to the IoT in the future. In other words, each of the sensors and actuators involved will be directly connected to the Internet (IoT). One advantage of this will be the possibility of letting the condition-based monitoring systems deployed at multiple facilities or sites of one customer or a group of customers cooperate to the benefit of all of them. This might involve sharing their data to enhance the prediction performance of the ML models that are created and trained for different hydraulics systems. However, in the case that privacy concerns and regulations discourage or prohibit sharing raw data, they may use federated ML techniques, through which a number of systems deployed at various sites may cooperate in order to create a more capable joint ML model without sharing any raw data. The proposed approach in this work enables the modeling infrastructure for edge analytics and federated ML since it allows augmenting any arbitrary *thing* with one or more data analytics component. Figure 1 illustrates the said hydraulics system that includes a primary working circuit and a secondary cooling and filtration circuit, as well as the predictive maintenance system for condition-based monitoring of the hydraulics system. The entire system is a CPS that is connected to the Internet (IoT). This use case scenario is an example of typical smart IoT services that can be modeled and their implementations can be automatically generated using the proposed approach.

2.2 Analytics modeling

Analytics modeling is a term that stands in contrast to *analytics operations*. In fact, the core focus of the data analytics, also known as the Knowledge Discovery and Data Mining (KDD) community is on analytics modeling, which involves developing new algorithms, methods and techniques to manage and analyze data, e.g., for business intelligence, decision making support, optimization, predictive maintenance and so forth. One of the fields that has recently very much helped them in achieving their goal is ML (especially its sub-discipline *deep learning*). Data scientists and ML engineers often practice analytics modeling. They usually offer the software that produces and trains DAML models, and are called (DAML) *model producers*. However, in order to deploy and use DAML models in real-world systems, we also need data engineers, who together with software engineers, database engineers/designers and system engineers take other aspects, such as the performance and scalability of the entire system into account. The tasks of data engineers that mainly involve large-scale data analytics and processing (often referred to as *big data analytics*) are grouped under the umbrella term analytics operations. Data engineers often provide the software that consume or use DAML models, thus called (DAML) *model consumers*, also known as *scoring engines* [42]. Note that in the stream processing (i.e., online learning) scenarios, where training the DAML model shall be an ongoing process that needs to be performed in a live manner, the boundaries between the mentioned groups of tasks may sometimes become blurred.

In the DAML community, the notion of *models* is generally understood as the abstractions about the observed data that can help in understanding, analyzing and managing the data to generate value, e.g., to generate plausible instances of such data in order to make predictions. Leskovec et al. [28] referred to several common approaches to models in this community. For instance, one may define such a model as an underlying probability distribution, from which the observed data are presumably drawn. This is called the statistical approach. Alternatively, one may consider a model for a dataset to be a summarization or an approximation of its data instances. Further, some models represent a dataset by its most extreme examples. Those are called feature-based models. Finally, ML models that are currently widely used in analytics modeling—and are the main focus of this work—may come from diverse families, e.g., linear models, decision trees, ensemble models, such as random forests, kernel-based models, e.g., Support Vector Machine (SVM), Artificial Neural Networks (ANN) and Probabilistic Graphical Models (PGM) [5]. Deep ANNs with several hidden layers are currently widely used in the industry. Also, Bayesian Deep Learning [51] is a promising approach for many industrial IoT/CPS use cases.

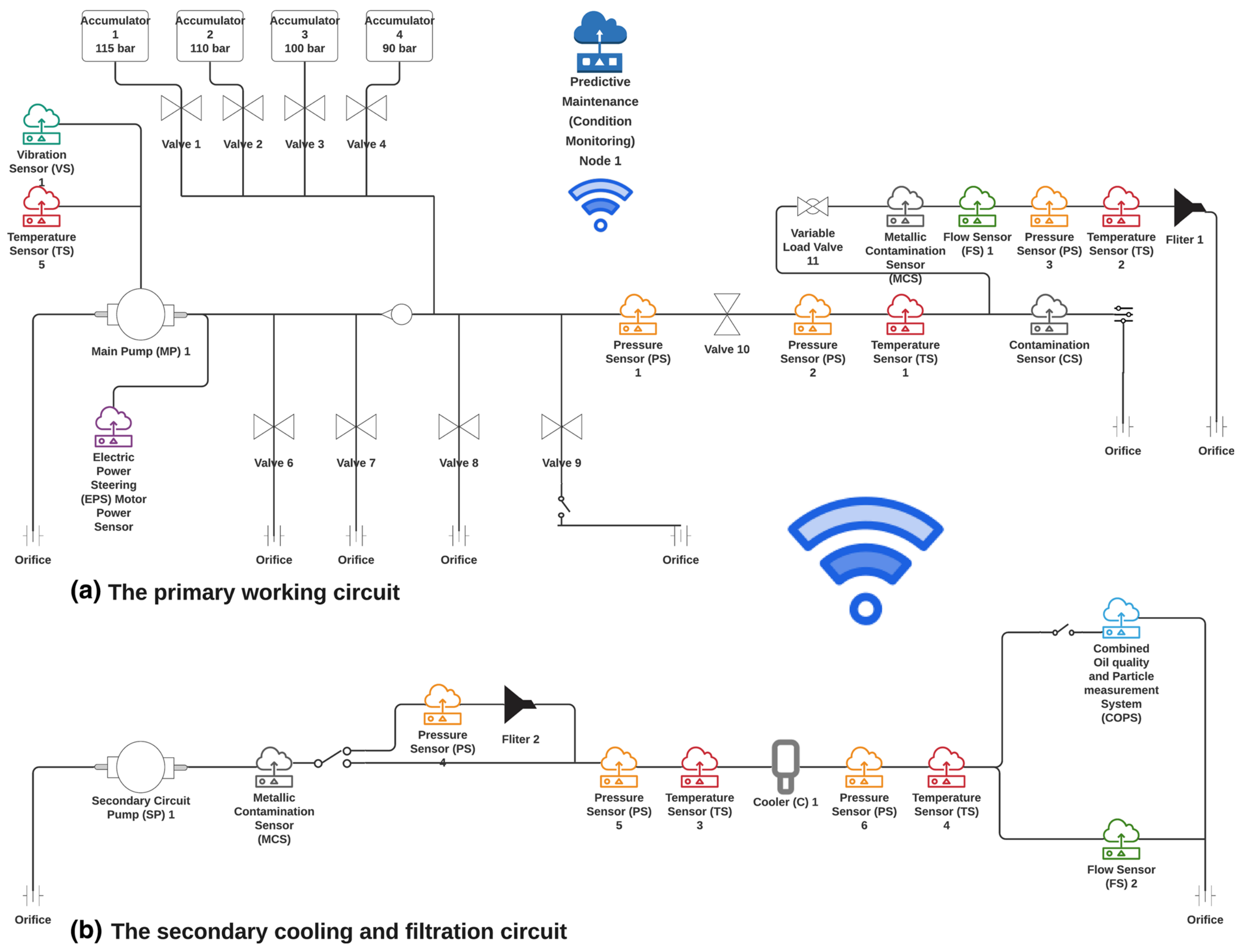

Accumulator 1 115 bar
Accumulator 2 110 bar
Accumulator 3 100 bar
Accumulator 4 90 bar
Predictive Maintenance (Condition Monitoring) Node 1
Vibration Sensor (VS) 1
Temperature Sensor (TS) 5
Valve 1
Valve 2
Valve 3
Valve 4
Variable Load Valve 11
Metallic Contamination Sensor (MCS)
Flow Sensor (FS) 1
Pressure Sensor (PS) 3
Temperature Sensor (TS) 2
Fliter 1
Main Pump (MP) 1
Pressure Sensor (PS) 1
Valve 10
Pressure Sensor (PS) 2
Temperature Sensor (TS) 1
Contamination Sensor (CS)
Electric Power Steering (EPS) Motor Power Sensor
Valve 6
Valve 7
Valve 8
Valve 9
Orifice
(a) The primary working circuit
Combined Oil quality and Particle measurement System (COPS)
Pressure Sensor (PS) 4
Fliter 2
Secondary Circuit Pump (SP) 1
Metallic Contamination Sensor (MCS)
Pressure Sensor (PS) 5
Temperature Sensor (TS) 3
Cooler (C) 1
Pressure Sensor (PS) 6
Temperature Sensor (TS) 4
Flow Sensor (FS) 2
(b) The secondary cooling and filtration circuit

Fig. 1 Predictive maintenance of a hydraulics system [25]

Furthermore, we need to clarify the terminology on model-based ML. Until recently (and even broadly today), model-based ML was (and is) understood as ML approaches that contrary to the so-called instance-based (also known as memory-based) ML approaches, they do not require storing any instances of the observed dataset that is used for training for the future uses. This means, the so-called model-based ML approaches, such as ANNs, have the ability to completely *learn* the recognized patterns in the data and function independently of the observed data, once training is done. In contrast, instance-based approaches, e.g., SVMs require at least part of the observed dataset even after training in order to be able to work [5].

However, a nuanced notion of model-based ML, which is in line with the understanding of the SSE community from the term model-based, emerged with Infer.Net [6,29]. According to this notion, which is also deployed here, model-based ML can be used with any ML model architecture, regardless of being instance-based or not.

2.3 Software modeling

In the SSE community, models are abstractions that describe the architecture of a software/system. Here, we are interested in software systems. Therefore, we concentrate on software models. Models can be at different levels of abstraction, thus having different degrees of details. Moreover, models may focus on different aspects of software systems. As long as a model can address the concerns of a stakeholder, it is interesting and relevant. A model instance shall conform to a meta-model, which specifies the syntax (and maybe also part of the semantics) of the corresponding modeling language. A modeling language might be general purpose, such as the Unified Modeling Language (UML) standard, or domain-specific, e.g., ThingML [49]. According to the ISO/IEC/IEEE 42010:2011 standard [27] for the architecture descriptions in systems and software engineering, an architecture description is made of one or often more architecture views. Several (software architecture) model instances may belong to one architecture view, which addresses one or several concerns of a stakeholder or a group of stakeholders. Based on the said standard, each architecture view is governed by one architecture viewpoint, which frames one or several concerns of a stakeholder or a group of stakeholders.

Further, if we consider the UML diagram notations, we observe that they can be categorized into two broad groups: (i) structural diagrams, e.g., the Class diagram, the Component diagram and the Object diagram; (ii) behavioral (including interaction) diagrams, e.g., the Activity diagram, the State machine diagram and the Use case diagram. The UML Activity diagram might be used for modeling the workflows (i.e., the flow of control) or data flows (i.e., the flow of data).

However, in this work, we are interested in Domain-Specific Modeling (DSM) with automated full code generation [26], a MDSE approach that has been adopted both by the ThingML methodology [20,49] and ourselves [30–32]. Nevertheless, there exist other approaches to software modeling which either do not promise automated full code generation (e.g., they just generate a skeleton), or do not consider models as the central artifacts, i.e., they are not model-driven (model-based), but rather use models for specific tasks, such as designing, early prototyping and documentation. In this work, we are not interested in such approaches.

3 Related work

Raising the level of abstraction to hide the complexity, and providing partial or full automation—e.g., via model-to-code transformations for code generation out of software models, or via model-to-model transformations for transforming one model to another model conforming to a different meta-model—are two pillars of the MDSE paradigm, which treats software models as first-class citizens. Both of the said pillars have already been introduced to some extent in the field of DAML as well. Raising the level of abstraction has been practiced through libraries and frameworks with higher level APIs. For instance, TensorFlow [1] offers a powerful API for deep learning using various advanced methods, while Keras [9] provides yet a higher layer of abstraction, which supports both the APIs of TensorFlow and other deep learning frameworks, e.g., Theano [48]. Moreover, DAML workflow designers, such as KNIME [4] and RapidMiner [44], and visualization toolkits, such as TensorBoard [47], offer a graphical and abstract layer beyond the code. However, none of the mentioned approaches followed the systematic and holistic approach of the MDSE paradigm, where *models* include the necessary information regarding the entire application, and model-to-code transformations are often capable of generating the software implementation out of them. The workflows in KNIME [4] and RapidMiner [44] or the Computational Graphs (CG), also known as the Data-Flow Graphs (DFG) in TensorBoard [47], which is the visualization toolkit for TensorFlow [1], never address any aspect or concern beyond DAML. Last but not least, some workflow designers, e.g., KNIME [4] provide the partial code generation functionality for DAML.

Furthermore, the idea of Model-Interchange Formats, such as Predictive Model Markup Language (PMML) [43], Portable Format for Analytics (PFA) [41,42] and Open Neural Network Exchange (ONNX) [37] is relevant to the principles and common practices of MDSE. PMML is an XML-based standard of the Data Mining Group (DMG) [13], which comes in the form of an XML-schema and is already supported by more than 30 vendors world wide. Also, PFA is an emerging standard of the DMG, which offers a much higher degree of flexibility and power compared to PMML. First, unlike PMML, that only supports a limited set of DAML models, PFA provides a DSL that enables the implementation of any DAML method. Second, with PFA one may model an entire workflow or pipeline, not just a single DAML model. In addition, ONNX supports building Artificial Neural Networks (ANN) models from various libraries and frameworks, e.g., TensorFlow [1], Keras [9], PyTorch [39], Scitkit-Learn [40], MXNET [8], Caffe2 (which is now part of PyTorch [39]), XLA (which is a domain-specific compiler for linear algebra that can accelerate TensorFlow models), Core ML (that allows integrating ML models into the iOS apps) and the Microsoft Cognitive Toolkit (previously known as CNTK) in an interoperable manner.

The second pillar of MDSE, namely automation, has been also applied to the DAML field. Infer.Net [6,29] proposed the idea of using ML models, specifically PGMs, as MDSE models, thus generating the entire software implementation automatically out of them. They only supported C# for code generation. Although this approach to ML has so far been the most relevant approach to the MDSE paradigm, it has a major shortcoming for real-world IoT/CPS applications, where the expressiveness of PGMs and other ML models does not suffice to model the entire software system and generate the full source code out of the model instances.

