Received June 9, 2021, accepted June 19, 2021, date of publication June 25, 2021, date of current version July 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3092528

SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments

JOSÉ A. BARRIGA, PEDRO J. CLEMENTE, ENCARNA SOSA-SÁNCHEZ, AND ÁLVARO E. PRIETO
Quercus Software Engineering Group, Department of Computer Science, University of Extremadura, 10003 Cáceres, Spain

Corresponding author: José A. Barriga (jose@unex.es)

This work was supported in part by the Ministry of Science and Innovation (MCI), for the State Research Agency (AEI) under Project RTI2018- 098652-B-I00, in part by the Government of Extremadura, Council for Economy, Science and Digital Agenda under Grant GR18112, in part by the European Regional Development Fund (ERDF) under Project IB20058, and in part by the Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

This work did not involve human subjects or animals in its research.

ABSTRACT Internet of Things (IoT) is being applied to areas as smart-cities, home environment, agriculture, industry, etc. Developing, deploying and testing IoT projects require high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software. New projects require high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software before each system can be developed. In addition, the systems should be developed to test them, which implies time, effort and development costs. However, in order to decrease the cost associated to develop and test the system the IoT system can be simulated. Thus, simulating environments help to model the system, reasoning about it, and take advantage of the knowledge obtained to optimize it. Designing IoT simulation environments has been tackled focusing on low level aspects such as networks, motes and so on more than focusing on the high level concepts related to IoT environments. Additionally, the simulation users require high IoT knowledge and usually programming capabilities in order to implement the IoT environment simulation. The concepts to manage in an IoT simulation includes the common layers of an IoT environment including Edge, Fog and Cloud computing and heterogeneous technology. Model-driven development is an emerging software engineering area which aims to develop the software systems from domain models which capture at high level the domain concepts and relationships, generating from them the software artefacts by using code-generators. In this paper, a model-driven development approach has been developed to define, generate code and deploy IoT systems simulation. This approach makes it possible to design complex IoT simulation environments and deploy them without writing code. To do this, a domain metamodel, a graphical concrete syntax and a model to text transformation have been developed. The IoT simulation environment generated from each model includes the sensors, actuators, fog nodes, cloud nodes and analytical characteristics, which are deployed as microservices and Docker containers and where elements are connected by using publish-subscribe communication protocol. Additionally, two case studies, focused on smart building and agriculture IoT environments, are presented to show the simulation expressiveness.

INDEX TERMS IoT systems, IoT simulation, fog computing, model-driven development, model to text transformation, data analysis.

I. INTRODUCTION

The Internet of Things (IoT) is widely applied in several areas such as smart-cities, home environments, agriculture, industry, intelligent buildings, etc. [46]. Usually, these IoT environments require using hundreds of sensors and actuators shared throughout these areas which are generating a vast amount of data. Data must be suitably stored, analysed and published using Big Data or Stream Processing techniques. Big Data or Stream Processing techniques must be applied

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaolong Li.

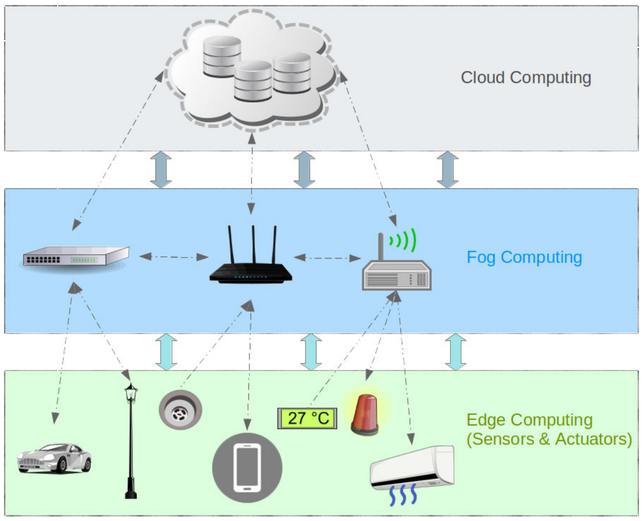


FIGURE 1. **IoT architecture: Cloud, Fog and Edge computing.**

to conveniently store and analyse published data. Taking into account where data are processed and stored, the IoT environment architecture can be defined by several computing layers (Edge, Fog and Cloud computing (see Figure 1) [30]). Edge layer is defined closer to data generators, Fog layer resides on top of the edge and act as intermediary layer with limited storing and processing capabilities and Cloud layer is defined with full storing and processing capabilities. Thus, the development of IoT systems requires the management and integration of conveniently heterogeneous technologies such as devices, actuators, databases, communication protocols, stream processing engines, etc. As a consequence, in order to implement, deploy and test the IoT systems a high investment must be made in time, money and effort.

Simulating IoT environments is one way to decrease this initial investment because the users can measure and dimension the artefacts needed to deploy and interconnect the systems. Thus, these artefacts can include several kinds of devices from sensors or actuators to NoSQL databases, messaging brokers or stream processor engines. However, although there are several simulation environments for wireless sensor networks (WSN), there is a lack of IoT simulator tools for designing IoT environments at a high level that enable modeling this kind of systems by using the domain concepts and relationships. In addition, there is a lack of IoT simulation tools that makes it possible to deploy the IoT system on multiple nodes in order to test the communications among the system's elements and where complex IoT components such as databases, complex event processing or message brokers can be suitable deployed and tested.

Currently, there is a lack of methodologies and tools to simulate IoT systems and allow users to properly describe the IoT environment. Currently, not only tools are needed but also methodologies to guide the simulation designing and simulation process of IoT environments. So, both methodologies and tools to simulate IoT systems are interesting research areas. should be developed. Thus, while methodologies

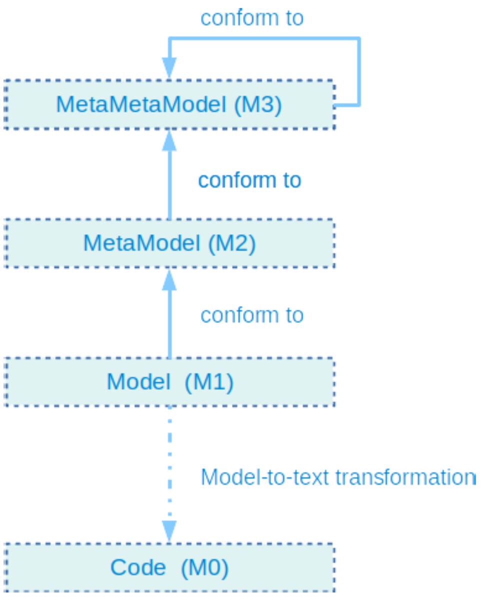


FIGURE 2. **Model-driven development. Four layers of metamodeling.**

would allow developers to describe the steps and the characteristics to simulate IoT systems, the tools would help to design and execute the IoT environment simulated in sandbox environments. These tools should take into account the main IoT characteristics including heterogeneous devices (sensors and actuators), heterogeneous communication mechanisms such as publish-subscribe communication protocol, analysis from information generated, storing of information, etc. However, an IoT environment is a broad and heterogeneous concept which involves heterogeneous technologies such as communication protocols such as publish-subscribe communication protocol, databases, analysis tools, etc. Not only should IoT methodologies and tools be designed and developed, but they should also be carried out using software engineering good practices.

Model-Driven Development is an emerging software engineering research area that aims to develop software guided by models based on Metamodeling technique. Metamodeling is defined by four model layers (see Figure 2). Thus, a Model (M1) is conform to a MetaModel (M2). Moreover, a Metamodel conforms to a MetaMetaModel (M3) which is reflexive [2]. The MetaMetaModel level is represented by well-known standards and specifications such as Meta-Object Facilities (MOF) [29], ECore in EMF [48] and so on. A MetaModel defines the domain concepts and relationships in a specific domain in order to model partial reality. A Model (M1) defines a concrete system conform to a Metamodel. Then, from these models it is possible to generate totally or partially the application code (M0 - code) by model-to-text transformations [44]. Thus, high level definition (models) can be mapped by model-to-text transformations to specific technologies (target technology). Consequently, the software code can be generated for a specific technological platform, improving the technological independence and decreasing error proneness.

So, Model-Driven Development (MDD) is proposed to tackle this heterogeneous technology (devices, actuators, complex event processing engines, notification technology, publish-subscribe communication protocol, etc.). Model-Driven Development [16], [20], [40], [43] increases the *abstraction level* where the software is implemented, focusing on the domain concepts and their relationships. These domain concepts (sensors, actuators, fog nodes, cloud nodes, etc.) and their relationships are defined by a model (M1), conform to a metamodel (M2), which can be analysed and validated using MDD techniques. Besides, the IoT environment code, including all the artefacts needed, can be generated from a model (M1) using model-to-text transformations, decreasing error proneness and increasing the user's productivity.

The main contributions of this paper include:

- This work shows that using Model-Driven Development techniques are suitable to develop tools and languages to tackle successfully the complexity of heterogeneous technologies in the context of IoT simulation environments.
- A methodology called *SimulateIoT* to describe each step needed to define an IoT simulation environment and execute it.
- A Model-Driven solution that supports the methodology proposed. It facilitates the development of each methodology phase by defining a *SimulateIoT* metamodel (M2), a graphical concrete syntax (graphical editor) to define models (M1) and a model-to-text transformation towards the code generation for specific IoT simulation environment (M0 - code). It includes the code generation to execute the IoT simulation. Furthermore, the IoT system generated can be deployed.
- An IoT deployment process that makes it possible to deploy the simulation based on microservices which are deployed on Docker containers, including components such as databases, complex-event processing engines or message brokers.
- The application of *SimulateIoT* to two case studies focused on different kinds of IoT systems (Smart buildings and Agricultural environment).

The rest of the paper is structured as follows. In Section 2, we give an overview of existing IoT simulation approaches centered on both low level and high level IoT simulation environments. In Section 3, we present the *SimulateIoT* methodology. Section 4 describes *SimulateIoT* design and implementation phases including the *SimulateIoT* metamodel, the graphical editor and the model-to-text transformation developed. In Section 5 two case studies are presented. Finally, Section 6 elaborates on the limitations of the presented approach before Section 7 concludes the paper.

II. RELATED WORKS

IoT environments and IoT simulation environments have been developed using several strategies with different targets and distinct abstraction levels. The abstraction levels are not related to the different IoT Architecture levels (Edge, Fog or Cloud layers) but also the concepts and relationships used to design the simulation at the IoT architecture level. For instance, you could use concepts to low level such as memory, network capabilities and use tools to manage this kind of configuration or using high level concepts such as Fog Node, Cloud Node or Complex Event Processing, engines, NoSQL storage where low level concepts could be transparently managed. Additionally, using high level abstractions could be used to generate code for specific technological targets. In this sense, among other, the concepts analysed for each different related work include: the abstraction level used to define the IoT environment, Edge modeling capabilities, Fog modeling capabilities, Cloud modeling capabilities, Complex Event Processing, Big data support, and Code generation support. So, in the following subsections several IoT simulation approaches are reviewed that focus on the different abstraction levels used for their definitions. So, we are examining i) Low level IoT simulation environments; and ii) High level IoT simulations environments and IoT modeling based on model-driven development. The former are based on defining sensors and actuators close to hardware (Contiki-Cooja, OMNeT++, IoT-Lab), so these proposals foster the knowledge of hardware, networks or energy consumption characteristics. The latter (COMFIT, CupCarbon, IoTSim) focus on defining IoT context and environments at a level of high abstraction.

A. LOW LEVEL IoT SIMULATION ENVIRONMENTS

Contiki-Cooja [42] is a network simulator tool based on the Contiki operating system. It is implemented in Java and allows users to define large and small Contiki *motes* (a node in a sensor network) which can be deployed throughout the network. Relevant information about the network such as mote output or time-lines could be obtained after the simulation execution. Note that a mote can be defined ad-hoc using the motes templates. Obviously, these simulations are defined at a low level focusing on hardware and network issues more than IoT contexts or communication patterns such as publish-subscribe.

OMNeT++ [51] is a general network simulator adapted to simulate IoT networks. It offers a Domain Specific Language for modelling the IoT context including aspects related to routers, switchers, routing protocols or network protocols (IPv4, IPv6, etc.). This is a powerful simulator focused on analysing low level aspects of network issues. It uses components and component-based compositions to define network simulations. This approach focuses on defining IoT environments at a low level of abstraction closed to hardware. So, it is not centered on describing the IoT environment and high-level component relationships. Therefore, simulating wide IoT environments could be tedious and error prone.

IoT-Lab [35] is a platform which allows deploying compiled WSN (Wireless Sensor Network)/IoT applications on a large WSN infrastructure. The applications can be installed on different types of sensors and can be developed on the

Contiki operating system, among others. Thus, the goal of the authors is showing how both local and global energy consumption can be precisely monitored.

Tossim [24] is a Wireless Sensor Network (WSN) simulator tool used over TinyOS. It can simulate thousands of nodes while it is able to capture the network behaviour with accuracy. It emulates the underlying raw hardware behaviour. Thus, the aim of this approach is simulating low level motes without defining communication patterns such as publish-subscribe or without using pattern data generations.

B. HIGH LEVEL IoT MODEL-DRIVEN DEVELOPMENT AND SIMULATION ENVIRONMENTS

This section includes both IoT development environments and IoT simulation environments which are based on graphical or textual domain concept descriptions, or model-driven technologies. There are several IoT metamodels [8], [36], [47] to model IoT systems, and usually the application code is partially generated from these models.

In [8] a Domain Specific Language has been defined to model IoT environments, taking into account several IoT concepts such as *devices, and input and output properties*. Its goal is modelling IoT environments and generating code for a specific platform such as KNX/EIB. Although it is not related to IoT simulation, it uses model-driven techniques in order to tackle designing IoT systems and it can be used for quick IoT system prototyping.

Another approach based on Model Driven Development [9] makes it possible to model complex event processing for near real-time open data. This approach is interesting because they present a methodology and a domain specific language to define models in order to model open-data sources, the processing nodes and the notifications agents. However, this approach does not focus on modeling and simulating IoT environments.

COMFIT [15] was a cloud environment to develop the Internet of Things system. It used model-driven techniques included in the Model-Driven Architecture (MDA) specification [18]. For instance, a model-to-text transformation towards code generation for specific operating system targets (for instance, Contiki or TinyOS operating systems) was implemented. It defined several UML Profiles such as PIM:UML Profile and PSM:UML Profile, a model to model transformation from PIM models to PSM models, and a model-to-text transformation. So, authors used well-known UML tools to model the IoT Systems, however they did not define an ad-hoc metamodel for IoT, but used UML diagrams such as detailed activity diagrams.

On the other hand, IoTSuite [36], [47] defined a high level domain specific language in order to model IoT environments including concepts such as *regions, sensors, actuator, storage, request, action, etc*. Thus, it joined computational services with spatial information related to regions such as *buildings* or *floors*. Several modelling languages were defined to model these kinds of systems: Srijan Vocabulary Language (SVL), Srijan Architecture Language (SAL) and Srijan Deployment Language (SDL). Then, a code generation process allows generating the application code. Although IoTSuite makes it possible to define IoT environments, it isn't an IoT simulator.

In [39] a component-based approach for the Web of Things was presented. They defined a Model Driven Development process to model Web of Things (WoT) systems by using model-driven techniques such as meta-modelling and model transformations. Thus, they defined a metamodel for WoT which related *Physical Entities* such as *Sensors* or *Actuators* with *Visual Entities* such as components deployed on a system. These models can automatically turn into code skeletons. However, this metamodel does not allow defining specific domain concepts related to simulation or storage issues, among others.

Other approaches focus on simulating IoT systems proposing specific tools [4], [27], [45]. Thus, CupCarbon [27] defined an IoT Simulator environment which allows users to describe IoT contexts using a graphical editor. For instance, a mote could be added on a map like Google Maps, taking into account parameters such as action radio. It implements an ad-hoc language to manage the sensor's communication and the business logic. It can execute simulations including the reactions to random events. So, although this approach allows describing IoT simulation issues, it does not allow describing the storage information or the complex communication protocols such as publish/subscribe using messages brokers.

IoTSim [53] is an extension of CloudSim [6] that focuses on simulating IoT applications in cloud environments. It supports and enables IoT big data processing using the MapReduce model in the cloud. However, in order to execute the IoT application to be simulated, users should implement the workflow that IoTsim proposes, including Datacenter configuration, IoTDataCenterBroker, JobTracker, etc. Obviously, this approach offers a framework to execute IoT applications on cloud, however it does not offer a designing tool to easily define the artefacts necessary to be deployed on the IoTSim. Additional extensions to CloudSim deal with the analysis and use of BigData. BigDataSDNSim [1] allows the simulation of the big data management system YARN, its related programming models MapReduce, and SDN-enabled networks in a cloud computing environment. On the other hand, IoTSim-Edge is another CloudSim extension specialised in EdgeComputing [22]. In this way, this simulator allows defining and simulating elements such as EdgeNodes (EdgeDevice, EdgeDataCenter, EdgeBroker), IoTDevices (sensors and actuators) and their characteristics such as battery consumption, mobility, communication protocol, etc. These simulators deal with relevant aspects of the IoT in detail, allowing the simulation of IoT environments or parts of these environments in a very realistic way. However, these works lack a high-level abstraction graphical interface to visualise and model the architecture of the environment. On the other hand, they lack a module capable of validating a configured environment before its simulation. Therefore, although these simulators are able to simulate an

IoT environment with high detail they need to define the configuration simulation environment using JSON files and Java code, which raise the learning curve. For instance, each sensor type needs to be implemented before being used on a configuration file. Finally, they do not model high concepts related to Complex Event Processing or they facilitate code-generation. Another important aspect is related with Simulation deployment which is carried out in the same machine without deploying a service-oriented architecture (common architecture where an IoT system is deployed), that is, all IoT aspects are simulated so code cannot be re-used for real implementation proposes.

Using the approach in [4] the developers can test their cloud and on premise MQTT (Message Queuing Telemetry Transport) [33] application for functional and load testing. So, it allows deploying IoT environments focused on using sensors, actuators and MQTT servers. This tool allows users to define sensors and actuators and publish/subscribe concepts to define the IoT environment. It defines a set of template sensors to be used in order to model the IoT environment. Besides, data generation can be defined by the users following several data patterns such as *concrete value*, *range values*, *random set* or based on *time & client*. However, the IoT environment does not make it possible to define stream rules to react to event patterns.

In [45] an IoT simulator was defined. It was written in Java and it allowed defining IoT simulations including agents, places and the context therein. The main steps to define a simulation included: i) defining the environment, ii) developing the behaviour and iii) packing and deploying it all together. The IoT system behaviour should be implemented ad-hoc using Java. So, this simulator required high expertise implementing Java agents. Furthermore, this approach did not resolve how to manage or analyse the device data.

Viptos [7] is an integrated graphical development and simulation environment for *TinyOS-based* [21] wireless sensor networks. Developers can model algorithms with the graphical framework included in *Viptos* and generate their code in *nesC* [17]. Besides, users can define environments to simulate the behaviour of these algorithms. These environments could have features such as communication channels, network topology (the nodes where the algorithms will be tested) and physical characteristics (low-level, such as OS interruptions) of the environment. In short, this framework allows application developers to easily transition between high-level simulation of algorithms to low-level implementation and simulation. However, due to the characteristics mentioned, this framework works with a low level of abstraction. For that reason, the application developers that use this framework need to know low level concepts about it and the domain which can simulate. In addition, modelling an extensive simulation could be complex and the use of simulators with a higher level of abstraction would be more suitable.

VisualSense [3] is a modelling and simulation framework for wireless sensor networks that builds on and leverages Ptolemy II [12]. This framework supports the modelling of sensor nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems among others characteristics. Besides, this framework supports the modelling of dynamic networks where nodes can change their connectivity in run-time. It's worth mentioning that the communication between nodes is via events with timestamps [5]. Finally, the models can be simulated and visualised at run time. However, this simulator is focused on modeling networks at a low level of abstraction, without including high level concepts based on *Cloud/Fog* computing, publish-subscribe communication protocols and so on.

To sum up, although there is a wide literature focus on defining the IoT environment and IoT simulation environments at different abstraction levels, several issues should be additional treated including fog computing, cloud computing, storage data, communication protocols or data analysis (see Table 1. The following sections describe the *SimulateIoT* methodology and tools which are proposed to tackle the complexity of the description and execution of IoT simulation environments.

III. SimulateIoT METHODOLOGY

This section describes the Simulation Methodology which has two phases, *simulation description* and *simulation execution*, as shown in Figure 3.