Moreover, as set out in Sect. 1, ThingML [14,20,34,49] and HEADS [24,33] supported the MDSE paradigm, specifically the DSM methodology [26] for full code generation in the IoT/CPS domain. While they mainly focused on the design-time of software systems, other approaches, such as μ-Kevoree [15] concentrated on *Models@Runtime*, thus fading out the borders between the design-time (modeling-time) and the runtime of IoT services. The major shortcoming of all of the said approaches is the lack of DAML support at the modeling level. In other words, the users of those DSMLs may not deploy the APIs of DAML libraries and frameworks in their software models. Hence, there is no seamless integration between the software models and the DAML models. In this work, we fill in this gap in the literature. We allow the DAML functionalities to be offered both by the cloud and by

Table 1 Related work in the literature compared to the proposed approach (ML-Quadrat)

Description	Work	Full code gen.	DAML support	IoT / CPS domain	Model type
ML libraries and frameworks	TensorFlow [1], Keras [9], Scitkit-Learn [40], etc.		✓		DAML models
DAML workflow designers	KNIME [4], RapidMiner [44], etc.		✓		DAML models
Model Interchange Formats (MIF)	PMML [43], PFA [41,42], ONNX [37]		✓		DAML models
"Model-based" ML	Infer.Net [6,29]	✓	✓		ML (PGM) & SE models
MDE4 IoT	ThingML [14,20,34,49] and HEADS [24,33]	✓		✓	SE models
Models@ Runtime	μ-Kevoree [15]	✓		Limited	SE models
Models@ Runtime + ML	GreyCat [18]	✓	✓	Limited	SE & DAML models
MDE4 IoT + ML	ML-Quadrat [30]	✓	✓	✓	SE & DAML models

the edge devices. Therefore, our model-driven approach also supports edge analytics and federated learning by design.

The original idea of enhancing MDSE models for integrating ML models and software models has been proposed in our previous work, i.e., the position paper [31] and the poster/extended abstract [32]. In addition, Benoit et al. [10] proposed a conceptual reference model for MDE of data-centric systems that helped in identifying different models, mainly ML models and software/system models, as well as their roles in the software/system life-cycle. In this manuscript, we formalize our prior work [31,32], realize its proof-of-concept and validate the underlying research hypotheses (see Sect. 1).

Further, based on the Kevoree Modeling Framework (KMF) and μ-Kevoree [15], Hartmann et al. [21–23] proposed GreyCat [18], which integrated ML with software models in MDSE. Their idea and concepts were relevant to the work of Moin et al. [31,32]. However, they only supported Java and Javascript/Typescript code generation, which was not sufficient for our purpose since we aim to cover code generation for the entire IoT systems that often consist of a range of heterogeneous IoT platforms, which may not be capable of running any Java Virtual Machine (JVM) at all, due to their resource constraints. Therefore, we build our approach on ThingML [14,20,34,49].

Finally, Table 1 compares the related work in the literature with the proposed approach. As we can see, the proposed approach, ML-Quadrat has all the benefits of the state of the art in MDE for the IoT (MDE4IoT), namely ThingML [14, 20,34,49] and HEADS [24,33], but can also support DAML and integrate DAML models with the SE models.

4 Proposed approach

In this section, we propose a novel approach to MDE for both analytics modeling (with a focus on ML) and software modeling, particularly for the IoT use case domain. In the following, we first illustrate the overall architecture of the proposed approach in Sect. 4.1. Then, we formalize the proposed approach in Sects. 4.2, 4.3 and 4.4. As stated in Sects. 1 and 3, we extend the open-source ThingML project [14,20,34,49], including the abstract syntax, i.e., the meta-model (grammar), the concrete syntax (model editors) and the semantics that are mostly realized in the model-to-code transformations, also known as code generators ('compilers'). The proposed approach and its implementation (see Sect. 5) are backward compatible, thus interoperable with the ThingML [49] (and HEADS [24]) models and code generators. In particular, we augment the meta-model (grammar) of the DSML of ThingML [49] with a new component, called Data Analytics (DA), which is responsible for enabling Data Analytics and Machine Learning (DAML) at the modeling level, such that practitioners using the DSML can obtain access to the APIs of the DAML libraries and frameworks (e.g., Scikit-Learn [40] and Keras [9]) in their software models at the design-time. To this aim, we also have to extend the action types of ThingML [49] (see Sect. 4.3). Additionally, we extend the Java code generator of ThingML [49] to generate Python code as well. The Python code, which is seamlessly integrated with the Java code, is responsible for realizing the DAML functionalities, using the APIs of Scikit-Learn [40] and Keras [9] (the latter with the TensorFlow [1] backend).

4.1 Overall architecture

The UML Component diagram that illustrates the logical view of a number of key functional software components is presented in Fig. 2. Most of them were also present in the prior work, ThingML [49]. However, we adapted and extended them. According to the legend of the diagram, the unchanged, adapted/extended, and generated components are depicted in blue, red and green, respectively. Here, we skipped the rest of the code generators that are inherited from the ThingML [49] project, e.g., the C/C++ code generators.

Most importantly, we introduce the DAML concepts and functionalities into the DSML grammar in the Xtext framework as the main innovation concerning the meta-model (grammar). We discuss the new elements, such as the new action types in Sects. 4.3 and 5. Many other components besides the modeling language grammar, shown in Fig. 2, such as the Ecore meta-model, the model editors, namely, the textual model editor in the Eclipse Modeling Framework (EMF), the tree-based model editor in the EMF, and the web-based textual model editor (in-browser), as well as the parser are generated automatically out of this grammar.

The UML Class diagram in Fig. 3 presents part of the abstract syntax (i.e., grammar or meta-model) of the proposed DSML.[1] Except for the Data Analytics class, the rest has been adopted from the prior work, ThingML [49]. Therefore, we allow each of the *things* to optionally include one or more Data Analytics (DA) components that are in charge of carrying out DAML tasks, such as predictions. The focus of the DAML part is mainly on the ML methods and statistical inferences rather than simple analytics via some basic statistics or rule-based engines. Currently, we handle supervised and unsupervised ML.

Finally, Fig. 4 depicts the UML Activity diagram that shows the usual workflow for deploying the proposed approach in the software development process of smart, data-driven IoT services.

4.2 Analytics models (focused on ML models)

We define an ML model, called DM (the abbreviation of **D**ata **M**odel) used in analytics modeling as follows:

$$DM = (\upsilon, P, \Phi, H, I) \tag{1}$$

Here, υ is an argument that indicates the structure or family type of the ML model DM, e.g., Decision Tree (DT), Probabilistic Graphical Model (PGM) or Multi-Layer Perceptron (MLP) Artificial Neural Network (ANN), P is a set which contains all of the *parameters* of the model DM with their respective values, Φ indicates the sequence of ML *features* (i.e., ML *attributes* and their values) with their respective data types, H is the set of all *hyperparameters*, e.g., the *optimization or learning algorithm* ζ that shall be used to *train* the model DM, the choice of the *error/loss/cost/objective function* e, the *batch size* bs, the number of *epochs* ne, the *learning rate* lr if applicable, etc., and I is the set of additional information or meta-data about the model and/or the data. I might include the following items: (i) Whether the model is already trained, if applicable what the training stage is and when the time of the last training was; (ii) The paths or URIs/URLs of the dataset(s) used for training, validation and testing; (iii) Whether any of the data instances has a label (in that case the last item of the sequence of features Φ indicates the ML class labels and its data type;[2]) (iv) If the dataset is sequential, e.g., time series, so that the order of the data instances matter; (v) Whether the training is performed online, i.e., stream processing or offline, i.e., batch processing. In the former case, the dataset is virtually unbounded, whereas in the latter case, the dataset is bounded.

Analytics modeling involves designing the model DM, and then training it, which means using ζ and other hyperparameters in H to fine-tune the values of the parameters in P, so that DM can then make reasonable predictions Y_{pred} for the previously unobserved data instances, say X_{new}, where the amount of the error/loss, e for the prediction of DM given the unobserved inputs, i.e., $pred(DM, X_{new})$ remains below a certain threshold ε:

$$DM = (\upsilon, P, \Phi, H, I), \quad train(DM) \rightarrow E[e(pred(DM, X_{new}))] < \varepsilon \tag{2}$$

Here, E is the expected value and e is the error/loss, which might be defined according to various metrics, e.g., the *Mean Absolute Error (MAE)*, also known as the *L1-norm* for regression:

$$e = \frac{1}{n} \sum_{i=1}^{n} \mid \hat{y}_i - y_i \mid \tag{3}$$

In the equation above, n is the number of data instances, $\hat{y}_i$ is the predicted numerical label by DM for the i_{th} data instance, and y_i is the actual numerical label of this data instance.

As mentioned, the choice of the metric for e, e.g., MAE, is specified in the hyperparameters H. Moreover, hyperparameter tuning is an important part of the analytics modeling

[1] Note that almost every class, e.g., State Machine, Data Analytics, etc. is in practice associated with the Platform Annotation class. However, to prevent the figure from becoming cluttered, those associations are not shown here.

[2] We also support array labels/outputs. In the future, we plan to support Sequence-to-Sequence models as well (see Sect. 7).

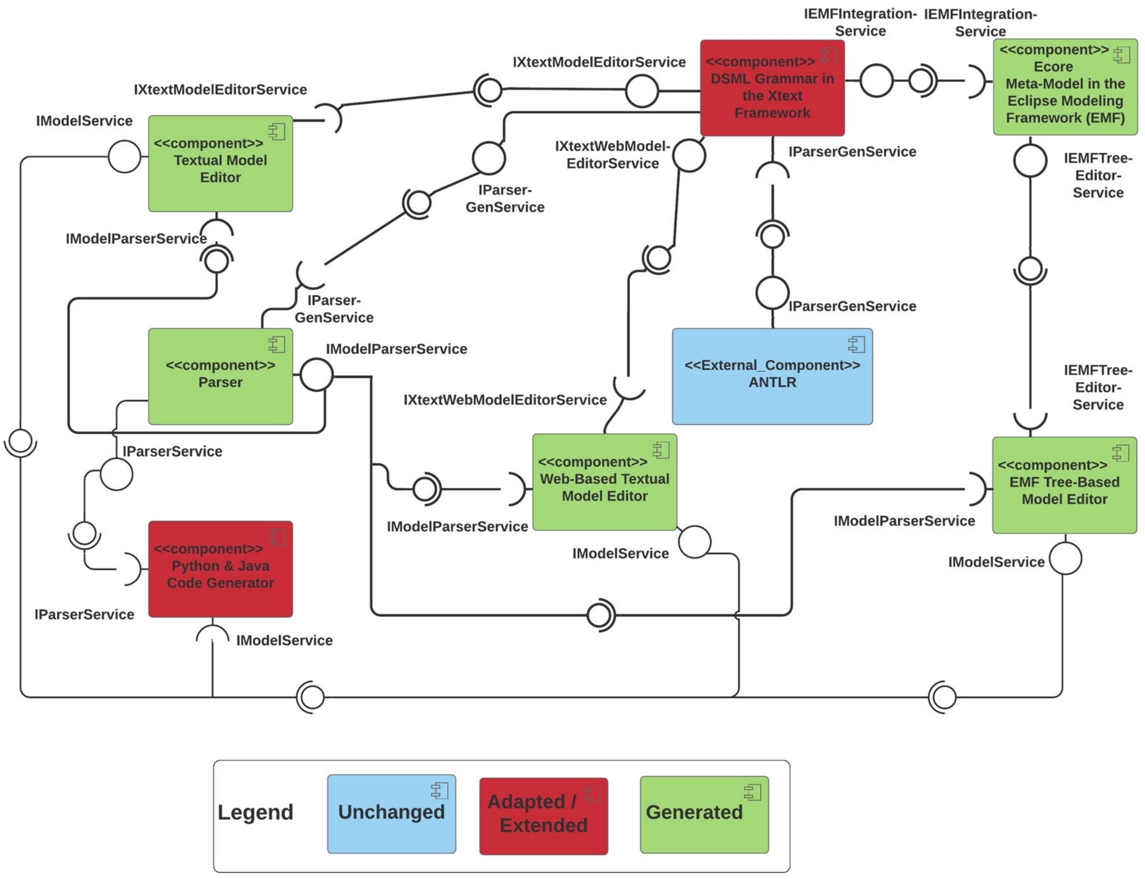
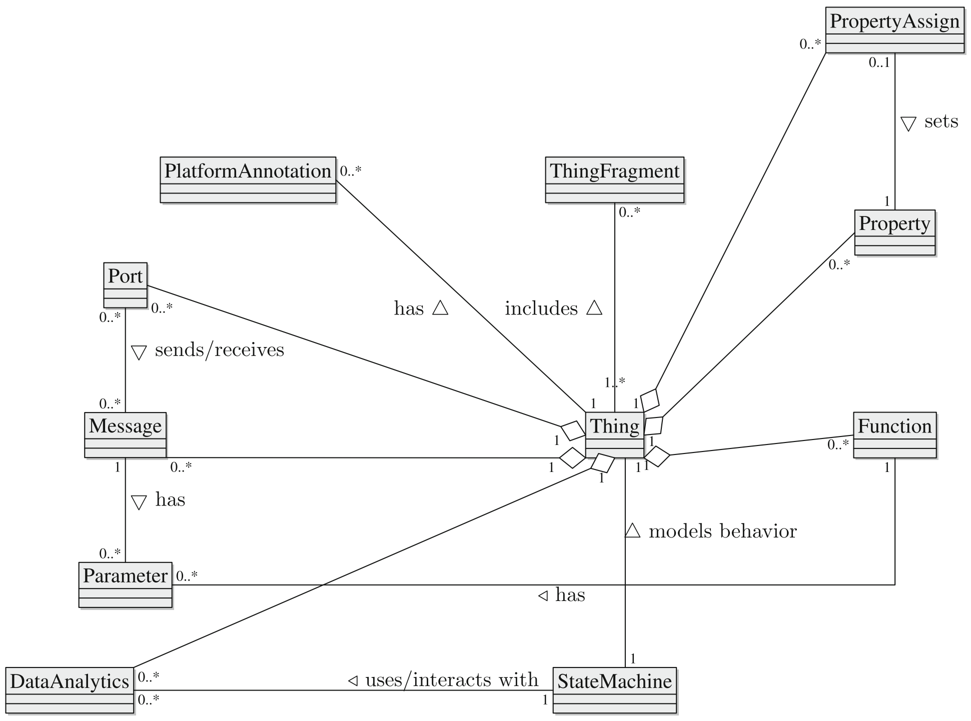