First, *simulation description* includes the following steps:

1) *Data and WSAN specification*: Users should define the wireless sensors and actuator network (WSAN) to identify the device characteristics (including their data inputs and outputs) The wireless sensors and actuator network (WSAN) should be defined to identify the device characteristics (including their data inputs and outputs). It allows defining the edge computing layer formed by sensors and actuators;
2) *Fog/Cloud computing spec* includes defining devices with different process capacities. For instance, these nodes can define how and where data should be stored, including the database characteristics (SQL database, NoSQL database, etc.);
3) *Processing data specification* defines the communication schemas, that is, the communication protocols to connect the devices and nodes previously identified. In addition, this phase should make it possible to describe how data should be processed using multiple technologies such as big data or stream processors.

Next, *Simulation execution phase* includes aspects related to the hosts where the IoT devices and nodes should be deployed. So, it includes where databases, message brokers, stream processors, etc. should be deployed. As a consequence, these aspects allow the IoT to tailor the simulation, adapting it to real situations.

IV. SimulateIoT TOOLS

This section describes the tools designed to implement the SimulateIoT methodology (defined in Section III) which

TABLE 1. Key elements of the related works summarized (IoT simulation and model-driven development).

Simulator	Field	Target	Abstraction level	Principal Modelable Components	Edge Modeling	Fog Modeling	Cloud Modeling	Complex Event Processing	Big Data support	IoT Code generation
Contiki Cooja [42]	IoT Network Simulation	Simulate large IoT Networks whre deploy TelosBDevices which run your own code	Low	RPL Network: Border Router, UDP Server, TelosBDevices (where run your own code)	Yes	No	No	No	No	No
Omnet++ [51]	General network simulations	Model communication networks	Low	Router, Switchers, choose routing or network protocols, define components or components compositions (to test in the network)	Yes	No	No	No	No	Yes
IoTLab [35]	IoT Devices simulation	Study the behavior of IoT devices	Low	WSN devices	Yes	No	No	No	No	No
Tossim [24]	IoT Network Simulation	Simulate low level motes without defining communication patterns such as publish-subscribe or without using pattern data generations.	Low	TinyOs WSN, Discrete event queue, TinyOs hardware abstraction components, Radius of action of the WSN, ADC models	Yes	No	No	No	No	No
HAAIS-DSL [8]	General IoT simulation	model IoT environments and generate code for a specific platform such as KNX/EIB.	High	Sensors (Output), Actuators (Input), other devices (I/O)	Yes	No	No	No	No	Yes
COMFIT [15]	IoT Application simulations	Simulate IoT Applications to study them behaviour	High	IoT applications	Yes	No	Yes	Yes	No	Yes
IOTSuite [36, 47]	IoT Applications development	Facilitate the development at each stage of the life cycle of an IoT application	High	Sensors, Actuators, logic of the above devices, deployment of the devices, connection between components.	Yes	No	No	Yes	No	Yes
Ruppen et al. [39]	WoT (Web of Things)	Facilitate the development of Smart Devices for the WoT	High	Actuators, Sensors and other secondary components, model the logic and the hardware of the devices, connection between components.	Yes	No	No	No	No	Yes
CupCarbon [27]	IoT WSN simulations	Study the behaviour of a network and its costs.	High	WSN devices, mobility of them, radius of actuation, geolocation, communication between sensors.	Yes	No	No	No	No	No
IoTSim [53]	IoT Big Data Applications	Deploy IoT Big Data Applications on the Cloud and study their behaviour	High	Devices and process should be modeled by the developers at low abstraction level.	Yes	No	Yes	No	Yes	No
BevyWise [4]	IoT Simulations	Simulate Iot environments to study their behaviour.	High	Sensors, actuators and MQTT Servers.	Yes	No	Yes	Yes	Yes	No
Siafu [45]	IoT Simulations	Simulate Iot environments to study their behaviour.	High	The map of the environment with multiple places (with environment context variables) where the IoT devices will be working. The behaviour of each device.	Yes	No	No	No	No	No
Viptos [7]	WSN Simulations.	Simulate Wireless Sensor Networks and study their behaviour	Low	Network components such as devices, the logic or the algorithms wich will be run on them; Network topology, communication channels, hardware features and physical characteristics such as OS interrumptions	Yes	No	No	Yes	No	No
VisualSense [3]	Network lifetime	Simulate Wireless Sensor Networks and study their behaviour	Low	Sensor nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems among others characteristics; Dynamic network behaviour where nodes can change their connectivity in run-time	Yes	No	No	Yes	No	No
Clemente and Tello [9]	CEP and Open Data	Apply CEP to Real-time Open Data	High	CEP engine mechanism wich can be applied to Open Data sources	No	No	No	Yes	No	No
BigDataSDNSim [1]	Big Data and Cloud computing	Simulation of Big Data processes in the Cloud	High	Cloud environment, Big Data management system YARN, SDN-enabled networks, Map Reduce processes	No	No	Yes	No	Yes	No
IoTSim-Edge [22]	Edge computing	Simulation of IoT Edge layer	High	EdgeNodes (EdgeDevice, EdgeDataCenter, EdgeBroker), IoTDevices (sensors and actuators) and their characteristics such as battery consumption, mobility, communication protocol, etc.	Yes	No	Yes	Yes	No	No
SimulateIoT	IoT Simulator	Simulate IoT environments and study their behaviour	High	IoT environment compose of devices such as Sensors, Actuators, FogNodes, and CloudNodes. Sensors: Syntethic data generation for publications, publication intervals, Topics where publish data; Actuators: Topics from wich receive data, actions; FogNodes and CloudNodes: Storage (such as MongoDB), Complex Event Processing (such as EsperTech), rules for CEP, notifications to other devices, redirection of data to other nodes in the network, Type of MQTT Broker, etc.	Yes	Yes	Yes	Yes	Yes	Yes

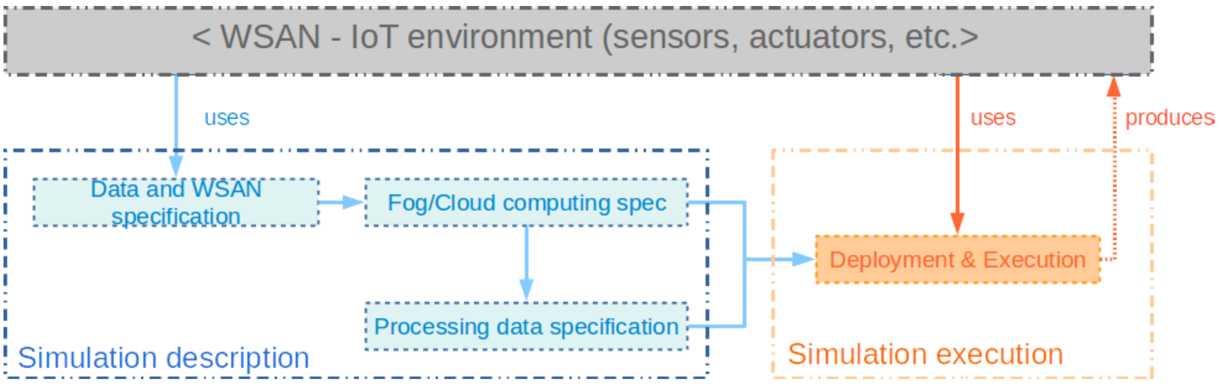

< WSAN - IoT environment (sensors, actuators, etc.>
uses
uses
produces
Data and WSAN specification
Fog/Cloud computing spec
Processing data specification
Deployment & Execution
Simulation description
Simulation execution

FIGURE 3. SimulateIoT methodology overview.

include a Domain-Specific Language (DSL) named *SimulateIoT* for defining and deploying IoT simulation environments. For this, *SimulateIoT* uses model-driven development techniques to manage the IoT simulation environment definition using models. So, the models guide the system description and the code generation. Later on, the code generated can be deployed through several hosts.

In a Model-Driven Development approach like this the software development is guided through Models (M1) which conform to a MetaModel (M2). Moreover, a Metamodel conforms to a MetaMetaModel (M3) which is reflexive. The MetaMetaModel level is represented by well-known standards and specifications such as Meta-Object Facilities (MOF), ECore in EMF and so on. A MetaModel defines the concepts and relationships in a specific domain in order to model partial reality. Then these models are used to generate totally or partially the application code by model-to-text transformations. Thus, the software code can be generated for a specific technological platform, improving the technological independence and decreasing error proneness.

Figure 4 shows a mapping among the phases defined in the *SimulateIoT methodology* and the *SimulateIoT design and implementation* which defines a model-driven development process and the *SimulateIoT deployment and execution phase*. Thus, it shows the main elements needed to build the *SimulateIoT ToolsExecution Environment* and includes: a Metamodel definition, a Graphical Concrete Syntax definition (Figure 4-1) and the model-to-text transformations (Figure 4-2) to generate the code artefacts needed to deploy, monitor and measure the IoT environment (Figure 4-3).

Thus, the *Design and Implementation* phase makes it possible to design the IoT models and generate the code which will be deployed and executed during the *Deployment and Execution* phase. Both the *Design and Implementation* phase and the *Deployment and Execution* phase together address users to design and implement the *SimulateIoT* methodology focusing on using well-known model driven software engineering practices such as metamodeling, validating, model transformations, etc. Using it improves the system development productivity and decreases the error proneness [43].

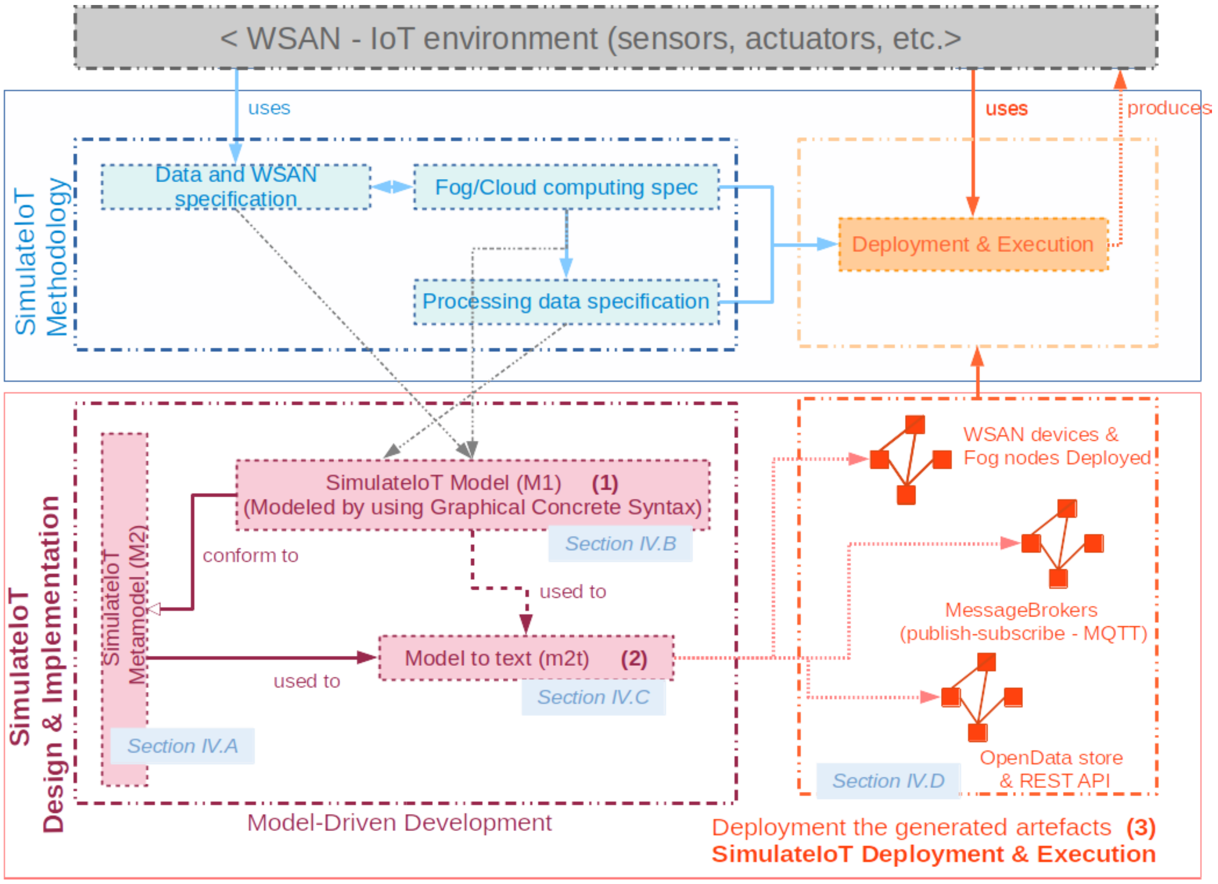
The main elements of the *Design and Implementation* phase such as the *SimulateIoT Metamodel* or *the model-to-text transformations* are described below.

A. SimulateIoT DESIGN AND IMPLEMENTATION PHASE. SIMULATE IoT METAMODEL

A MetaModel defines the concepts and relationships in a specific domain in order to model partially reality [43]. Then these models could be used to generate total or partially the application code. Thus, the software code could be generated for a specific technological platform, improving its technological independence and decreasing the error proneness.

Figure 5 defines the domain metamodel including concepts related to sensors, actuators, databases, fog and cloud nodes, data generation, communication protocols, stream processing, and deploying strategies, among others. The relevant elements are summarised below:

- The *Environment* element defines the global parameters of the IoT simulation environment, including *simulationSpeed* and the number of messages to be interchanged among the nodes (numberOfMessages). These attributes define global policies to manage simulation resources to be applied on all the *Node* elements defined.
- *Node* is an abstract concept to represent each node in the IoT simulation environment. It is extended by several concepts such as *EdgeNode* or *ProcessNode* in order to specialise each kind of node. A *Node* can publish and subscribe to a specific *Topic*. It defines *publish* or *subscribe* references towards a *Topic* element in which it is interested. Note that, later on, each concrete kind of *Node* could be defined with specific constraints. Thus, the device position (*Coordinates* element) can be defined using *latitude* and *longitude* attributes. *latitude* and *longitude* attributes define the device position (*Coordinates* element). Furthermore, with the *RaspBerryPi* attribute, the generation of the node will be carried out for this kind of device.
- The *EdgeNode* element makes it possible to define simple physical devices such as a sensor or an actuator without process capacities. Moreover, with the attribute *quantity*, it is possible to define how many *EdgeNodes* of a type must be generated. Each *EdgeNode* could be linked with *ProcessNode* elements by *Topic* elements. *Topic* elements allow link each *EdgeNode* with *ProcessNode* elements. Moreover, each *EdgeNode* can be mapped with a physical device such as a temperature sensor, a humidity sensor, a turn on/off light device or an irrigation water flow device at the IoT environment.
- A *Sensor* element extends the *EdgeNode* element. It is the device that publishes the data that the IoT environment works with.
  A *Sensor* element analyses a specific environment issue (temperature, humidity, people presence, people counter, etc.) and sends these data to be analysed later. A *Sensor* element is able to publish on *Topic* elements which propagate data throughout the simulation nodes. To perform this data propagation, *Sensors* could integrate the element *AdditionalConfiguration* that, together with the element *RedirectionConfiguration*, can define a redirection route of *ProcessNode* through which their data can flow. Thus, *Sensors* are able to publish their data in *Topics* not accessible to them.
- An *Actuator* element is a device in the IoT environment which can execute an action from a set of inputs. For instance, the inputs could determine that an actuator turn on or turn off a light; other actuators could require data input to define the light's luminosity. In order to receive data, an *Actuator* element should be subscribed to *topics*.
- *Topic* is a central element in this metamodel because it defines the information transmitted among any kind of *Node* elements. Thus, *Topic* elements are defined from *CloudNode* and *FogNode* elements, and help users to model a publish-subscribe communication model. *Node* elements should identify the target *Topic* for both publication or subscription. Consequently, the *Topic* element is a flexible concept to model how data should be interchanged.
- *Data* element defines the simple data type to be generated (Boolean, short, integer, real, string). It has a *DataSource* element to model either the *DataGeneration* element or *LoadFromFile* element. The former (*DataGeneration* element) models how synthetic data are generated, for instance, using an *Aleatory* strategy among two values defined in a *GenerationRange* element. The latter (*LoadFromFile* element) models the path-file that contains the historic data, for instance, it could be defined by a *CSVload* element. In addition, external tools such as [11], [19] can be linked to increase the capabilities to offer additional data generation patterns.
- The *ProcessNode* element defines an IoT node with process capability. For this, two subtype nodes could



< WSAN - IoT environment (sensors, actuators, etc.>
uses
uses
produces
SimulateIoT Methodology
Data and WSAN specification
Fog/Cloud computing spec
Processing data specification
Deployment & Execution
SimulateIoT Design & Implementation
SimulateIoT Metamodel (M2)
SimulateIoT Model (M1) (1)
(Modeled by using Graphical Concrete Syntax)
Section IV.B
conform to
used to
Model to text (m2t) (2)
Section IV.C
used to
Section IV.A
Model-Driven Development
WSAN devices & Fog nodes Deployed
MessageBrokers (publish-subscribe - MQTT)
OpenData store & REST API
Section IV.D
Deployment the generated artefacts (3)
SimulateIoT Deployment & Execution

FIGURE 4. SimulateIoT methodology and SimulateIoT execution design & implementation phase and deployment & execution phase related.

be defined: *CloudNode* and *FogNode*. Essentially, both have the same properties and only differ in their process capability. Thus, in order to classify the *ProcessNode* capacity (the *size* attribute) related to batteries, CPU, memories, etc. a set of granularity values have been defined (XS, S, L, XL and XXL). They make it possible to define different kinds of nodes and apply different kinds of policies. Thus, Model-Driven Development helps to deal with the complexity of IoT systems and policies management by model abstractions and constraints.

Using labels (XS, S, L, XL and XXL) to define the node capacity simplifies the knowledge needed to model the IoT environment, overall in a changing environment such as IoT. Labels are used to simplify the reality taken into account the user's knowledge and expertise. For instance, Scrum agile methodology [41] makes it possible to define the effort needed for a set of developers to develop a specific user history by using labels. Concretely they use the Planning Poker technique which uses Poker cards to estimate the effort needed to carry out a specific task summarising the developer's knowledge and expertise, task complexity, context changing, and so on. In the same sense we estimate the node capacities using the labels defined. The resources that different users can associate to a specific label can change throughout the time or taking into account their knowledge and expertise.

This strategy allows specifying the *ProcessNode* element capacity and associating specific constraints. For example, in an XS *ProcessNode* a *ProcessesEngine* such as Complex Event Processing (CEP) engine cannot be deployed. Hence, granularity labels are used as in a Scrum project development to define task complexity. As mentioned, *ProcessNode* can define *Topic* elements, which can be referenced by any kind of *Node* elements. Besides, the *redirectionTime* attribute defines the frequency that stored data are flushed towards the next *ProcessNode* element defined by *redirect* references (redirection route defined in *Sensors*). The attribute *BrokerType* defines the message-oriented broker that currently is established by *Mosquitto* [32]. In addition, the *ProcessNode* element hides the complexity about how data should be gathered and processed. For instance, it defines how data will be stored, published or offered to be analysed by stream processing engines (SP) or complex event processing engines (CEP) by defining *Component* elements. Note that either the stream