Fig. 2 The UML Component diagram illustrating the logical architecture view of the proposed approach

Fig. 3 The UML Class diagram showing part of the meta-model of the proposed DSML



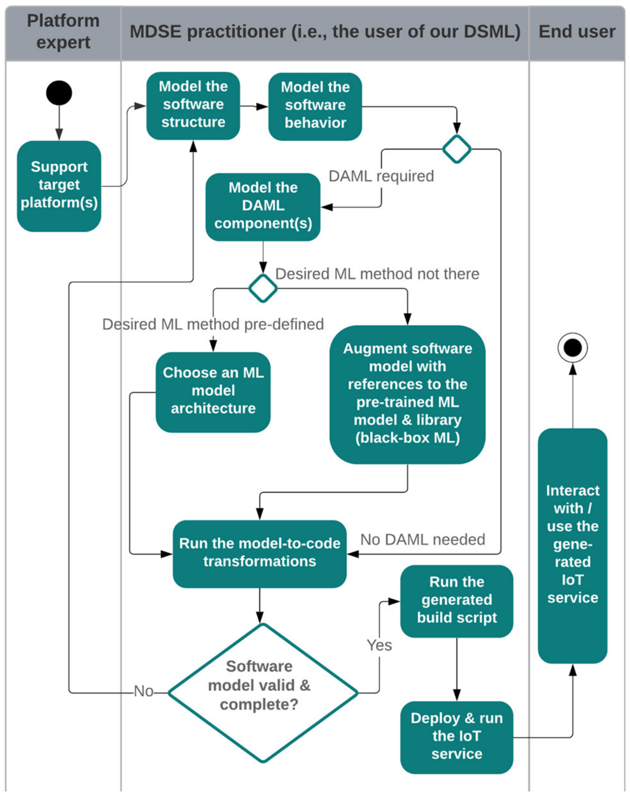

Platform expert
MDSE practitioner (i.e., the user of our DSML)
End user
Support target platform(s)
Model the software structure
Model the software behavior
DAML required
Model the DAML component(s)
Desired ML method not there
Desired ML method pre-defined
Choose an ML model architecture
Augment software model with references to the pre-trained ML model & library (black-box ML)
Run the model-to-code transformations
No DAML needed
Run the generated build script
Yes
Software model valid & complete?
No
Deploy & run the IoT service
Interact with / use the generated IoT service

Fig. 4 The UML Activity diagram illustrating the usual workflow of using the proposed approach

practices. Currently, this has to be done manually. In the future, we plan to support more Automated ML (AutoML) functionalities to offer automated hyperparameter tuning too (see Sect. 7).

If the data instances are labeled, the task is a *supervised* ML task, thus the prediction implies finding the correct class label for a new, previously unobserved data instance. However, if the data instances do not possess class labels, it is called an *unsupervised* ML task. For instance, in the case of *clustering*, which is an example for unsupervised learning, prediction refers to finding the right cluster for each new data instance. In many applications, only some instances may already have class labels and some or many of them may not have one. This latter case is called *semi-supervised* learning. Further, a supervised ML task with numerical class labels is called *regression*, whereas a supervised ML task with categorical class labels is known as *classification*.

4.3 Software models (in domain-specific MDSE for the IoT)

We define a software model, or more precisely a software architecture model instance, called SM as shown in Equation 4, where Ψ is the set of structural elements, and B is the set of behavioral elements.

$$SM = (\Psi, B) \tag{4}$$

However, since we are interested in domain-specific MDSE with automated full code generation, we augment the said software model formulation with a set of annotations, A and a set of configurations, C, thus as defined in Equation 5.

$$SM = (A, \Psi, B, C) \tag{5}$$

Annotations The Annotations (A) often help attach additional semantics to model instances. For example, one may specify which of the available library (API) choices for a certain task, such as ML methods, or the communication protocols shall be used for code generation. This means, if, for example, both Scikit-Learn and Keras offer a certain ML model/algorithm, which is desired, e.g., the MLP-ANN, one may choose through an annotation whether the APIs of Scikit-Learn or the APIs of Keras must be generated by the model-to-code transformation that generates Python code.

Structural elements The structural elements (Ψ) specify the *static* aspect of the software system. In the IoT/CPS context (see the use cases in Sect. 6.1), Ψ consists of the *things* T (in the sense of IoT cloud and edge devices in a distributed system), and for each *thing* $\tau_i \in T$, the ports P_i for communication with other *things* $\tau_j, j \neq i$, the messages M_{p_i} associated to each port for message-passing, and the properties or local variables Γ_i. Each message $m_{p_{i_j}} \in M_{p_i}$ must have a direction (inbound/outbound) and may include one or more parameter(s) $par(m_{p_{i_j}}) \in Par(m_{p_{i_j}})$. Both the properties/variables $\gamma_{i_j} \in \Gamma_i$ and the message parameters $par(m_{p_{i_j}}) \in Par(m_{p_{i_j}})$ are *typed*, e.g., integer, float/double, String, etc. How each of the mentioned types in the model instance shall be translated or mapped to the specific types of the target platforms for code generation, e.g., whether the type integer shall be mapped to short, int or long in Java, must be set through the annotations $a_i \in A$.

Behavioral elements The behavioral elements (B) specify the *dynamic* aspect of the software system. We consider a Finite-State Machine (FSM) (also known as a finite-state automaton) model, called $FSM_i \equiv B_i$ for the behavior of each of the *things* $\tau_i \in T$. We define the FSM model as follows:

$$FSM = (\Sigma, S, s_0, \delta, F, \Pi) \tag{6}$$

Here, Σ is a set of inputs (explained below) which must be finite and non-empty by definition, S is a set of states for the thing $\tau_i \in T$ which is also finite and non-empty, $s_0 \in S$ is an initial state that must be specified, $\delta : S \times \Sigma \rightarrow S$ is the state-transition function, $F \subseteq S$ is a (possibly empty) set of final states, and Π is a set of actions (illustrated below). In this work, we assume the finite-state automaton to be deterministic, i.e., given an input and a particular state, there will be only one output state for the transition function δ, not a set of states.

Moreover, since we adopt the event-driven programming paradigm, which is a natural fit for reactive and interactive IoT systems, the inputs $\sigma_i \in \Sigma_i$ in $FSM_i \equiv B_i$ (i.e., the behavioral model of $\tau_i \in T$) are basically events, e.g., the incoming messages sent from other things $\tau_j \in T, j \neq i$ to τ_i. However, the actions $\pi_i \in \Pi$ may be diverse actions, such as printing a text in the standard output, storing a message $m_{p_{i_j}}$ or one of the parameters of a message $par(m_{p_{i_j}})$ in a local variable (property) γ_{i_j} of the thing, or sending a message from τ_i to another thing $\tau_k \in T, k \neq i$. The new action types that we added to the existing DSML of ThingML [49] are the following ones for DAML: (i) *DA_Preprocess:* This action results in pre-processing the data and making them ready for training the ML model. (ii) *DA_Train:* This action leads to performing ML model training. (iii) *DA_Predict:* This action enables asking the ML model for prediction. (iv) *DA_Save:* This action supports appending the prediction of the ML model to the dataset that was used for training the ML model. Please note that the trained ML models that are resulted from the *DA_Train* action will be serialized and stored in any case regardless of the *DA_Save* action.

Configurations The configurations (C) include a set of instantiations of the things, which is analogous to object instantiation from the classes in the Object-Oriented Programming (OOP) paradigm. Also, it is at this place of the model instance where the desired connections between the ports of the instantiated things are set out. Last but not least, configurations may optionally also include annotations, e.g., specifying which model-to-code transformations shall be used for code generation, and/or which communication protocols shall be employed (e.g., MQTT, HTTP, CoAP). Hence, we define a configuration C_i for $\tau_i \in T$ as follows:

$$C_i = (A_{C_i}, \Theta, \Xi) \tag{7}$$

In Equation 7, A_{C_i} is the set of annotations for the configuration, Θ is the set of instances of things and Ξ is the set of connectors between the ports of two things. Each instance $\theta \in \Theta$ has an instance name and a type, i.e., the corresponding thing $\tau_i \in T$. Further, a connector $\xi \in \Xi$ has a starting point, i.e., a thing instance and its port $\theta_a.p_j$, as well as an end point, i.e., another thing instance and its port $\theta_b.p_k$.

Finally, in the adopted domain-specific MDSE methodology with full code generation in an automated manner (see [20,26]), the assumption is that the software model SM contains sufficient amount of information (i.e., it is semantically *complete*) and is syntactically correct (i.e., it is *valid*) according to the meta-model or the context-free grammar of the modeling language, so that the model-to-code transformations can generate the entire implementation of the software for the respective target hardware and software platforms out of the model instance SM. Formally, this means:

$$\exists \Delta, \quad is\_valid(SM) \,\&\, is\_complete(SM) \rightarrow \Delta(SM) \equiv full\_source\_code \tag{8}$$

Here, Δ is a model-to-code transformation, is_valid returns a Boolean value that is true if and only if the model instance is valid, and $is\_complete$ returns a Boolean value that is true if and only if the model instance is complete. The parser and the model editor that we inherited from the ThingML project [49] and extended in this work concerning the DAML functionalities, support the user of the DSML to design a valid and complete model instance that conforms to the meta-model (grammar) of the DSML. The user of the DSML receives the possible error messages, warnings and hints for each of the lines of the textual model instance if applicable.

4.4 AI-enhanced MDSE models (for smart IoT services)

Recall that we define a software model as shown in Equation 5. However, this corresponds to the classic approach to software systems, which tend to exhibit a pre-defined/fixed, stationary or static structure and behavior. Many intelligent systems today, especially for the IoT/CPS use case scenarios, pose a degree of dynamicity, where their structure and/or behavior may change, based on the runtime situation, e.g., the data coming from the surrounding environment. Therefore, either their structure or their behavior, or maybe even both, may be affected by the AI components of the system over the time. The proposed approach in this manuscript deploys ML to let the software model become adaptable. In other words, we propose considering Ψ and/or B as functions of ML models. We call this AI/ML-enhanced software model, Smart Software Model (SSM), and formalize it in the following way:

$$SSM = (A, f_\Psi(DM_1), f_B(DM_2), C) \tag{9}$$

Here, DM_1 and DM_2 are two ML models for learning and controlling the dynamicity of the structure and the behavior of the smart software model, respectively. Thus, the structure and the behavior turn into functions of these ML models.

In the present work, we remove DM_1 for simplicity, and only employ ML for the behavior of the software model. Thus, we consider the simplified form below for our current implementation and validation (DM_2 is renamed to DM):

$$SSM = (A, \Psi, f_B(DM), C) \tag{10}$$

In Equation 10, DM is considered to be the ML model as defined in Equation 1, Φ is the sequence of ML features (attributes) of the ML model, $< \phi_1, \phi_2, \cdots >$, and $\phi_i \in \Gamma$, i.e., the ML features are chosen from the local variables (properties) of the respective thing τ. Note that if the data instances are labeled, i.e., we have a supervised ML task (either classification or regression), as mentioned in Sect. 4.2, the last item of the sequence of ML features Φ is considered as the class label, which shall be predicted by the ML model for new data instances. In practice, the local variables (properties) $\gamma_i \in \Gamma$ may be used in order to store the incoming messages and/or their parameters, so that they can be employed as ML features. Also, they can be used for storing the prediction of the ML model, e.g., to be used in a message, or to trigger an action by the same or another *thing*.

5 ML-quadrat: open-source prototype

In this section, we present our open-source prototype, called ML-Quadrat, which implements the proposed approach. This prototype is used for the case study that is illustrated in Sect. 6.1. The source code, the documentation and a number of examples are available in our Github repository [30] under the terms of the Apache License Version 2.0. Our prototype is built on top of the ThingML project [49], which is also based on the Eclipse Modeling Framework (EMF) and the Xtext framework.

Furthermore, we offer a web-based version of the prototype that is not included in the open-source distribution, but is available upon request for the reproducibility of the results of the empirical evaluation in Sect. 6.2. The web-based interface helps us conduct the experiments with the external evaluators as they do not need to install any software on their side, but simply use the web application in their web browsers.

In the following, we first illustrate the abstract syntax and the concrete syntax of the DSML in Sects. 5.1 and 5.2, respectively. Then, we explain the model-to-code transformations (code generators) that realize the semantics and generate the full source code out of the software model instances, in Sect. 5.3. Further, we elaborate on the DAML matters, specifically on the ML methods that are supported out-of-the-box in the DSML, as well as how to deploy them, in Sect. 5.4. However, we also enable the practitioners (e.g., software developers, data scientists and ML experts) who use the proposed approach, to deploy any arbitrary ML method in the so-called *Black-box ML mode*. This is explained in Sect. 5.5. Finally, in Sect. 5.6 below, we demonstrate a sample IoT service, which is a basic client-server interaction (ping-pong) to highlight the advantages of our work compared to the prior work, ThingML [49].

5.1 Abstract syntax of the DSML

The abstract syntax of the proposed DSML is defined in its grammar that is implemented with the Xtext framework. This is available in the source code repository of the open-source project on Github [30].[3] The Ecore meta-model of the DSML is generated automatically out of the Xtext grammar. As mentioned in Sect. 4.1, Fig. 3 depicts part of the meta-model of the DSML using a UML Class diagram.

As stated in Sect. 4, the *Data Analytics* class that is shown in Fig. 3, which realizes DM in Equation 1 (see Sect. 4.2), was not present in the prior work, ThingML [49]. This is explained in Sect. 5.4 and via the sample IoT service that is illustrated in Sect. 5.6. However, the rest has been adopted from the ThingML project [49] and partially extended to make it compatible with the proposed approach.

Most importantly, the imperative *action* language of ThingML [49] that supports event-driven programming on the state machines, which realize the behavioral models (i.e., B in Sect. 4.3) of *things*, is extended. Using this action language, one may specify which actions (see Π in Sect. 4.3) must be taken upon the occurrence of a particular event, such as upon the receipt of a certain message type on a specific port of a *thing*. For example, a state transition might happen due to the event. Also, various types of actions, such as conditional actions, loop actions, print actions, etc. were possible with the prior work. However, we introduced the new action types that were named in Sect. 4.3 in order to enable the DAML functionalities, namely creating and running the data pre-processing pipeline (i.e., *DA_Preprocess*), conducting ML model training (i.e., *DA_Train*), making predictions using the trained ML models (i.e., *DA_Predict*), and optionally saving the predictions in the dataset (i.e., *DA_Save*).[4] Sect. 5.6 illustrates this using a simple example.