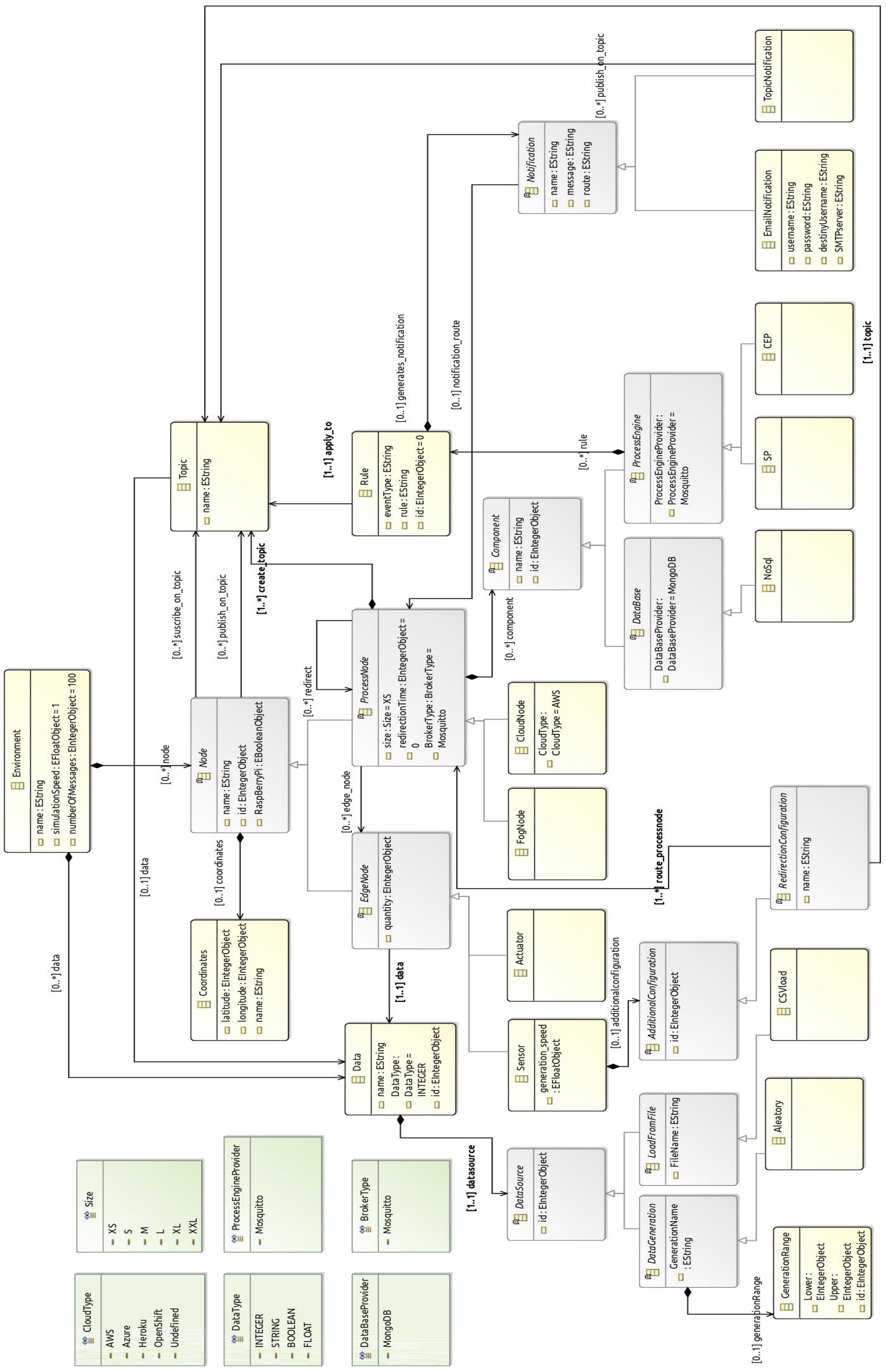


FIGURE 5. SimulateIoT metamodel.

processing or the complex event processing capabilities help to define when an *Actuator* element should carry out an action.

- *FogNode* allows users to describe fog computing instances [6] which could manage and coordinate several devices or actuators. Thus, this concept focuses on aggregating data for a limited time or connection conditions that are released later on. Furthermore, a *FogNode* element can include persistent data storing and data processing.
- *CloudNode* extends *ProcessNode* and allows describing a special node deployed on a public or private cloud computing environment.
- The *ProcessEngine* element should be linked to a *ProcessNode*, to allow real time data analysis defining coming from *ProcessNode* elements or *EdgeNode* elements. To do this, defining complex event patterns can be carried out by *Rule* elements. These patterns analyse *Topic* data in real time. Currently, the SimulateIoT environment works with WSO2 Stream Processor [37] and Esper CEP [13]. Usually, a CEP (Complex Event Processing) engine has a higher process capacity and lower latency than an ESP (Event Stream Processing) engine [25], [26].
- *Rule* elements are linked with the *ProcessEngine* elements defined at the *ProcessNode* element. *Rule* elements can be defined using the Event Processing Language (EPL) [14] defined for a concrete *ProcessEngine* kind. Note that the *eventType* attribute is used to name a rule.
- *Notification* elements make it possible to throw alerts by using several notification kinds: *TopicNotification* or *eMailNotification*. Obviously, *Notification* hierarchy could be extended in further metamodel versions. Mention that the *Notifications* are carried out by messages. Mention that messages carry out the *Notifications*. In this manner, the attribute *message* could define the notification message which will be notified.

B. SimulateIoT DESIGN AND IMPLEMENTATION PHASE. GRAPHICAL CONCRETE SYNTAX AND VALIDATOR

The Design phase includes creating models conforming to the *SimulateIoT* metamodel. So, in order to do this, a Graphical Concrete Syntax (Graphical editor) has been generated using the Eugenia tool [23]. —– So, in order to do this, the Eugenia tool [23] —– makes it possible to generate a Graphical Concrete Syntax (Graphical editor). The Graphical Concrete Syntax generated from SimulateIoT metamodel is based on Eclipse GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Tools). Consequently, models (EMF and OCL (Object Constraint Language) [34] based) can be validated against the defined metamodel (EMF and OCL based). Note that OCL is a standard to define model constraints. Figure 6 shows an excerpt from this graphical editor. It helps users to improve their productivity allowing not only defining models conforming to the *SimulateIoT metamodel*, but also their validation using OCL constraints [34]. OCL rules have been defined as part of the SimiulateIoT metamodel using OCLInEcore Tools (https://wiki.eclipse.org/OCL/OCLinEcore). Each OCL rule, defined as **invariant**, has its own context which is related to the **class** where it is established. Some of these OCL constraints are the following:

- An *EdgeNode* element can only send data to *Topic* elements defined in one *FogNode*:

```
class EdgeNode {
    \ldots
    invariant send_data_to_one_node: self.
        publish-> forall (topic1, topic2 |
        topic1.oclContainer() = topic2.
        oclContainer());
    \ldots
}
```

- Each *EdgeNode* element should be connected (to publish or to subscribe) with a *Topic*:

```
class Sensor {
    \ldots
    invariant sensor_publish: self.publish > 0
    \ldots

class Actuator {
    \ldots
    invariant actuator_subscribed:  self.
        subscribed > 0
    \ldots
```

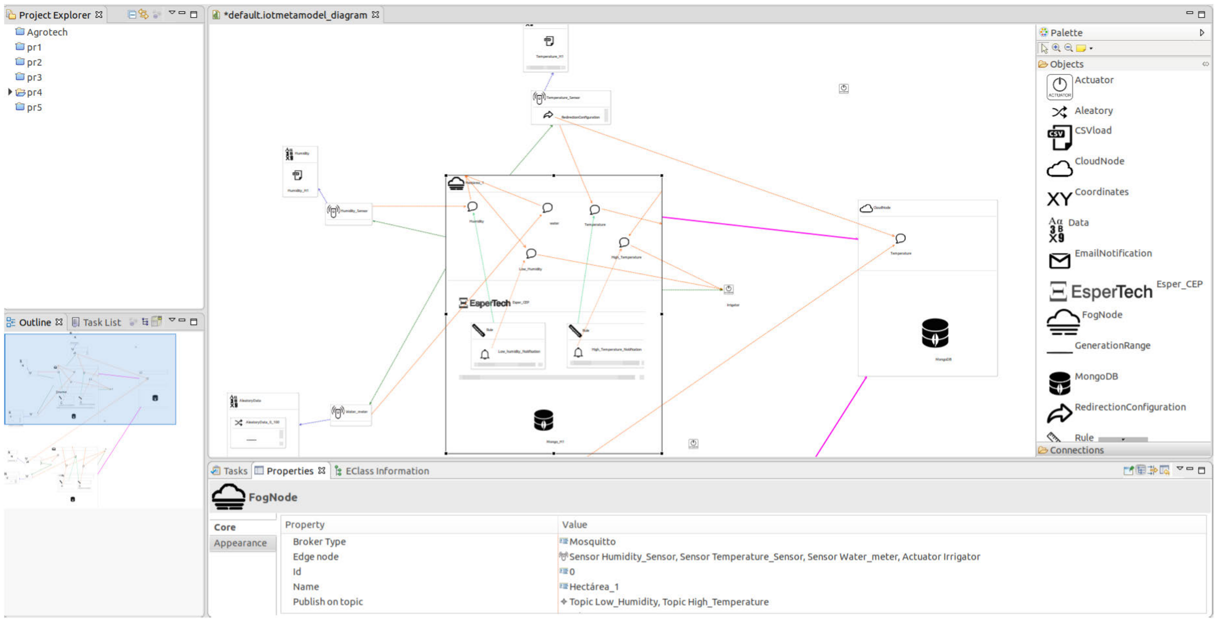
- *TopicNotification* generated by a *Rule* should be published on a *Topic* created by the *FogNode* which analyses data with this *Rule*:

```
class ProcessNode {
    \ldots
    invariant TopicNotificationPublication:
        self.create_topic->includesAll(self.
        component->selectByKind(ProcessEngine)
        .rule.generates_notification->
        selectByKind(TopicNotification).
        publish_on_topic);
    \ldots
}
```

- *ProcessNode* could be a *FogNode* or a *CloudNode*, the main difference between these two kinds of node are their computation power, a characteristic defined by the *ProcessNode* attribute *size* which should be greater than *L* in the *CloudNode* element and smaller than or equal to *L* in the *FogNode* element:

```
class CloudNode {
    \ldots
    invariant cloudSizeMajorThanL: self.size.
        toString() = 'XL' or self.size.
        toString() = 'XXL';
    \ldots
}

class FogNode {
    \ldots
    invariant fogSizeMinorThanXL: self.size.
        toString() <> 'XL' and self.size.
        toString() <> 'XXL';
    \ldots
}
```



FogNode

Property	Value
Broker Type	Mosquitto
Edge node	Sensor Humidity_Sensor, Sensor Temperature_Sensor, Sensor Water_meter, Actuator Irrigator
Id	0
Name	Hectárea_1
Publish on topic	Topic Low_Humidity, Topic High_Temperature

FIGURE 6. **Graphical editor based on the Eclipse to model conforming to the SimulateIoT metamodel.**

- The *ProcessNode* element has the ability to redirect data. To redirect data *ProcessNode* must have data persistence, be connected to another *ProcessNode* and its attribute *redirectionTime* must be greater than 0. If *redirectionTime* is equal to 0, *ProcessNode* won't redirect the data and does not have to meet these requirements.

```
class ProcessNode{
    \ldots
    invariant redirectionRequeriments: self.
        redirectionTime = $0$~or  self.
        redirectionTime > $0$~and self.
        component->selectByKind(DataBase) <>
        null and self.redirect->size() > 0;
    \ldots
    }
```

To sum up this subsection, the graphical concrete syntax (based on an Eclipse plugin) developed offers a suitable way to model the IoT environment by using the high-level concepts defined in the SimulateIoT metamodel. Later on, the graphical concrete syntax will be used to model and validate several case studies.

C. SimulateIoT DESIGN AND IMPLEMENTATION PHASE. MODEL-TO-TEXT TRANSFORMATION

Once the models have been defined and validated conforming to the *SimulateIoT metamodel*, several artefacts can be generated using a model-to-text transformation defined using Acceleo . a model-to-text transformation defined using Acceleo [38] can generate several artefacts.

The generated software includes, MQTT messaging broker (based on MQTT protocol [33]), device infrastructure, databases, a graphical analysis platform, a stream processor engine, docker container, etc. In this regard, Table 2 summarises each node type characteristic including the Docker container, NoSQL database, MQTT broker, Monitoring using graphical visualisation and analysing characteristics labelled as Complex Event Processing (CEP).

D. SimulateIoT DEPLOYMENT AND EXECUTION PHASE

The Execution phase involves deploying all the artefacts generated from the models. So, several software artefacts such as the MQTT messaging broker, device infrastructure, databases, graphical analysis platform, etc. can be configured and deployed.

Code is generated to allow users to package code, deploy and monitor the simulation. Thus, the simulation can be deployed through several hosts where each node should be deployed. Figure 7 shows an example of the IoT simulation deployed. It shows the different elements that can be deployed including a *CloudNode* or *FogNode*, *Sensors* and *Actuators*. Thus, each *CloudNode* and *FogNode* is implemented as a micro-service based on Thorntail [49] and it is deployed on a Docker container [28]. Besides, each node can be deployed on hardware with different characteristics such as Rasberry Pi, Jaguarboard, Orange Pi or Pine64. Note that these micro-computers run under several versions of Linux and Docker containers can be deployed on them.

Furthermore, each *CloudNode*/*FogNode* can define a Complex Event Processing Engine (e.g. Esper) or Event Stream Processing Engine (e.g. WSO2). Besides, it includes an MQTT broker (e.g Mosquitto), a No-SQL database (e.g. MongoDB) and a REST API. Likewise, as can be observed, all of these elements are inter-connected and are deployed on Docker containers. Finally, all Docker containers are orchestrated using Docker Swarm.

TABLE 2. Available code generation for each different kind of node.

	Docker	NoSQL DB	MQTT Broker	Monitoring	ESP analyser	CEP Analyser
CloudNode	X	X	X	X	X	X
FogNode	X	X	X	X	X	X
EdgeNode	X					

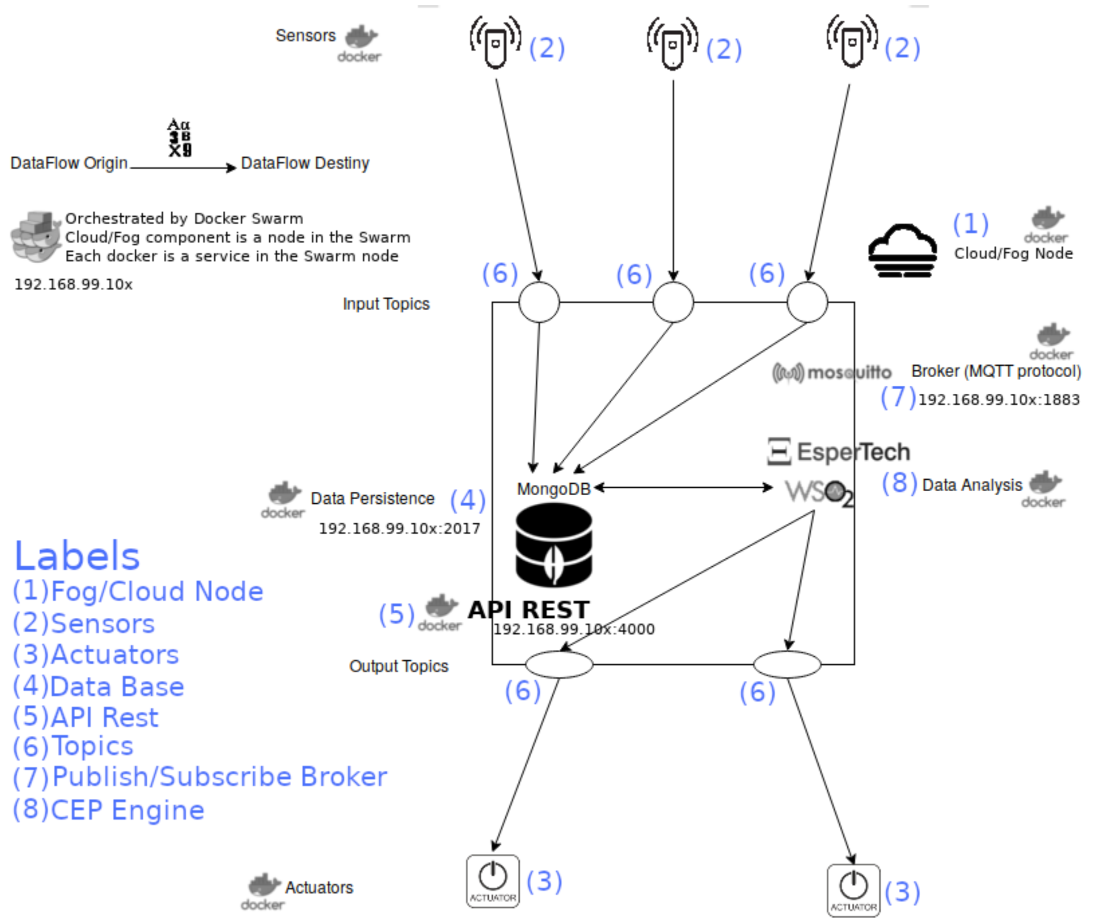


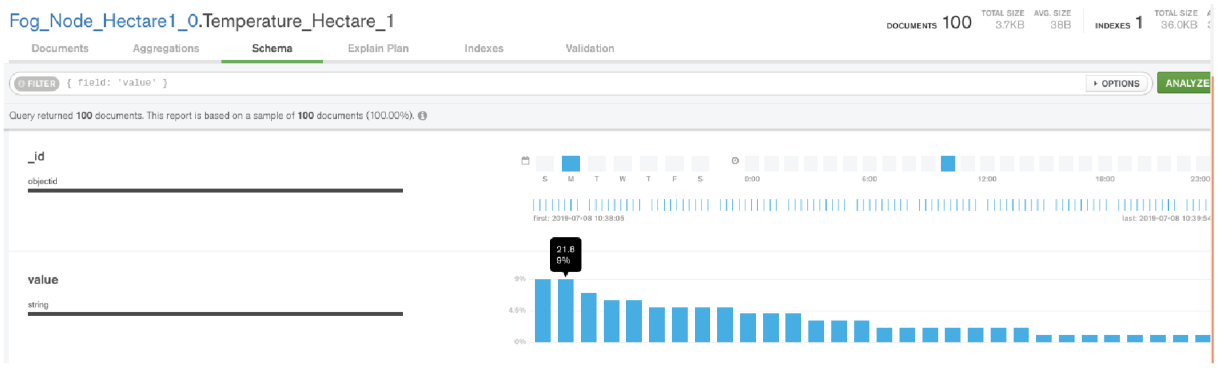
FIGURE 7. Deploy diagram.

Moreover, each node deployed with storing characteristics includes a specific monitoring tool. Figure 8 shows an excerpt from the monitoring environment based on Compass [10]. So, users take over the monitoring tool including several kinds of graphical elements such as bar graphs, data lists and so on. The monitoring environment makes it possible to query the data stored.

Finally, an overview dashboard is generated to monitor the simulation execution. So, each node defined can be queried. For instance, the data stored on a specific *ProcessNode* can be queried in real-time. For instance, the user can query the data stored on a specific *ProcessNode* in real-time. So, during simulation execution the console of each *ProcessNode* shows the simulation execution log. Later on, the simulation logs and data stored in the *ProcessNode* with storage capacity are available to be queried.

The simulation execution process including the following steps: i) compiling and deploying the artefacts previously generated from a *SimulateIoT* model; ii) data generation to commence the simulation process, consequently the defined sensors start to generate data and send them towards the defined Topics; iii) data propagation, data analysis and actions are carried out taking into account the defined data flows; and iv) log simulation can be analysed both in real-time querying the databases or after simulation execution by querying the log simulation. For instance, the following characteristics can be analysed: the performance of each component (in real-time) including *CPU* or *RAM* usage, the total memory used for each component, the amount of data sent and received for each component over its network interface, etc.

Algorithm 1 shows a simplified simulation execution process. It focuses on the actions carried out in the Docker containers deployed to execute the simulation process. Note that each Docker container has its own behaviour depending on the simulation component deployed (Sensor,



Fog_Node_Hectare1_0.Temperature_Hectare_1

DOCUMENTS 100 TOTAL SIZE 3.7KB AVG. SIZE 38B INDEXES 1 TOTAL SIZE 36.0KB

FILTER `{ field: 'value' }`

Query returned **100** documents. This report is based on a sample of **100** documents (100.00%).

first: 2019-07-08 10:38:05

last: 2019-07-08 10:39:54

FIGURE 8. MongoDB compass to monitor the data stored in the MongoDB databases.

Actuator, FogNode or CloudNode) as has been described previously.

The number of interactions in Algorithm 1 grows to $O(N * M)$ that is $O(N^2)$, where N is the number of *Node* elements and where M the number of messages to be interchanged. Note that, each *Node* is deployed on a concrete Docker container where each *Node* should execute its behaviour ($O(N)$).

The generated IoT system defines a mesh topology network where sensors, actuators, fog nodes and cloud nodes could be interconnected following the model defined. The system modeller can use the Graphical Concrete Syntax that has been developed to describe the *Node* elements interactions.

V. CASE STUDIES

Next, two case studies have been defined using the SimulateIoT methodology and tools previously presented. The first one defines an IoT simulation on a smart building while the second one defines an IoT simulation in an agricultural environment.

Below is a synthesis of the methodology required to use *SimulateIoT* and the processes carried out by this tool to simulate these use cases in order to illustrate them more effectively.

1) *Model definition:* This step refers to the modelling of the *IoT Environment* that the user wants to deploy. This model corresponds to the *DSL* and therefore can contain all the elements defined in it. Two examples of *IoT Environment* models are shown in Figure 9 and in Figure 11.
2) *Code generation and deployment:* Once the model has been defined, the source code of all the elements involved can be generated from it. *Sensors*, *Actuators*, *FogNodes*, *CloudNodes* and all their sub-components and configuration files will be ready for the deployment phase. The deployment performs many steps for the correct deployment of all previously generated components.