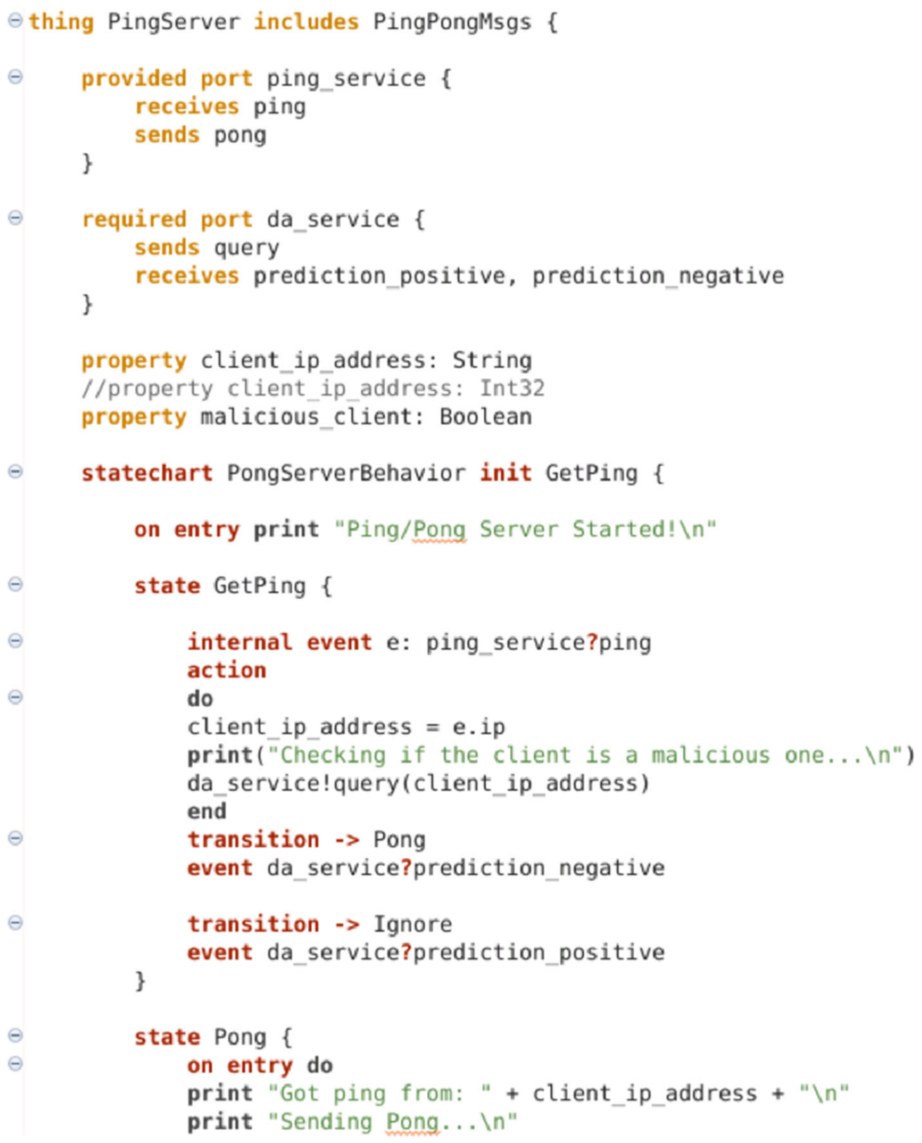
Additionally, *Thing* (*Thing Fragment*), *Platform Annotation*, *Port*, *Message*, *Parameter* and *Property* (see Fig. 3) realize $\tau \in T$, $a \in A$, $p \in P$, $m \in M$, $par(m) \in Par(m)$ and $\gamma \in \Gamma$, respectively, that are mentioned in Sect. 4.3. Last but not least, other elements, such as *Function* and *Property Assign*, as well as those which are not shown in Fig. 3, fall outside of the scope of the focus of this work, thus can be found in the related work, for example, [14,20,34,49].

5.2 Concrete syntax and model editors

We provide three model editors. First, a model editor, based on Xtext, is available in the EMF. This posses a textual concrete syntax, as well as the syntax highlighting and auto-

[3] See https://github.com/arminmoin/ML-Quadrat/blob/master/ML2/language/thingml/src/org/thingml/xtext/ThingML.xtext.

[4] Trivially, DA_Preprocess and DA_Train are skipped in the case of a pre-trained ML model (see Sect. 5.5).



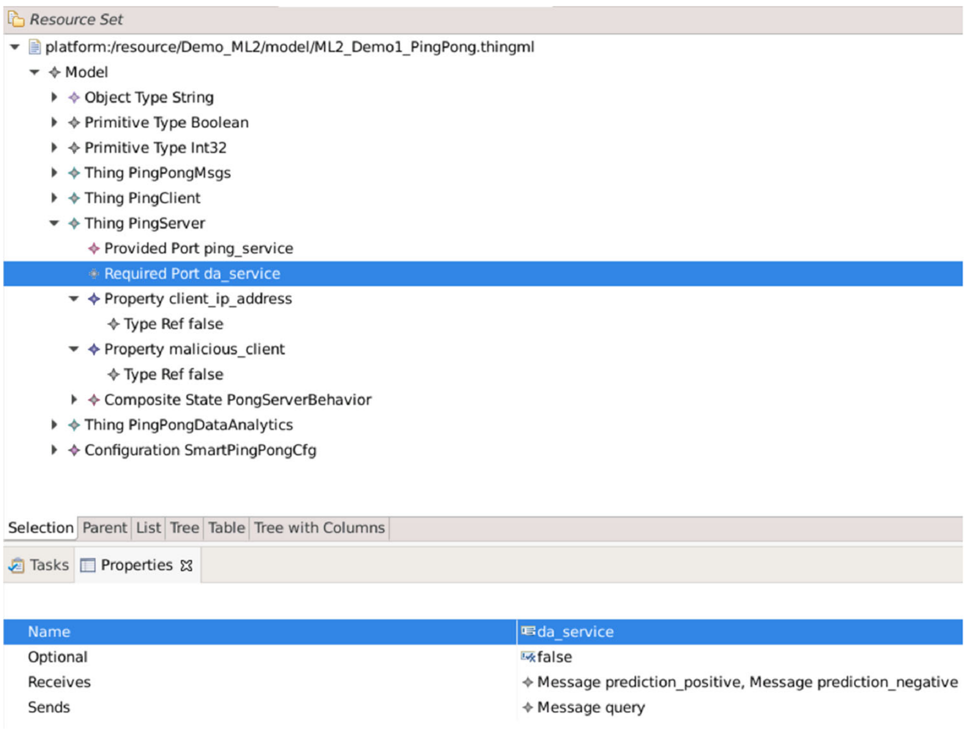
```
thing PingServer includes PingPongMsgs {

    provided port ping_service {
        receives ping
        sends pong
    }

    required port da_service {
        sends query
        receives prediction_positive, prediction_negative
    }

    property client_ip_address: String
    //property client_ip_address: Int32
    property malicious_client: Boolean

    statechart PongServerBehavior init GetPing {

        on entry print "Ping/Pong Server Started!\n"

        state GetPing {

            internal event e: ping_service?ping
            action
            do
            client_ip_address = e.ip
            print("Checking if the client is a malicious one...\n")
            da_service!query(client_ip_address)
            end
            transition -> Pong
            event da_service?prediction_negative

            transition -> Ignore
            event da_service?prediction_positive
        }

        state Pong {
            on entry do
            print "Got ping from: " + client_ip_address + "\n"
            print "Sending Pong...\n"
```

Fig. 5 The textual model editor, showing part of a sample model for the PingPong example (see Sect. 5.6)

complete features, and can give a number of hints and tips to help the practitioner (i.e., the user of the modeling tool) in designing a valid and complete model instance, out of which code generation for a working IoT service with the desired functionality is feasible. Figure 5 shows this model editor. Second, we offer a tree-based (form-based) model editor through the EMF. This is automatically generated in the EMF out of the Ecore meta-model of the DSML, which is itself generated automatically out of the Xtext grammar of the DSML. The tree-based model editor is demonstrated in Fig. 6. While the textual version might be more suitable for developers, the tree-based editor might suite domain experts of the target IoT domains without software development skills well, so that they can modify certain properties of the software model instances, e.g., for the maintenance, upon possible future changes in the requirements. Last but not least, we develop a web-based prototype using the Java Servlets technology and the Xtext web integration. This web application offers a textual model editor with the auto-complete feature and some basic syntax highlighting. This is depicted in Fig. 7.

5.3 Semantics and model-to-code transformations

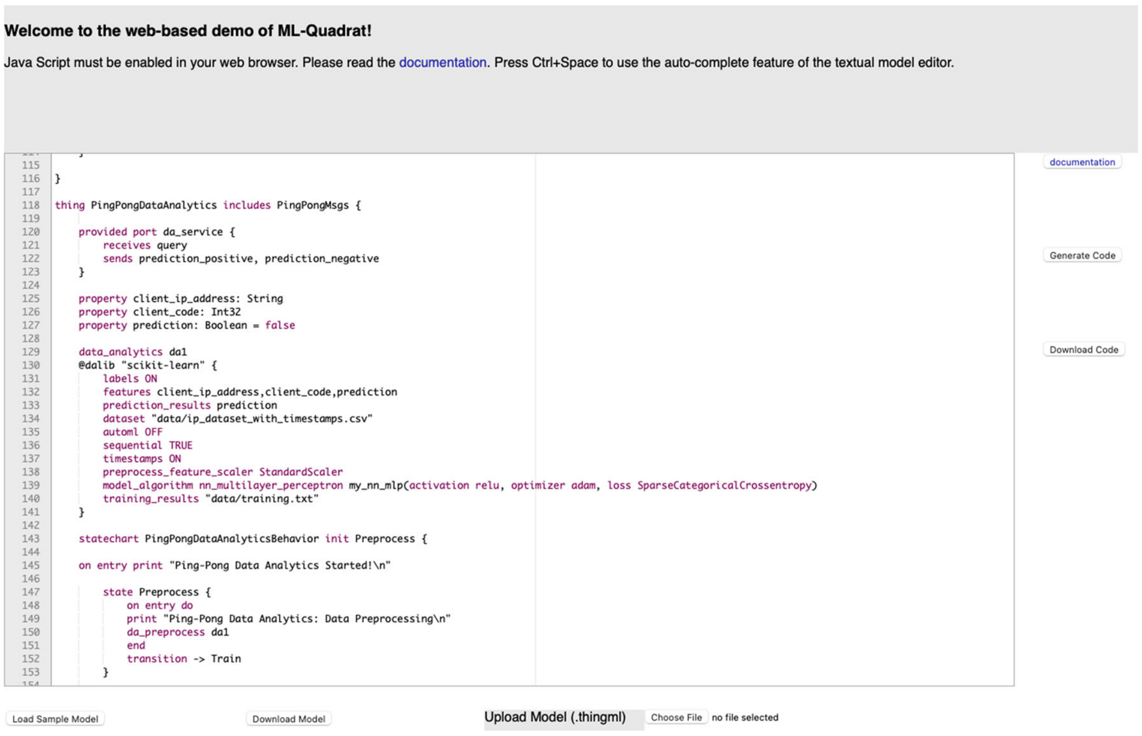
Part of the semantics of the DSML are included in the model-to-code transformations (i.e., Δ in Sect. 4.3), also known as code generators or *compilers*, and the associated constraint-checking mechanisms, which shall execute before the code generation. In addition, another part of the semantics is integrated into the grammar or meta-model, to enable type-checking and enforcing certain constraints at the design-time through the model editors (i.e., before executing the code generators). Furthermore, a number of annotations (i.e., A in Sect. 4.3), e.g., concerning the datatype mappings on



```
Resource Set
▾ platform:/resource/Demo_ML2/model/ML2_Demo1_PingPong.thingml
  ▾ Model
    ▸ Object Type String
    ▸ Primitive Type Boolean
    ▸ Primitive Type Int32
    ▸ Thing PingPongMsgs
    ▸ Thing PingClient
    ▾ Thing PingServer
        Provided Port ping_service
        Required Port da_service
      ▾ Property client_ip_address
          Type Ref false
      ▾ Property malicious_client
          Type Ref false
      ▸ Composite State PongServerBehavior
    ▸ Thing PingPongDataAnalytics
    ▸ Configuration SmartPingPongCfg
```

Name	da_service
Optional	false
Receives	Message prediction_positive, Message prediction_negative
Sends	Message query

Fig. 6 The graphical, EMF tree-based model editor, showing part of a sample model for the PingPong example (see Sect. 5.6)



Welcome to the web-based demo of ML-Quadrat!

Java Script must be enabled in your web browser. Please read the documentation. Press Ctrl+Space to use the auto-complete feature of the textual model editor.

```
}

thing PingPongDataAnalytics includes PingPongMsgs {

    provided port da_service {
        receives query
        sends prediction_positive, prediction_negative
    }

    property client_ip_address: String
    property client_code: Int32
    property prediction: Boolean = false

    data_analytics da1
    @dalib "scikit-learn" {
        labels ON
        features client_ip_address,client_code,prediction
        prediction_results prediction
        dataset "data/ip_dataset_with_timestamps.csv"
        automl OFF
        sequential TRUE
        timestamps ON
        preprocess_feature_scaler StandardScaler
        model_algorithm nn_multilayer_perceptron my_nn_mlp(activation relu, optimizer adam, loss SparseCategoricalCrossentropy)
        training_results "data/training.txt"
    }

    statechart PingPongDataAnalyticsBehavior init Preprocess {

    on entry print "Ping-Pong Data Analytics Started!\n"

        state Preprocess {
            on entry do
            print "Ping-Pong Data Analytics: Data Preprocessing\n"
            da_preprocess da1
            end
            transition -> Train
        }
```

Fig. 7 The web-based prototype, showing part of a sample model for the PingPong example (see Sect. 5.6)

specific target platforms, the choice of specific libraries for DAML, particular communication protocols, and model-to-code transformations are allowed on the modeling layer.