A. CASE 01. SCHOOL OF TECHNOLOGY

Our first case study presents the simulation of a smart building, more specifically, we have modelled the School of Technology at the University of Extremadura. It has six buildings (Computer Science, Civil Works, Architecture, Telecommunications, Research and a Common Building). So, each building has its own environment with a set of sensors, actuators and analysis information processes.

1) CASE 01. MODEL DEFINITION

Figure 9 shows an excerpt from the School of Technology model. Note that Figure 9 also includes numerical references for each node which are then used to describe the use case. It is a design of an IoT system which includes several nodes shared throughout the different buildings. Each building takes over its own *ProcessNode* (Figure 9, references 1.1, 1.2, 2) which recovers all the information produced by the sensors (Figure 9, references 3.1, 3.2). Thus, these data are suitably stored on specific databases (Figure 9, references 6.1, 6.2, 6.3), analysed and monitored in *ProcessNode* elements. In this case study, a *FogNode* element is defined for each building (Figure 9, references 1.1, 1.2). For instance, *Common_Building or Computer_Science* are *FogNode* elements (Figure 9, references 1.2, 1.1). Furthermore, a *CloudNode* named *SchoolTechnologyCloudNode* (Figure 9, reference 2) is defined to store information gathered from the *FogNode* elements. Both *FogNode* and *CloudNode* elements define several *Topic* elements such as *heating_temperature*, *presence*, *smoke-detection* topics (Figure 9, references 5.1, 5.2, 5.3). These *Topic* elements communicate data among the *Node* elements defined in the IoT system (Figure 9, references 1.1, 1.2, 2, 3.1, 3.2, 3.3).

In order to model the School of Technology case study, several sensors such as *heating_temperature_meter*, *presence_detector*, *smoke_detector* (Figure 9, reference 3.1) and so on have been defined in Figure 9. Each of them publishes its own data on a specific *Topic* element (Figure 9, reference 5.1). As can be observed in Figure 9, the *Sensor* elements publish data to several *FogNode* through *Topic* elements.

Note that *Sensor* elements are *EdgeNode* elements which generate data, so the data pattern generators should be defined (Figure 9, references 4.1, 4.2). For instance, in order to describe the synthetic data generated by a temperature sensor

Algorithm 1: Deploying and Executing the *IoT* Simulation

```
begin

    //Step 1)~Connections and configuration of each component

    Compile and deploy each IoT component by using Docker Swarm
    Subscribe each Node (Fog-CloudNode, Sensors, Actuators) to the Topics offered by MQTT Brokers
    Subscribe each ProcessNode (FogNode and CloudNode) to the Topics on other Fog-CloudNode
    Configure~the CEP/SP Engine with their EPL rules

    //Step 2)~Start the message flow, the~components start their processes

    //Start Data Generation
    foreach Sensor do
        start to publish data from it sources (.csv, syntheticDataGeneration(), etc.) to Topic
    done

    // Main process executed in parallel by each Node
    while (data in Sensors is available) do
        Nodes (FogNode, CloudNode, Actuator) subscribed to Topic receive the data

        //each Node (FogNode, CludNode or Actuator) process the data received
        switch (NodeType n)

            ProcesNode:

                //2.1  CEP/SP Analysis
                if (n has CEP/SP engine) then
                    foreach rule to apply to data do
                        ruleObserved=CEP-SP.applyRule(rule[i])
                        if ruleObserved == True then
                            CEP-SEP.sendNotification(rule[i].notificationDestiny)
                        endif
                    endforeach
                endif

                // 2.2. Data Store
                if (n has Persistence) then
                     n.saveData(MongoDB)
                endif

                // 2.3. Data redirection
                if (n has redirection data) then
                    redirectionData = n.checkredirectionableData(MongoDB)
                    foreach redirectionData do
                                n.redirectData(redirectionData.Destiny)
                    endforeach
                endif

            Actuator:
                n.doSomeAction(data)
        endswitch
    done
end
```

a.csv input file has been defined. It makes it possible to reuse historical data. Other sensors can define their synthetic data generators using a random pattern, incremental pattern, etc. So, the approach can consume synthetic data based on simple data, range data, a specific set of values, the values obtained from a.csv file, data obtained from a url source or data generated form the external tools such as [11], [19].

As mentioned, in Figure 9 each *FogNode* has its own characteristics about how data should be managed including storing, analysing or addressing. For instance, the *ComputerScience FogNode* element (Figure 9, reference 1.1) addresses the information every *thirty seconds*, storing the data obtained in a specific NoSQL database (Figure 9, reference 6.1). Then all data are flushed to the next node *FogNode or CloudNode* defined in the architecture and named in the example SchoolofTechnology_CloudNode. On the other hand, the *Common_Building FogNode* element (Figure 9, reference 1.2) defines a different behaviour in order to analyse the data and take advantage of being close to the devices that should carry out some action. For instance, the *Common_Building FogNode* defines a *CEP engine* component (Esper_CEP) and several *Rule* elements

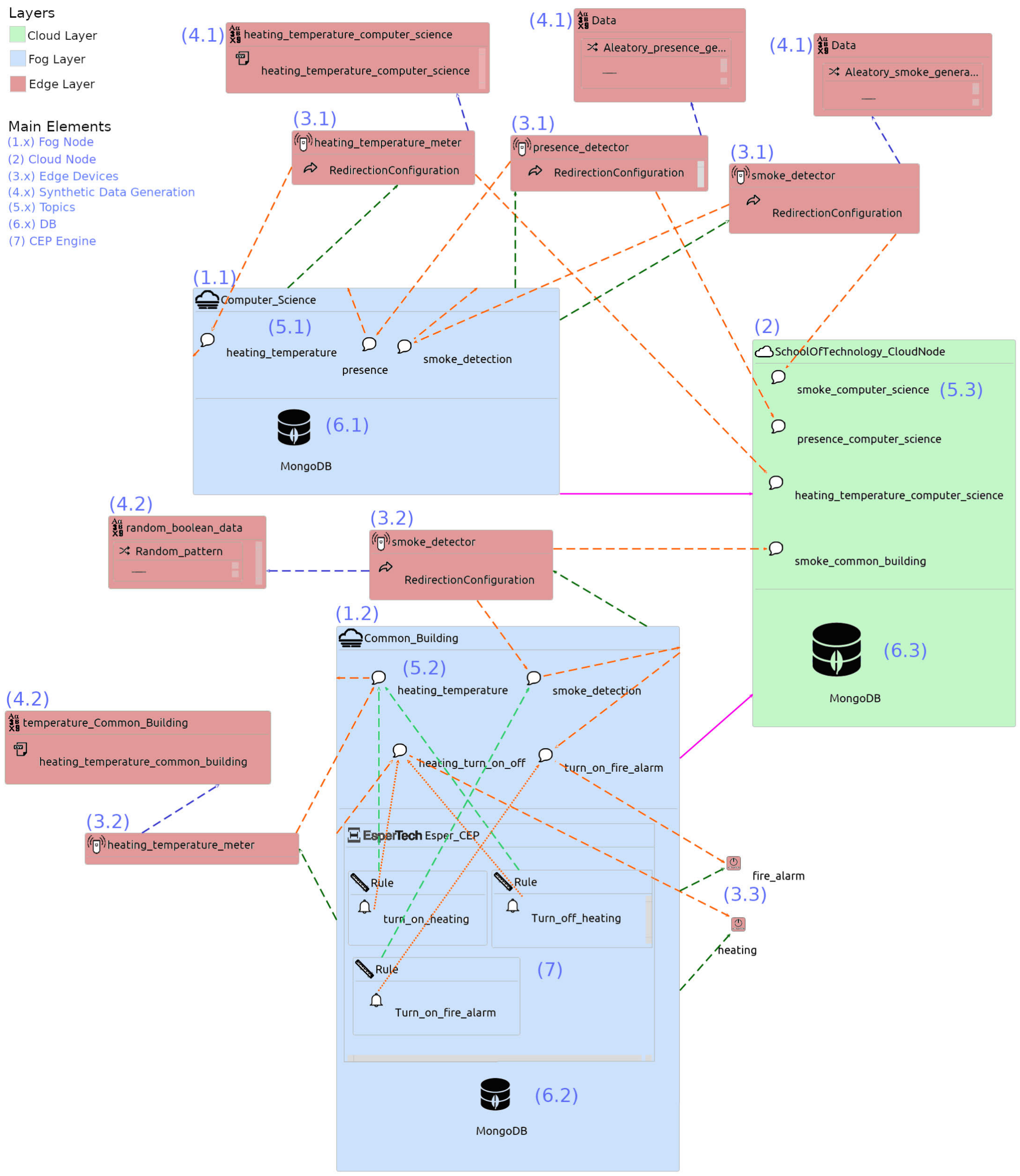


FIGURE 9. Case 01. The school of technology model conforms to the SimulateIoT metamodel.

(Figure 9, reference 7), for example, the *rule_heating* analyses the data obtained from a specific *Topic* named *heating_temperature* to notify a specific action to another *Topic* named *turn_on_heating* which is subscribed by specific *Actuator* named *heating*. Thus, the *rule_heating* rule analyses the temperature sent to the *heating_temperature* Topic element from the *heating_temperature_meter* Sensor. Consequently, it is gathered and analysed by the *CEP engine* by means of the *rule_heating* Rule. Consequently, the *CEP engine* can gather data and analyse them by means of the *rule_heating*