The proposed approach supports code generation in Python and Java. The Python code is responsible for the DAML functionalities of the target IoT services, and supports the APIs of Scikit-Learn [40] and Keras [9] with the TensorFlow [1] backend. The model-to-code transformations are implemented in Java and Xtend (which is a modern variant of Java). They can be found in our Github repository [30].[5]

5.4 Supported ML methods and techniques

The proposed approach allows each *thing* to possess one or more components for DAML. Thus, it supports not only analytics in the cloud, but also edge analytics. Unlike the behavioral component of *things*, i.e., the state machine (statechart), the DAML component, called Data Analytics (DA) is not mandatory. To exhibit DAML capabilities, a *thing* has to include a *data analytics* section in its model. This component that realizes DM in Equation 1, might affect the behavior of the *thing*, modeled via the corresponding state machine. As mentioned before, this corresponds to $f_B(DM)$ in Equation 10. In other words, the behavior of the thing becomes a function of the DAML model. Hence, if a *thing* has a data analytics part, this part shall emerge before the state machine section in the textual model instance, so that the actions specified in the state machine may use and refer to the data analytics component.

Below, we list and briefly explain the possible parameters and options in the said data analytics section of the ML-enhanced software model instances that conform to the meta-model (grammar) of the proposed DSML (see Figs. 9 and 10):

1. **Data_analytics**: This parameter determines the name of the DAML component, e.g., da_1.
2. **Dalib**: The optional *@dalib* annotation specifies the name of the library or framework which must be used for DAML. If this is absent, or it is set to *auto*, or the desired ML method is not implemented in the selected library, the tool will try to automatically select the best option in the Automated ML (AutoML) mode (i.e., if AutoML is ON, see below).
3. **Labels**: This is a binary parameter. If it is ON, it implies that the ML task is supervised. Hence, the last item on the list of features (see below) will be considered as the label. If the data type of that item, defined as the data type of the corresponding property (local variable) of the thing is numeric, e.g., Integer or Float/Double, then the ML task is a regression task. Otherwise, it is a classification task. Furthermore, if the parameter is set to OFF, then the task is unsupervised, e.g., clustering. This parameter also partially realizes I as referred to in Sect. 4.2.
4. **Features**: This is a list of the properties (local variables) of the thing which shall be considered as the ML features (attributes). The local variables might include the messages or parameters of the messages that shall be received from other *things*. As stated above, these are all considered as ML features only if Labels is OFF. In the case that Labels is ON, then the last item is not considered as a feature, but rather as the label (i.e., the class label for classification, or the target value for regression). This parameter realizes Φ as introduced in Sect. 4.2. Simultaneously, the features are properties (local variables) of the corresponding *thing*, thus also partially realizing $\gamma \in \Gamma$ in Sect. 4.3.
5. **Prediction_results**: This parameter determines the property (local variable) of the *thing* in which the prediction result, i.e., the output of the ML model prediction must be stored. Note that the properties were denoted by $\gamma \in \Gamma$ in Sect. 4.3. The value of this property can be then later used in the *actions* of the state machine, in order to let the ML model affect the behavior of the *thing*.
6. **Dataset**: The path of the dataset on the file system that shall be used for training the ML model. This must be a CSV (Comma-Separated Values) file without a header line.
7. **AutoML**: This is a binary parameter indicating whether the AutoML mode must be used. If set to ON, a number of AutoML functionalities will be supported that can assist the practitioner, especially the novice users in the DAML field. By default, this is set to OFF.
8. **Sequential**: This is a Boolean parameter that indicates whether the input data are sequential, e.g., time series, where the order of data instances matter. In this case, shuffling and cross-validation must be avoided. This parameter partially realizes I as referred to in Sect. 4.2.
9. **Timestamps**: This binary parameter states if the data instances have timestamps or not. If this is ON, it has at least two implications. First, if new messages or parameters shall be appended to the dataset (using the DA_Save action), timestamps will be automatically added by the tool. Second, the DAML method will be informed that the first column in the dataset, i.e., the CSV file, must be considered as the timestamp. The expected format is *dd-mm-yyyy HH:MM:SS*, e.g., *17-03-2021 22:49:06* for *March 17, 2021 at 10:49:06 pm*. Obviously, if the timestamps parameter is ON, it is very likely that we are dealing with time series, i.e., sequential data.[6] Therefore,

[5] See https://github.com/arminmoin/ML-Quadrat/tree/master/ML2/compilers/python_java.

[6] Note that the reverse does not always hold, as e.g., DNA data are sequential, but not time series data.

if the sequential parameter is not specified, the AutoML service of the tool, if it is set to ON, will automatically set the sequential parameter to True. However, if the user explicitly states that sequential is False, then the decision will not be overridden. The timestamps parameter also partially realizes I as referred to in Sect. 4.2.

10. **Preprocess_feature_scaling**: This parameter specifies the feature scaling technique that must be used in the data preparation (pre-processing) pipeline. If it is not mentioned, in the case that AutoML is ON, then the best choice of scaling for the respective ML model/algorithm (see below) will be selected. For instance, for the higher performance of Artificial Neural Networks (ANNs), having numerical data that possess a relatively similar scale is an extremely important factor. Thus, for example, standardization (also known as the Z-Score normalization) is automatically set in the AutoML mode. This parameter partially realizes H as set out in Sect. 4.2.
11. **ML Model/Algorithm**: Here, one can specify the particular ML method, including the ML model architecture (family) that must be deployed, e.g., Multi-Layer Perceptron (MLP) ANN, Decision Tree, etc. Additionally, the hyperparameters, e.g., the choice of the error/loss function (e), the learning/optimization algorithm (ζ), the learning rate (lr), etc. might be given in parenthesis. Each family of ML models may have a different set of possible hyperparameters. The auto-complete feature (usually activated by pressing the Control and Space keys together for the textual model editors) helps in finding the possible options. Further, the documentation of the prototype, as well as the API documentations of the target frameworks and libraries (e.g., Scikit-Learn) must be studied. Also, a number of exception handling and logging mechanisms are available to support the user of the tool. This parameter realizes υ, as well as H in Sect. 4.2. The parameters of the ML model (i.e., P in Sect. 4.2) are controlled by the hyperparameters (H) during the learning process.
12. **Training Results**: This is the path of the text file in which the log of ML model trainings shall be stored. The log includes information about the time of each training and the chosen ML model/algorithm. This parameter also partially realizes I mentioned in Sect. 4.2.

We can see how the above-mentioned parameters are used in practice in the basic example provided in Sect. 5.6 below.

Currently, the following ML models and algorithms are supported for supervised ML (i.e., for labeled data) out-of-the-box: (i) Linear Regression, (ii) Logistic Regression for linear classification, (iii) Naïve Bayes (the Gaussian, Multinomial, Complement, Bernoulli and Categorical variants), (iv) Decision Tree (both Regressor and Classifier), (v) Random Forest (both Regressor and Classifier), (vi) the Multilayer Perceptron (MLP) ANN. The APIs of Scikit-Learn are used for the items (i) to (v). However, for the MLP ANN, i.e., (vi) both Scikit-Learn and Keras are supported. By default Keras will be used for this family of ML models. However, the user may explicitly set the library for DAML to Scikit-Learn to override this recommended setting. This is possible through the annotation *dalib* at the *data_analytics* section of the model instance. Moreover, a number of other techniques, e.g., for data preparation, specifically standardization or normalization of the numerical features using various methods are provided.

Moreover, the unsupervised ML methods that are also pre-defined, thus supported out-of-the-box are as follows: (i) K-Means, (ii) Mini-Batch K-Means, (iii) DB-SCAN, (iv) Spectral Clustering and (v) Gaussian Mixture Model. The APIs of the Scikit-Learn library are used for enabling them.

If the desired ML model, algorithm or technique is not pre-defined, one may either extend the open-source prototype (see the online documentation on Github [30]), or use the so-called *Black-box* ML mode (also known as the *hybrid/mixed* MDSE/non-MDSE mode) as described in Sect. 5.5 below. In the latter case, one can bring any arbitrary pre-trained ML model and *connect* it to the MDSE model.

5.5 The black-box ML (hybrid/mixed MDSE/non-MDSE) mode

Suppose that one does not want to use an existing ML method which is already available in our prototype, or has already an existing, pre-trained ML model that they want to deploy. In this case, the *Black-box ML* mode, also called the *hybrid* or *mixed* MDSE/Non-MDSE mode shall be preferred. The drawback here is that the software model will not have any clue about the deployed ML method. Therefore, the ML model seems to the software model as a black-box. However, the advantage is that the user of the DSML will achieve a much higher degree of flexibility concerning ML. Hence, they may, in principle, introduce any pre-trained ML model with any arbitrary architecture and trained with any learning algorithm, and *connect* or *plug* it into the software model.

This can be done by using a parameter, called *blackbox_ml* and setting its Boolean value to true. In this case, using the *model_algorithm* and the *training_results* parameters will not be allowed in the data analytics section of the model instance as no training is required by the AI-enhanced MDSE model. The pre-trained ML model has to be stored in a separate directory. The path of this directory must be given through a parameter, called *blackbox_ml_model* in the data analytics section of the model instance. The pre-trained ML model might have been trained with or without the proposed approach. Moreover, the ML method which is imported from the corresponding DAML library must be specified using a parameter, called *blackbox_import_algorithm*.

5.6 Sample IoT service

In this section, we illustrate an example from the ThingML project [49], and elaborate on the shortcomings of ThingML [49] by showing our extended (*smart*) version of this example. Moreover, this sample IoT service was among the use cases which we originally used to create our DSML and modeling tool. However, the use cases that are provided in the case study in Sect. 6.1 are deployed for validating the proposed approach.

Ping-Pong This example originally came from the ThingML project [49]. In a distributed system, there exist two nodes, called *things*, that are connected to the IoT: (i) the ping client and (ii) the pong server. The *things* are involved in a basic client-server interaction, where the server simply waits for incoming ping messages from the client. As soon as a ping message arrives, the server responds with a pong message.

Smart Ping-Pong We argue that in a real-world scenario with an enormous number of clients, which may send a ping message to the server, the example above can be enhanced via ML, in order to prevent the so-called Distributed Denial of Service (DDoS) attacks. Hence, we introduce a new *thing* that is responsible for DAML, in order to predict if a client is prone to be an attacker or not. Upon receiving a ping message, the server consults this new *thing*, which might even be a *thing fragment* for the server, to see if the ping message shall be responded to with a pong or it would be safer to ignore the request, and perhaps even put the client in a blacklist for a certain period of time. Note that this was not possible using the ThingML DSML [49], whereas our extended version supports DAML at the modeling level. Using the proposed DSML, one may enhance the model instance to become capable of DAML.

Figure 8 depicts the state machines that model the behaviors of the ping client, the pong server and the data analytics server.

Below, we demonstrate part of the model instance for the smart ping-pong example (see Figs. 9 and 10. The full model instance may be found in our Github repository.[7]

Finally, the user documentation available in our Github repository [30] provides further details for creating the desired smart IoT services using our modeling tool, as well as for getting involved in the development of the prototype as a contributor.

[7] See https://github.com/arminmoin/ML-Quadrat/blob/master/ML2/org.thingml.samples/src/main/thingml/ML2_Demo_PingPong.thingml.

6 Validation and evaluation

Section 1 set out the underlying research hypotheses that must be assessed and validate in this work. They lead to the following Research Questions (RQ): **RQ1.** Can we enhance software models in domain-specific MDSE with the capability to automatically produce and train ML models, and deal with them, while maintaining the feasibility of full source code generation? **RQ2.** Will enhancing the software models and integrating them with the ML models contribute to the performance leap of software development in the IoT domain and lead to a higher level of satisfaction of the practitioners who use the proposed approach?

RQ1 that is concerned with the feasibility of the proposed approach is assessed using a case study with two use case scenarios in Sect. 6.1. The research method here involves the implementation, simulation and testing of working examples [36]. Further, RQ2 is assessed through an empirical user evaluation with four external volunteers in Sect. 6.2. Finally, we discuss the possible threats to validity in Sect. 6.3.

6.1 Case study

The selected use case scenarios are from the domain of IoT/CPS, specifically smart energy systems in smart homes. The residential building, which is the data source, is located in the United Kingdom (UK). The data are publicly available through the REFIT datasets [35,45]. We use the data from House/Building 1 from this dataset, which is a single-family dwelling with two inhabitants (a couple). Various sensors have recorded different conditions in their environment over a period of 21 months starting from October 2013. The parameters of interest here are the individual loads (i.e., active power measured in Watts) of the following electrical appliances, as well as the aggregate load, i.e., the total power consumption of the entire house. The samples are recorded at a frequency of 0.125 Hz, i.e., once every 8 seconds. They include the following loads: (i) fridge, (ii) freezer-1, (iii) freezer-2, (iv) washing machine, (v) dishwasher, (vi) computer, (vii) television site, (viii) electric heater, and (ix) washer dryer.

Some electricity providers, especially those who possess smart grids may offer certain discounts if the electrical appliances with higher consumption levels are avoided during the peak hours. Let us assume, there exists a database server that reads the values of the smart meters periodically and stores them for various smart home and ambient assisted living use cases. In this case study, we consider a smart grid that is also granted access to read this database. For them, it is only important whether a certain high energy consuming appliance, e.g., the washer dryer has been turned on during the peak hours or not. The exact power consumption does not really matter. However, due to various reasons, such as sensor malfunctions, power or network outages, or database

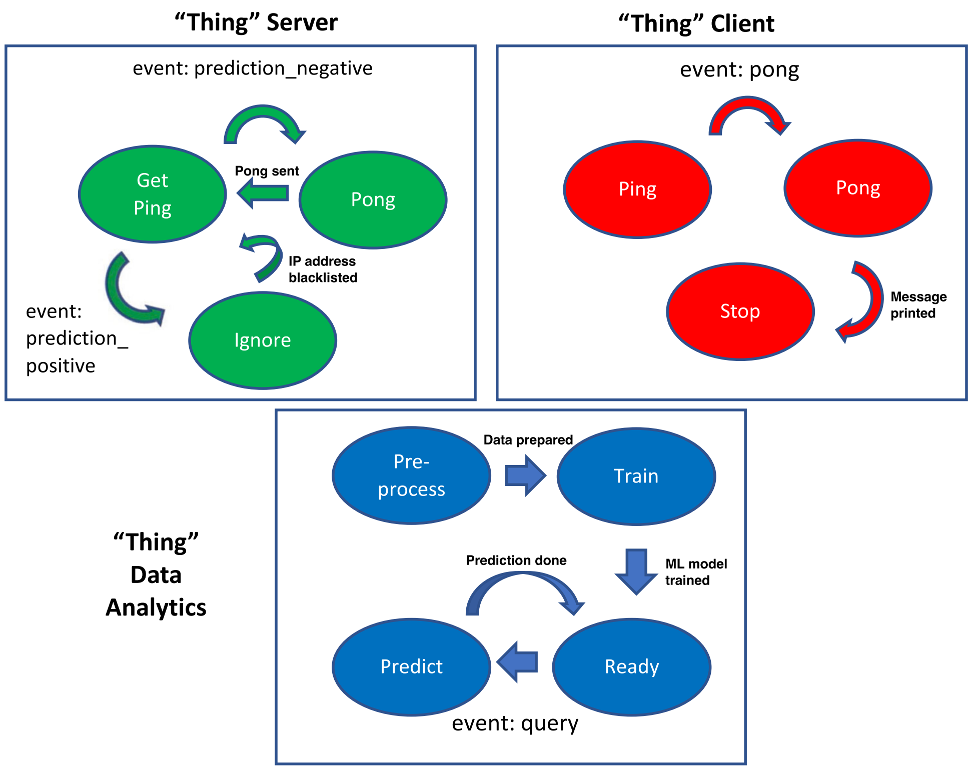


Fig. 8 The state machines modeling the behaviors of the three *things* of the smart ping-pong example

failures, one might be faced with several missing values in the database. There exist different approaches to imputation of missing values in time series data. In this work, we deploy ML models as explained below, in order to predict the state (ON/OFF) of the washer dryer when the data are missing. Nevertheless, if the numerical value of the missing items must be estimated, e.g., in order to improve the quality of predictions of the ML models for other missing values in the future, then regression can be used (see scenario 3 below).

We consider four different scenarios (see below): (i) Classification, (ii) Clustering, (iii) Regression, and (iv) Black-box ML. In each case, the model instance comprises twelve *things*: the nine electrical home appliances above, the said database server, as well as a meter that measures the aggregate load of the entire house, and a DAML server, which is responsible for the predictions of possible missing values in the database. In fact, in practice, the database server and the DAML server may or may not be deployed on the same physical node. Moreover, a gateway could be deployed at the entrance of the house. However, since the IoT advocates direct machine-to-machine communications and direct connections of the devices using their unique addresses [2], we skip the gateway in the present implementation. Figure 11 illustrates the overall architecture of the system.

Each meter sends the active power of the corresponding appliance to the database server every eight seconds. Further, the DAML server sends a query to the database server in a periodic manner (e.g., once every 15 minutes), asking for the latest sensor readings, i.e., the active powers of the nine appliances and the aggregate load of the house. Once the DAML server receives the response of the database server, which includes the ten requested values as message parameters, the DAML server can make a prediction about the missing values that are marked, for example, by *NaN* in the database.

In the following, we illustrate the said scenarios. The full implementations of the respective model instances are included in the supplementary material of this work.[8]

Scenario 1: classification (supervised ML)

We assume that the loads or active powers of the above-mentioned appliances are given together with the aggregate load of the house for time t_i. The task is to predict the binary status (ON/OFF) of the washer dryer at time t_i. The status of the washer dryer is used for the binary class labels of samples in the training dataset. We let the software model train the supervised ML model using 80% of the available data. Thus, we keep 20% of samples for testing the ML model. This is common practice in ML. For example, the Scikit-Learn [40] library offers the *train_test_split* method that is widely used [19]. This method, by default, dedicates 25% of the data to the test dataset unless another value is set for the *test_size* parameter. However, many practitioners simply follow the *Pareto Principle* that is also called

[8] See https://doi.org/10.5281/zenodo.5501356.

Fig. 9 Part of the model instance of the smart ping-pong example

```
/* This is a part of the model instance. The full model
    instance is available in the Git repository on Github. */

thing PingPongDataAnalytics includes PingPongMsgs {
/* The messages are not shown here, but defined in a thing
    fragment, called PingPongMsgs. */

 provided port da_service { /* This port communicates with the
      da_service port of pingServer. */
  receives query /* This port may receive a query message from
       pingServer. */
  sends prediction_positive, prediction_negative /* This port
      may send a response to pingServer. The response might be
       positive, i.e., malicious prediction or negative, i.e.,
       non-malicious prediction. */
 }

 /* The properties are the local variables of the thing. */
 property client_ip_address: String /* The IP address of
     pingClient is a String. */

 property client_code: Int32 /* This is just a secret integer
     code that is shared between pingClient and pingServer or
     alternatively a serial ID number for the ping message. */

 property prediction: Boolean = false /* This Boolean property
      shall store the prediction of the DAML model and is
     initialized as false here. This mean, by default, the
     client is non-malicious. */

 data_analytics da1 /* Please see Section 5.4. */
 @dalib "scikit-learn" {
    labels ON
    features client_ip_address,client_code,prediction
    prediction_results prediction
    dataset "data/ip_dataset.csv"
    automl OFF
    sequential TRUE
    timestamps OFF
    preprocess_feature_scaler StandardScaler
    model_algorithm nn_multilayer_perceptron my_nn_mlp
    (activation relu, optimizer adam, loss
        SparseCategoricalCrossentropy)
    training_results "data/training.txt"
}

 statechart PingPongDataAnalyticsBehavior init Preprocess {
  /* The statechart specifies the behavior of this thing.
      Since this thing is responsible for DAML, its behavior
      can be modeled via a Finite-State Machine (statechart)
      that has four states: preprocess, train, ready and
      predict. Initially, the Preprocess state is necessary to
       do the data preparation. */

  on entry print "Ping Pong Data Analytics Started!\n"
  state Preprocess {
    on entry do
    print "Ping Pong Data Analytics: Data Preprocessing\n"
    da_preprocess da1 /* This action carries out the actual
        data preprocessing / preparation. */
    end
    transition -> Train /* This leads to the transition of the
         state machine (statechart) to the next state, Train.
        */
  }
```

Fig. 10 Part of the model instance of the smart ping-pong example (continued)

```
state Train {
 on entry do
  print "Ping Pong Data Analytics: Training\n"
da_train da1 /* This action performs the training of the
    DAML model. */
 end
 transition -> Ready /* Once the training is done, the thing
      shall switch to the Ready (or idle) state to simply
     keep waiting for the incoming queries. */
}

state Ready {
 on entry do
  print "Ping Pong Data Analytics: Ready for Prediction\n"
 end
 transition -> Predict
 event m: da_service?query
 /* As soon as a message is received on the da_service port,
      the thing must switch to the Predict state. */
 action do
  /* Additionally, the following actions must be taken. */
  client_ip_address = m.client_ip  /* First, the value of
      the message parameter, called client_ip needs to be
      stored in the thing property (local variable)
      client_ip_address. */
  client_code = m.client_code /* Second, the value of the
      message parameter, called client_code must be stored
      in the thing property (local variable) client_code. */
 end
}

state Predict {
on entry do
print "Ping Pong Data Analytics: Predicting\n"
da_predict da1(client_ip_address, client_code) /* This
    action asks the DAML model to make a prediction. */
if(prediction==false)
da_service!prediction_negative() /* If the prediction is
    false, send the prediction_negative message to
    pingServer, stating that pingClient is not likely to be
    an attacker. */
else
da_service!prediction_positive() /* Otherwise, send the
    prediction_positive message to pingServer, stating that
    pingClient is prone to be an attacker.  */
end
transition -> Ready /* In any case, switch back to the Ready
     (i.e., idle) state. */
on exit da_save da1 /* This optional action results in
    appending the prediction to the dataset (CSV file). */
}
 }
}
```

the 80/20 rule, which states that in most cases, 80% of effects come from 20% of causes. Further research will be needed to see if a different split would yield better or worse results. Moreover, please note that we do not shuffle the data, i.e., we do not randomly split the data since they are sequential (namely time series) data and the order of the data instances matters. The supervised ML method deployed in this example is the Multi-Layer Perceptron (MLP) classifier from the Artificial Neural Networks (ANN) family with one hidden layer of size 100, the *Relu* activation function,

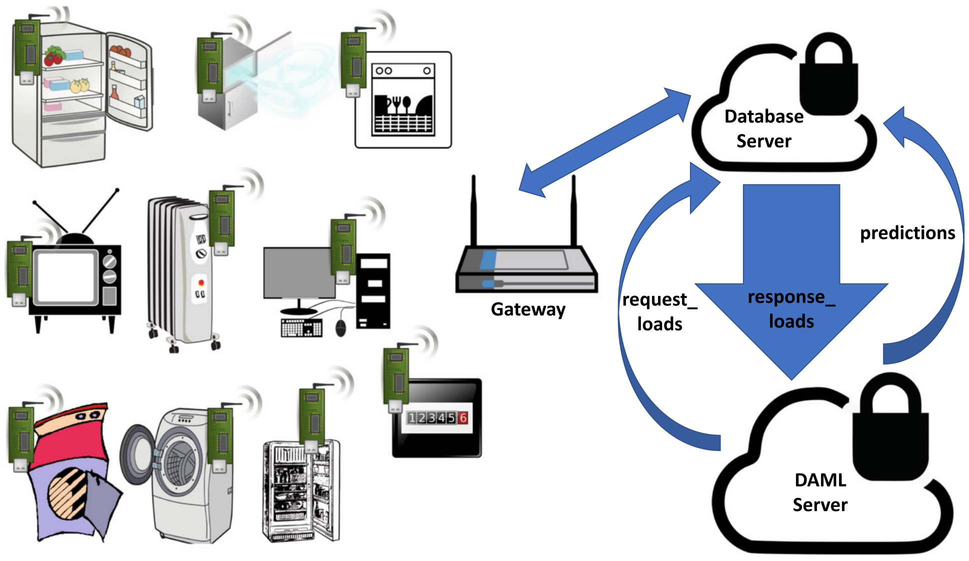


Fig. 11 The overall architecture of the target system for the case study

the *Adam* optimizer, the *Sparse Categorical Cross Entropy* loss function, and the default values for the rest of the arguments/parameters of this ML method in the Scikit-Learn library.

The created software model instance has 545 lines in the textual form. The model-to-code transformations generate 4, 032 Lines of Code (LoC) out of this. The generated source code contains 3, 875 lines of Java code and 157 lines of Python code. The latter is responsible for the DAML functionalities and is seamlessly integrated with the Java code using the Java Process Builder API. Note that the scenarios below also exhibit the same number of LoC since we generate the APIs of the DAML library (in this case Scikit-Learn) and only the name of the ML method, as well as certain parameters/arguments change (but the number of the lines of code remain unchanged).

Furthermore, training the said ML model took 3, 552 seconds, and it performed with 100% accuracy on the unseen test data (the ground truth comes from the mentioned open data, i.e., the REFIT datasets [35,45]). Typical ML performance metrics include but are not limited to accuracy, precision, recall and F1-Measure. In the case of binary classification, with the positive and negative classes, these are defined as follows:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{11}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{12}$$

$$\text{Recall} = \text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{13}$$

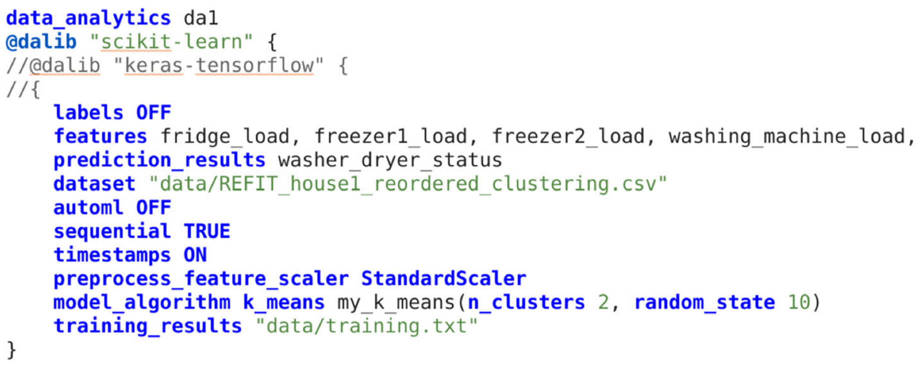
$$\text{F1-Measure} = \frac{2.\text{Precision}.\text{Recall}}{\text{Precision} + \text{Recall}} \tag{14}$$

In the equations above, TP, TN, FP, and FN are the True-Positive, True-Negative, False-Positive and False-Negative number of cases, respectively.

In the said experiment, the other ML performance metrics, namely the precision, recall and F1-Measure were 99.9%, 100% and 99.9%, respectively. The high performance was foreseeable given the fact that the ML task was not challenging for the MLP ANN classifier that is a highly capable one. In any case, the focus of this case study is not on measuring the performance of the ML methods since we only deploy the APIs of the target libraries for this purpose. The focus is rather on showing the feasibility of the proposed approach through the working examples. Hence, the reported performance figures in this section serve only for information purposes and are not supposed to contribute to the validation.

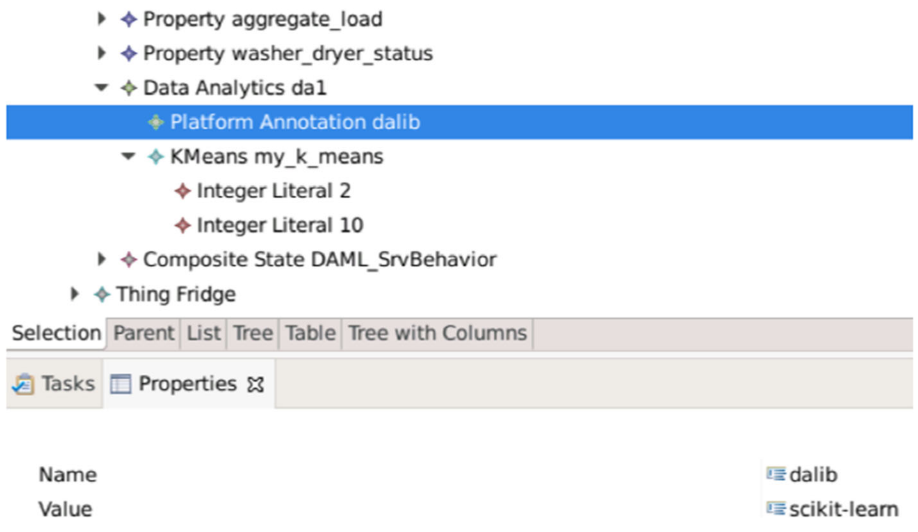
Scenario 2: clustering (unsupervised ML)

Again, we assume that the loads or active powers of the above-mentioned appliances are given together with the aggregate load of the house for time t_i. Also, we have the same task, namely predicting whether the washer dryer is ON or OFF at time t_i. However, the training dataset this time has no labels for the data instances. This means, we do not know which sample in the training data belongs to the case when the washer dryer has been OFF and which one corresponds to the ON state of the washer dryer. The goal is to use the available data to train a clustering ML algorithm that can group the instances into two clusters: cluster A and cluster B. Cluster A, which we call it cluster 0 in the dataset corresponds to the OFF state of the washer dryer. In contrast, cluster B, which we call it cluster 1 in the dataset means the washer dryer has been ON. Note that 0 and 1 here are just the labels or names



```
data_analytics da1
@dalib "scikit-learn" {
//@dalib "keras-tensorflow" {
//{
    labels OFF
    features fridge_load, freezer1_load, freezer2_load, washing_machine_load,
    prediction_results washer_dryer_status
    dataset "data/REFIT_house1_reordered_clustering.csv"
    automl OFF
    sequential TRUE
    timestamps ON
    preprocess_feature_scaler StandardScaler
    model_algorithm k_means my_k_means(n_clusters 2, random_state 10)
    training_results "data/training.txt"
}
```

Fig. 12 The data analytics part of the software model instance in textual form



```
▸ ◆ Property aggregate_load
▸ ◆ Property washer_dryer_status
▾ ◆ Data Analytics da1
    ◆ Platform Annotation dalib
  ▾ ◆ KMeans my_k_means
      ◆ Integer Literal 2
      ◆ Integer Literal 10
▸ ◆ Composite State DAML_SrvBehavior
▸ ◆ Thing Fridge
Selection | Parent | List | Tree | Table | Tree with Columns
Tasks | Properties

Name     dalib
Value    scikit-learn
```

Fig. 13 The data analytics part of the software model instance in the EMF tree-based editor

for the clusters and have no numerical interpretations. The unsupervised ML method deployed in this example is the K-Means clustering method with the values 2 and 10 provided for the arguments/parameters regarding the desired number of clusters and the random state of the algorithm, respectively. For the rest of the arguments/parameters of the method, the default values for this method in the Scikit-Learn library are considered.

Furthermore, training the said clustering model took only 13 seconds (extremely fast compared to the supervised model above), and it performed with 92% accuracy on the unseen test data.

Figures 12 and 13 show a small part of the corresponding software model instance using the textual and the tree-based views of the concrete syntax in the Eclipse Modeling Framework (EMF).

Scenario 3: regression (supervised ML)

This use case scenario is very similar to the first scenario above. However, instead of predicting the ON/OFF class labels, the task is to predict the numerical values of the active power of the washer dryer. We deployed the MLP ANN Regressor in Scikit-Learn.

For measuring the performance of regression, the typical error measures, Mean Absolute Error (MAE), also known as the L1-Norm, as well as the *Mean Squared Error (MSE)*, also known as the *L2-Norm* or the *Euclidean Norm* are common choices. These are defined as follows:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} \mid \hat{y}_i - y_i \mid \tag{15}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{16}$$

Here, n is the number of data instances, $\hat{y}_i$ is the predicted numerical label for the $i - th$ data instance, and y_i is the actual numerical label for this data instance.

The achieved MAE and MSE in the experiment above were 10.1 and 29, 962.1, respectively.

Fig. 14 The data analytics part of the software model instance that shows the black-box ML mode in textual form

```
data_analytics da1
@dalib "scikit-learn" {
//@dalib "keras-tensorflow" {
//{
    labels OFF
    features fridge_load, freezer1_load, freezer2_load, washing_machine_load,
    prediction_results washer_dryer_status
    blackbox_ml true
    blackbox_ml_model "pre_trained/pre_trained_ml_model.pickle"
    blackbox_import_algorithm "from sklearn.cluster import KMeans"
    //blackbox_label_encoder "pre_trained/pre_trained_label_encoder.pickle"
}
```

Table 2 The expertise levels of the evaluators

Eval.#	Software engineering	DAML	MDSE	IoT/CPS
1	High	High	Low	Low
2	Medium	Low	Medium	Medium
3	High	High	Low	Low
4	Medium	Low	Medium	Medium

Scenario 4: black-box ML

We train an unsupervised ML model without using the proposed approach. Thus, we develop the ML part manually. However, we use the same dataset. Then, we connect the pre-trained ML model to the software model using the black-box ML mode. The rest is the same as the unsupervised ML scenario above (including the performance). Figure 14 demonstrates a small part of the respective software model instance.

6.2 Empirical evaluation

We ask four external experts in software engineering to use and evaluate our DSML through a number of experiments in a four-hour one-on-one video call over the Internet with short breaks in between. Two of them have a background in ML as well. Moreover, two of them work in academia and the other two work in the industry. Further, two out of four possess a PhD, whereas the rest have a Master's degree. Last but not least, they all belong to the age group of 25-39 years old, and one of them is a female. Table 2 illustrates the self-reported levels of expertise of the evaluators in various fields, collected before carrying out the user experiments.

The evaluators are familiar with Java and Python programming. However, none of them has any background knowledge in the deployed DSMLs (neither in ThingML [49] nor in ours). During the four-hour sessions with the evaluators, we first deliver a 50-minute tutorial for using the proposed DSML, as well as the prior work on which we built our DSML, namely ThingML [49]. To this aim, we have already prepared a few samples, including a 'HelloWorld' example. Moreover, we offer them our web-based prototype (see Sect. 5). We ask each evaluator to work on two tasks in three *modes*: (a) Using pure manual software development (i.e., no MDSE); (b) Using the prior work, namely ThingML [49]; (c) Using the proposed DSML. We change the orders of the tasks, as well as the orders of the *modes* for the four participants to avoid any bias and make the experiments fair. Both tasks are based on the case study set out in Sect. 6.1 above. However, in the first task, we ask the evaluator to use supervised ML (specifically classification as in the first scenario in Sect. 6.1), whereas in the other task we ask for unsupervised ML (specifically clustering as in the third scenario in Sect. 6.1). Recall that the use case scenario that was depicted in the case study in Sect. 6.1 involved 12 *things*. Implementing each of them gives the evaluator one point. An incomplete, but satisfactory implementation might result in 0.25, 0.5 or 0.75 points, depending on the completeness and correctness of the implementation. Also, implementing the DAML component of each *thing* (if it should have any) has one extra point (which may be granted only partially, depending on the status of the implementation as mentioned before). Table 3 summarizes the obtained points of the evaluators for all tasks and *modes*. For each task, they have 75 minutes time, which includes 25 minutes per each *mode*. During the experiments, they may maintain their access to their resources, e.g., the tutorials on the Internet and their own prior work, to make the experiments similar to the real-world practices of software developers and ML experts (e.g., data scientists).

For the pure manual developments (i.e., in *mode* a), we ask them to use Python for the ML part, with the APIs of the Scikit-Learn library and the ANN MLP classifier for the supervised task (i.e, task 1), as well as the K-Means clustering method for the unsupervised task (i.e., task 2). For the rest of their manual implementations, they are free to choose between Python and Java. However, in *mode* b, they must deploy our web-based interface that offers the DSML and the code generators of ThingML (i.e., the prior work [49]) too, and implement the ML part manually in Python, so that their Python code can call the Java APIs of the generated Java code. Finally, in *mode* c, no manual development will occur. They only use our web-based interface that offers our DSML and code generators to create their model instances.

Table 3 The scores of the 4 evaluators (Eval. #1-4)

Task-mode	Eval. #1	Eval. #2	Eval. #3	Eval. #4	Total score	Max.
1-a	1	1	2	2	6	52
1-b	0.5	5.5	3.25	10.25	19.5	52
1-c	2.25	2	5	12	21.25	52
2-a	0	1	2.25	1	4.25	52
2-b	2.25	1.25	4	2	9.5	52
2-c	2.25	1.25	5	5	13.5	52

The full source code can be generated automatically. For the ML part, we currently generate Python code that is automatically integrated with the rest of the generated code.

As illustrated in Table 3, using the proposed approach (see the rows 1-c and 2-c) has increased the scores of the evaluators, both compared to the prior work (see the rows 1-b and 2-b) and to the pure manual software development (see the rows 1-a and 2-a). The last column illustrates the total sum of the maximum possible scores for all of the evaluators, whereas the one before last column shows the total sum of the scores achieved by the evaluators in the experiments. Thus, we argue that the proposed approach may contribute to the improvement of the software development process efficiency. According to the experiments, the performance leap has been around 25% on average, compared to the prior work (i.e., ThingML [49]) and around 236% compared to the pure manual software development (see Sect. 6.3). We believe that the selected ML task was rather easy and only for one platform. One should be able to perceive a greater value in our proposed approach once heterogeneous IoT cloud and edge platforms need to be deployed. In the conducted experiments, many evaluators just started working on the DAML part from the very beginning. This should have resulted in a smaller difference between the productivity of software development in *modes* b and c. Nevertheless, even 25% productivity leap can still justify deploying the proposed approach.

Finally, we ask the opinions of the evaluators about their overall experience and satisfaction through a brief questionnaire at the end of the session. Compared to the prior work (ThingML [49]), two evaluators (#1 and #4) rated their level of satisfaction about the proposed approach as *high*. Moreover, the other two evaluators chose the option *medium*. The options were *high*, *medium* and *low*. In contrast, when compared to pure manual software development, one of the evaluators selected the option *low*. However, they emphasized that this answer is given the current exercises since the selected IoT platforms were not heterogeneous and it was rather easy for them to implement it manually. The other evaluators chose the answer options *high*, *medium* and again *high* concerning this question. Hence, all in all, we argue that the proposed approach may contribute to the user experience and satisfaction of the practitioners.

6.3 Discussion and threats to validity

The conducted experiments in Sects. 6.1 and 6.2 validated the first and the second hypotheses, respectively. First, we provided the proof-of-concept and showed the feasibility of enhancing MDSE models in the DSM methodology for developing IoT services with ML models if Artificial Intelligence (AI), in particular ML capabilities are required. Second, we verified empirically that the ML-enhanced software models used with the proposed approach can lead to performance leap for software development in the IoT domain and a higher satisfaction level of the practitioners compared to the prior model-driven work, namely ThingML [49] and the pure manual software development.

Recall that we claimed that DAML models (i.e., DM in Equation 1) may affect the behavioral models of software systems (i.e., B in Equation 4. This was shown formally through Equation 10. The way that the ML models, e.g., the ones corresponding to the above-mentioned use case scenarios for the case study in Sect. 6.1 can affect the behavioral models of software is through the use of the action type *DA_Predict* in the actions of the state machines (statecharts) that specify the behavioral models of the respective *things*. The supplementary material of this paper[9] shows the use of this action type for all of the depicted use case scenarios in Sect. 6.1.

One key strength of this work for the SE community is expected to be that they gain access to the DAML methods and techniques out-of-the-box and can deploy them in their software models for the IoT. However, the major limitation is that ML methods cannot perform well if their hyperparameters are not tuned properly and/or the data that are used for training them are not prepared well. Therefore, more advanced AutoML features, e.g., concerning automated or semi-automated hyperparameter choices and tuning, as well as data preparation, e.g., for high-dimensional and non-i.i.d (independent and identically distributed) data are necessary (see Sect. 7).

Further, a major advantage of this work for the DAML community is assumed to be that they can become involved in large-scale IoT projects easier as they will be able to work

[9] See https://doi.org/10.5281/zenodo.5501356.

with the abstract software models that are easier to understand, adapt and use for them. Moreover, they may introduce any desired pre-trained ML model with any arbitrary architecture, learning algorithm and technique. This shall bring them a lot of flexibility as they will not be limited to the pre-defined options. However, the implication for them (as well as for the SE community) is that they have to be familiar with the DSML of the modeling tool and be willing to model their desired software using this DSML.

There exist a number of possible threats to the validity of the research results. First, we validated the first research hypothesis through a case study in Sect. 6.1. We showed the feasibility of the proposed approach via a number of working examples with different use case scenarios. Although this is a well-established research method in engineering research (see, e.g., [36]), we only had one overall case study domain (namely, smart energy data for smart home) and the selected case study and vertical application domain might not be representative enough for the entire domain of IoT/CPS. Thus, the generalized conclusions made here about the entire target domain might not be rigorous.

Second, the empirical evaluation conducted in Sect. 6.2 involved only four professionals. Consequently, the conclusions drawn may not hold for a larger sample group. In particular, the ideal research design should have involved randomized controlled experiments. However, our study was neither randomized nor had any control group. In contrast, we used convenience sampling and invited four independent, external volunteers to participate in our empirical evaluation. Further, the tasks chosen for the experiments were only two rather similar programming tasks with simple DAML requirements and no combination of heterogeneous resource-constrained IoT devices. This was due to the time and resource constraints for the experiments with the experts, but might be biased. Ideally, the tasks should have been more diverse and possibly more tasks would have been required, in order to be fair to different participants with different strengths. Additionally, we swapped the task and mode orders. However, we cannot rule out possible biases as a result of working on one task in a certain mode, e.g, using our DSML, and then in the following slot on the same task, but in a different mode, e.g, via pure manual software development. Also, it is clear that the time constraint may have an impact on the performance of evaluators in these tasks. For example, the manual task (namely, the *a mode*) is expected to require more time than the tasks in the *b mode* and the *c mode*. Therefore, allotment of the same amount of time may not work ideally in all the modes. Finally, this was an exploratory user study/pilot study and a more rigorous evaluation with more evaluators is required in the future. Hence, the achieved preliminarily results might not be sufficient to perform a quantitative analysis.

7 Conclusion and future work

In this manuscript, we proposed a novel approach to marry the models in Artificial Intelligence (AI), specifically Machine Learning (ML), with the models in Software and Systems Engineering (SSE), particularly in Model-Driven Software Engineering (MDSE) following the Domain-Specific Modeling (DSM) methodology with full code generation. We showed how MDSE models can be integrated with ML models, thus become capable of producing and/or dealing with ML models. We concentrated on the Internet of Things (IoT) and Cyber-Physical Systems (CPS) domains, where both ML and MDSE are widely used. However, the proposed Domain-Specific Modeling Language (DSML), which is built based on the prior work in the literature, ThingML [14,20,34,49], is not tied to any specific vertical application domain. Similar to the ThingML project [49], we also supported full code generation in an automated manner through our ready-to-use model-to-code transformations. In addition to inheriting the code generators of ThingML [49], we introduced a Python and Java code generator that can generate the APIs of the Scikit-Learn [40] and Keras [9] libraries and frameworks for ML.

The two research questions concerning the feasibility and the impact of the proposed approach were validated through the case study and the empirical user evaluation in Sect. 6, respectively. It transpired that the proposed approach can lead to a higher performance and a better experience of the practitioner (e.g., software developer) for developing smart, data-driven IoT services. However, as stated in Sect. 6.3, a large-scale user study in the form of a randomized controlled experiment is required in the future.

The proposed approach has a number of limitations that can be addressed in future work. First, we supported supervised and unsupervised ML, whereas semi-supervised ML in which the data are only partially labeled is also desirable and beneficial in many use cases. Second, the pre-defined ML methods can be extended, e.g, with kernel methods, such as Support Vector Machines (SVM), Probabilistic Graphical Models (PGM), as well as more advanced ANN architectures, such as Long Short-Term Memory (LSTM) for Sequence-to-Sequence and End-to-End ML models. Third, more target platforms, programming languages and libraries can be supported. For instance, a pure Java code generator that uses the Java libraries WEKA or MOA (Massive Online Analysis) for DAML can be beneficial. Similarly, a pure Python code generator that does not have to mix the Python and Java codes for the IoT service might be advantageous for certain use cases, where Java might not be desired or useful. Last but not least, more advanced AutoML functionalities, e.g., concerning data preparation, as well as automated or semi-automated hyperparameter tuning will be very useful, in particular for software developers who might be novice in the field of DAML.

Further, we implemented one specific variant of the proposed approach in Sect. 4, where the DAML model may have an impact on the behavioral model of the software. However, it would be interesting to explore and realize other setups, e.g., where the DAML model might affect the structure of the software model, or even both the behavior and the structure. For instance, Pigem [11] studied how ML can be employed to *learn* finite-state machines. Hence, there might be some potential in adopting such approaches and integrating them with the proposed approach to make the MDSE models even more intelligent. In fact, this would mean letting them learn the behavioral model of the software, in part or completely, on their own, using the existing data, instead of having the practitioner (i.e., the user of the DSML) specify it.

Finally, by enabling every *thing* to possess one or more DAML components, we have enabled the modeling infrastructure for deploying edge analytics and federated learning. This paves the way for future work to provide a complete solution to supporting federated ML in the proposed DSML.

Acknowledgements This work is partially funded by the German Federal Ministry for Education and Research (BMBF) through the Software Campus initiative (project ML-Quadrat). We are sincerely grateful to our external evaluators, Fatma Bozyigit from Izmir Bakircay University, Turkey, Burak Karaduman from University of Antwerp, Belgium, Andrei Mituca from DriotData UG, Germany, as well as the anonymous evaluator. The authors would like to also thank Stephan Rössler from Software AG and Marouane Sayih (alumnus of the Technical University of Munich) for their collaboration and support.

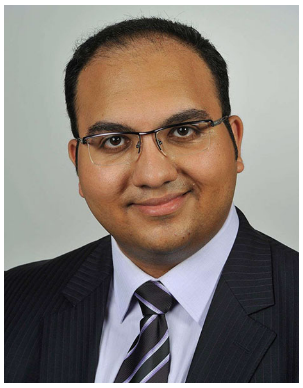
Funding Open Access funding enabled and organized by Projekt DEAL.

Data availability The authors are committed to the open science initiative. Therefore, the entire research data are available as open data under the terms of the Creative Commons Attribution 4.0 International license at https://doi.org/10.5281/zenodo.5501356.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, GS, Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J, Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V, Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, software available from tensorflow.org (2015)
2. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
3. Berners-Lee, T., Hendler, J.: Publishing on the semantic web. Nature **410**, 1023–4 (2001). https://doi.org/10.1038/35074206
4. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Kötter, T., Meinl, T., Ohl, P., Thiel, K., Wiswedel, B.: KNIME—the Konstanz Information Miner: Version 2.0 and Beyond. SIGKDD Explor Newsl. **11**(1), 26–31 (2009). https://doi.org/10.1145/1656274.1656280
5. Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin (2006)
6. Bishop, C.M.: Model-based machine learning. Philos. Trans. R. Soc. A **371**(1984), 1–17 (2013). https://doi.org/10.1098/rsta.2012.0222
7. CERN The birth of the web. https://home.cern/science/computing/birth-web. Accessed 06-09-2021
8. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C, Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. (2015) arXiv:1512.01274
9. Chollet, F., et al. Keras (2015). https://keras.io
10. Combemale, B., Kienzle, J., Mussbacher, G., Ali, H., Amyot, D., Bagherzadeh, M, Batot, E., Bencomo, N., Benni, B., Bruel, JM., Cabot, J., Cheng, B., Collet, P, Engels, G., Heinrich, R., Jézéquel, JM., Koziolek, A., Mosser, S., Reussner R., Wimmer, M.: A hitchhiker's guide to model-driven engineering for data-centric systems. IEEE Software. https://doi.org/10.1109/MS.2020.2995125 (2020)
11. de Balle Pigem, B.: Learning finite-state machines – statistical and algorithmic aspects. Ph.D. thesis, Universitat Polit'ecnica de Catalunya, Spain. https://borjaballe.github.io/other/phdthesis.pdf (2013)
12. DiNucci, D.: Fragmented future. Print **53**(4), 32 (1999)
13. DMG (2021) Data Mining Group (DMG). http://dmg.org. Accessed 09-03-2021
14. Fleurey, F., Morin, B., Solberg, A., Barais, O.: Mde to manage communications with and between resource-constrained systems. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 349–363 (2011)
15. Fouquet, F., Morin, B., Fleurey, F., Barais, O., Plouzeau, N., Jezequel, J.M.: A dynamic component model for cyber physical systems. In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, Association for Computing Machinery, New York, NY, USA, CBSE '12, pp. 135–144. https://doi.org/10.1145/2304736.2304759 (2012)
16. Geisberger, E., Broy, M. (eds) Living in a networked world. Integrated research agenda Cyber-Physical Systems (agendaCPS). acatech STUDY, Herbert Utz Verlag, Munich, Germany (2014)
17. Greer, C., Burns, M., Wollman, D., Griffor, E.: Cyber-physical systems and internet of things. https://doi.org/10.6028/NIST.SP.1900-202 (2019)
18. GreyCat: Next-Gen Live Analytics using Temporal Graph. (2018) https://github.com/datathings/greycat. Accessed 02-09-2021
19. Géron, A.: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, CA 95472, USA (2019)
20. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: 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, MODELS '16 (2016)
21. Hartmann, T., Moawad, A., Fouquet, F., Le Traon, Y.: 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), p. 180. https://doi.org/10.1109/MODELS.2017.32 (2017)
22. Hartmann, T., Fouquet, F., Moawad, A., Rouvoy, R., Traon, Y.L.: Greycat: efficient what-if analytics for data in motion at scale. arXiv:1803.09627 (2018)
23. Hartmann, T., Moawad, A., Fouquet, F., Le Traon, Y.: The next evolution of MDE: a seamless integration of machine learning into domain modeling. Softw. Syst. Model. (SoSyM) **18**, 1285–1304 (2019). https://doi.org/10.1007/s10270-017-0600-2
24. HEADS: Heterogeneous and Distributed Services for the Future Computing Continuum. (2015) https://cordis.europa.eu/project/id/611337. Accessed 01-09-2021
25. Helwig, N., Pignanelli, E., Schütze, A.: Condition monitoring of a complex hydraulic system using multivariate statistics. In: 2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings, pp. 210–215, https://doi.org/10.1109/I2MTC.2015.7151267 (2015)
26. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation, 1st edn. Wiley, Hoboken (2008)
27. ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description. Standard, ISO/IEC/IEEE (2011). https://www.iso.org/standard/50508.html
28. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd edn. Cambridge University Press, USA. http://www.mmds.org (2014)
29. Minka, T., Winn, J.M., Guiver, J.P., Zaykov, Y., Fabian, D., Bronskill, J.: Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer (2018). Accessed 08-09-2020
30. ML-Quadrat: ML2. (2020). https://github.com/arminmoin/ML-Quadrat. Accessed 12-09-2020
31. Moin, A., Rössler, S., Günnemann, S.: Thingml+: augmenting model-driven software engineering for the internet of things with machine learning. In: Hebig R, Berger T (eds) 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, CEUR-WS.org, CEUR Workshop Proceedings, vol. 2245, pp. 521–523 (2018) http://ceur-ws.org/Vol-2245/mde4iot_paper_5.pdf
32. Moin, A., Rössler, S., Sayih, M., Günnemann, S.: From things' modeling language (thingml) to things' machine learning (thingml2). In: Guerra, E., Iovino, L. (eds) MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings, ACM, pp. 19:1–19:2 (2020) https://doi.org/10.1145/3417990.3420057
33. Morin, B., Fleurey, F., Husa, K.E., Barais, O.: A generative middleware for heterogeneous and distributed services. In: 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), pp. 107–116 (2016) https://doi.org/10.1109/CBSE.2016.12
34. Morin, B., Harrand, N., Fleurey, F.: Model-based software engineering to tame the IoT jungle. IEEE Softw. **34**(1), 30–36 (2017). https://doi.org/10.1109/MS.2017.11
35. Murray, D.: A data management platform for personalised real-time energy feedback. In: Proceedings of 8th International Conference on Energy Efficiency Domestic Appl Lighting (EEDAL), pp. 1–15 (2015)
36. Newman, W.: A preliminary analysis of the products of HCI research, using pro forma abstracts. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, CHI '94, pp. 278–284 (1994)
37. ONNX: Open Neural Network Exchange. (2021). https://github.com/onnx. Accessed 09-03-2021
38. Papatheocharous, E., Axelsson, J., Andersson, J.: Issues and challenges in ecosystems for federated embedded systems. In: Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems, pp. 21–24. https://doi.org/10.1145/2489850.2489854 (2013)
39. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch. In: NIPS-W (2017)
40. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
41. PFA: Portable Format for Analytics. (2021) http://dmg.org/pfa/index.html. Accessed 09-03-2021
42. Pivarski, J., Bennett, C., Grossman, R.L.: Deploying analytics with the portable format for analytics (PFA). In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Association for Computing Machinery, New York, NY, USA, KDD '16, pp. 579–588. https://doi.org/10.1145/2939672.2939731 (2016)
43. PMML: Predictive Model Markup Language. (2021) http://dmg.org/pmml/v4-4-1/GeneralStructure.html. Accessed 09-03-2021
44. RapidMiner (n/a) Depth for Data Scientists, Simplified for Everyone Else. https://rapidminer.com. Accessed 08-09-2021
45. REFIT: REFIT datasets. https://www.refitsmarthomes.org/datasets/. Accessed 01-09-2020(2015)
46. Schaetz, B.: The role of models in engineering of cyber-physical systems—challenges and possibilities. In: CPS20: CPS 20 years from now-visions and challenges, CPS Week (2014)
47. TensorBoard (n/a) TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard. Accessed 08-09-2021
48. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints arXiv:1605.02688 (2016)
49. Things Modeling Language: ThingML (2015) https://github.com/TelluIoT/ThingML. Accessed 29-04-2020
50. Time and causality in interactive distributed systems (lecture slides) (2008). https://www5.in.tum.de/~huckle/Broy.pdf . Accessed 06-09-2021
51. Wang, H., Yeung, D.Y.: A survey on bayesian deep learning. ACM Comput. Surv. (2020). https://doi.org/10.1145/3409383



Armin Moin is a doctoral researcher of the chair for Data Analytics and Machine Learning (DAML) at the department of Informatics of the Technical University of Munich (TUM) in Germany. His research is at the intersection of Machine Learning and Software Engineering, in particular domain-specific Model-Driven Software Engineering (MDSE) for smart Internet of Things (IoT) services and Cyber-Physical Systems (CPS). He is a reviewer of conferences and journals, such as PLOS ONE, SoftwareX and Expert Systems with Applications.

Atta Badii is a professor in Secure Pervasive Technologies at the Department of Computer Science of the University of Reading, United Kingdom (UK). He has established a track record of key contributions to over 40 projects, including more than 30 large-scale collaborative research programmes. He has coordinated over 12 large-scale international projects funded by a variety of sources including the UK Engineering and Physical Sciences Research Council, the UK Ministry of Defence Grand Challenge and the European Commission.

Moharram Challenger is a tenure-track assistant professor in the Department of Computer Science at the University of Antwerp, Belgium. His research interests include domain-specific modeling languages, model-driven engineering, multi-agent systems, cyber-physical systems and the IoT. Previously, he was a postdoc researcher at the University of Antwerp, Belgium. Moreover, he received his Ph.D. in IT from the International Computer Institute at Ege University, Turkey in February 2016. He has led several national and international research projects, e.g., in Turkey and Belgium.

Stephan Günnemann is a professor at the Department of Informatics of the Technical University of Munich in Germany and Director of the Munich Data Science Institute. His research focuses on making machine learning reliable, thus, enabling its safe and robust use, e.g., for graphs/networks and temporal data. He acquired his doctoral degree in Computer Science at RWTH Aachen, Germany in 2012. Also, he has been an associate of Carnegie Mellon University, USA, and a researcher at the Simon Fraser University in Canada and Siemens AG in Germany.