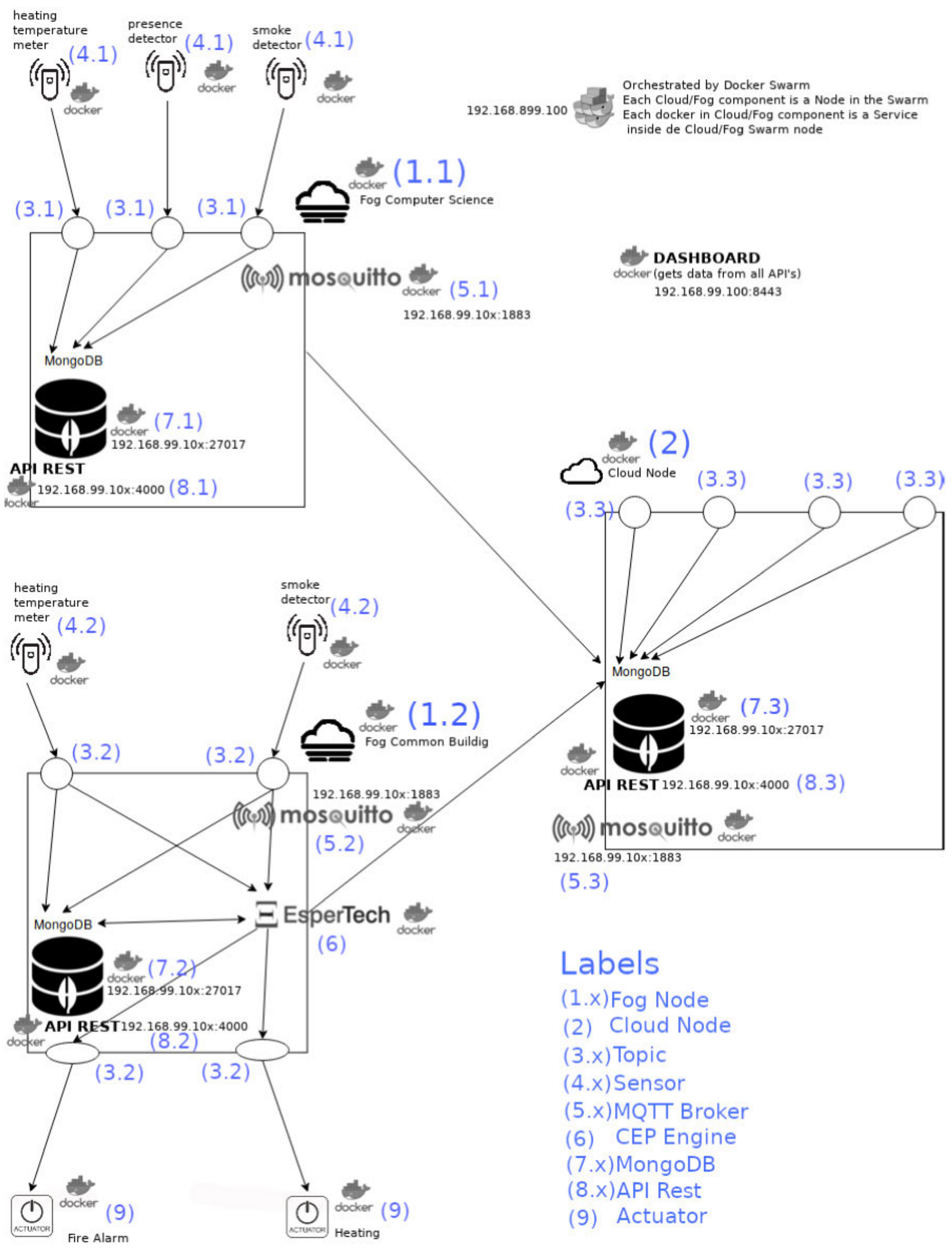

heating temperature meter (4.1)
presence detector (4.1)
smoke detector (4.1)
Orchestrated by Docker Swarm
Each Cloud/Fog component is a Node in the Swarm
Each docker in Cloud/Fog component is a Service inside de Cloud/Fog Swarm node
192.168.899.100
(1.1) Fog Computer Science
(3.1)
mosquitto (5.1)
192.168.99.10x:1883
DASHBOARD (gets data from all API's)
192.168.99.100:8443
MongoDB (7.1) 192.168.99.10x:27017
API REST 192.168.99.10x:4000 (8.1)
(2) Cloud Node
(3.3)
MongoDB (7.3) 192.168.99.10x:27017
API REST 192.168.99.10x:4000 (8.3)
mosquitto 192.168.99.10x:1883 (5.3)
heating temperature meter (4.2)
smoke detector (4.2)
(1.2) Fog Common Buildig
(3.2)
192.168.99.10x:1883
mosquitto (5.2)
EsperTech (6)
MongoDB (7.2) 192.168.99.10x:27017
API REST 192.168.99.10x:4000 (8.2)
Fire Alarm (9)
Heating (9)
Labels
(1.x) Fog Node
(2) Cloud Node
(3.x) Topic
(4.x) Sensor
(5.x) MQTT Broker
(6) CEP Engine
(7.x) MongoDB
(8.x) API Rest
(9) Actuator

FIGURE 10. Case 01. The school of technology model deployed.

Rule. As a consequence, when the pattern defined is matched (for instance, *if (temperature < 20) then switch on heating*), the CEP engine generates an event to *turn_on_off_heating* Topic. As a consequence, the CEP engine generates an event to *turn_on_off_heating Topic* when the pattern defined is matched (for instance, *if (temperature < 20) then switch on heating*).

2) CASE 01. CODE GENERATION AND DEPLOYMENT

Once the model has been defined, the model-to-text transformation is applied with the following goals: i) to generate Java code which wraps each device behaviour; ii) to generate configuration code to deploy the message brokers necessary, including the *topic* configurations defined; iii) to generate the configuration files and scripts necessary to deploy the databases and stream processors defined; and finally, to generate the code necessary to query the databases where the data will be stored; iv) to generate for each *ProcessNode* and *EdgeNode* a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm*.

Figure 10 shows an excerpt from the School of Technology IoT model deployed and it includes the following: Each *Node* has been deployed on a *Docker* container using *Docker Swarm* technology. Each *Docker container instance* deploys the characteristics defined on the IoT model, including: where the nodes are deployed, and what the components included in each *ProcessNode* are.

Finally, executing the simulation modelled and later on deploying it, makes it possible to analyse the final IoT environment before it is implemented and deployed. Thus, each *EdgeNode* and *ProcessNode* element carries out its own functions such as sending messages, processing and storing messages, acting from messages, etc. Consequently, the code generated can be reused on the final system deployed. For instance, the *EdgeNode* elements can be replaced by physical devices (both sensors and actuators), and the *Process Node* can be deployed as *Docker* containers either on premise or on cloud. Not only is the simulation code generated, but also the final IoT system code is partially generated.

B. CASE 02. AGRICULTURAL ENVIRONMENT

This case study focuses on designing an IoT system for managing irrigation and weather data in order to improve crop production. So, the case study has been designed to simulate the sensors and actuators distributed over the countryside which can be monitored in real time. Nowadays, the agricultural domain has several requirements [50], [52]: i) Collection of weather, crop and soil information; ii) Monitoring of distributed land; iii) Multiple crops on a single piece of land; iv) Different fertilizer and water requirements for different pieces of uneven land; v) Diverse requirements of crops for different weather and soil conditions; vi) Proactive solutions rather that reactive solutions.

For instance, sensors such as temperature sensors, humidity sensors, irrigation sensors, *PH* sensors and actuators such as irrigation artefacts help to monitor and save water, optimising crop production.

This agricultural IoT environment has been designed over ten hectares of soil where tomatoes are being cultivated. So, for each hectare a set of sensors and fog nodes has been shared. So, using fog nodes decreases the communication requirements among them. The sensor network is built by temperature, humidity, irrigation and water pressure sensors. These sensors send data to a specific *Topic* element linked to a *FogNode* element which is gathering data and re-sending it, if it is needed. In addition, the irrigation actuators have been defined for controlling irrigation water. The notification events from the *FogNode* elements are sent to *Actuator* elements using *Topic* elements.

1) CASE 02. MODEL DEFINITION

In Figure 11 an excerpt from an IoT model conforming to the SimulateIoT metamodel is defined. It shows different *Sensor* elements such as *ph_H1, temperature_H1, Humidity_H1, etc.* (Figure 11, reference 3.2) which generates data for simulation. Moreover, several fog computing nodes have been defined, although in Figure 11 (for the sake of simplicity) only two *FogNode* elements are shown (Figure 11, references 1.1, 1.2). They define several *Topics* such as *Humidity, Temperature, pH, Water_pressure, etc* (Figure 11, references 5.1, 5.2). In addition, each *FogNode* element defines a MongoDB database (Figure 11, references 6.1, 6.2) and an ESP engine (Figure 11, references 7.1, 7.2) by means of *Component* elements. Besides, several *Rule* elements (event pattern definitions) such as *rule_Humidity* or *rule_pH* have been defined to analyse the data gathered from *Topic* elements in real-time. Likewise, when an event pattern is matched, a *Notification* element such as *Low_pH, High_pH, Low_Humidity, High_Humidity* and so on is thrown. For instance, the *Actuator* element named *Irrigator* (Figure 11, references 3.1) activates when the *Notification* element named *Low_Humidity* is thrown.

2) CASE 02. CODE GENERATION AND DEPLOYMENT

Once the model has been completed and validated, a model-to-text transformation is carried out obtaining the simulation code, which can be deployed on a specific platform. Thus, the code generated includes several modules defined using several frameworks or programming languages. Thus, in order to define a scalable IoT environment, each deployable element (*EdgeNode*, *CloudNode*, *FogNode*, *Actuators* and *ProcessEngine*) is defined as a microservice, wrapping each *Node* element in a *Docker* container. Figure 12 shows an excerpt from the case study deployment architecture including the Docker containers defined and deployed. In Figure 12 the main characteristics of each node can be observed. For instance, each *ProcessNode* (Figure 12, references 1.1, 1.2, 2) defines a MongoDB database (Figure 12, references 8.1, 8.2, 8.3), a Mosquitto MQTT message broker (Figure 12, references 5.1, 5.2, 5.3), and a WSO2 Stream Processor engine (Figure 12, references 6.1, 6.2). In addition, the *Rule* elements defined are processed through the WSO2 engine defined.

Each *Docker* container has its own characteristics:

- *CloudNode* (Figure 12, reference 2) is composed of a message-driven broker (Figure 12, reference 5.3) like Mosquitto [32] (that implements a MQTT communication protocol) and a NoSQL database like MongoDB [31] (Figure 12, reference 7.3). Besides, the MongoDB instance exposes the data stored using a REST API (Figure 12, reference 8.3). Moreover, the *CloudNode* deploys a Compass instance [10] to monitor the data gathered.
- Each *FogNode* (Figure 12, references 1.1, 1.2) is composed of a message-driven broker (Figure 12, references 5.1, 5.2) like Mosquitto [32] (that implements a MQTT communication protocol) and a NoSQL database like MongoDB [31] (Figure 12, references 7.1, 7.2). MongoDB stores the temporal data gathered by the *FogNode* instance. Currently, the main difference between a *CloudNode* and a *FogNode* is the process capability. Using the *size* attribute at *FogNode* element makes it possible to define the process capabilities of the node. Consequently, both *CloudNode* elements and *FogNode* elements are deployed as Docker containers on hardware nodes such as PC, VM or Raspberry Pi.

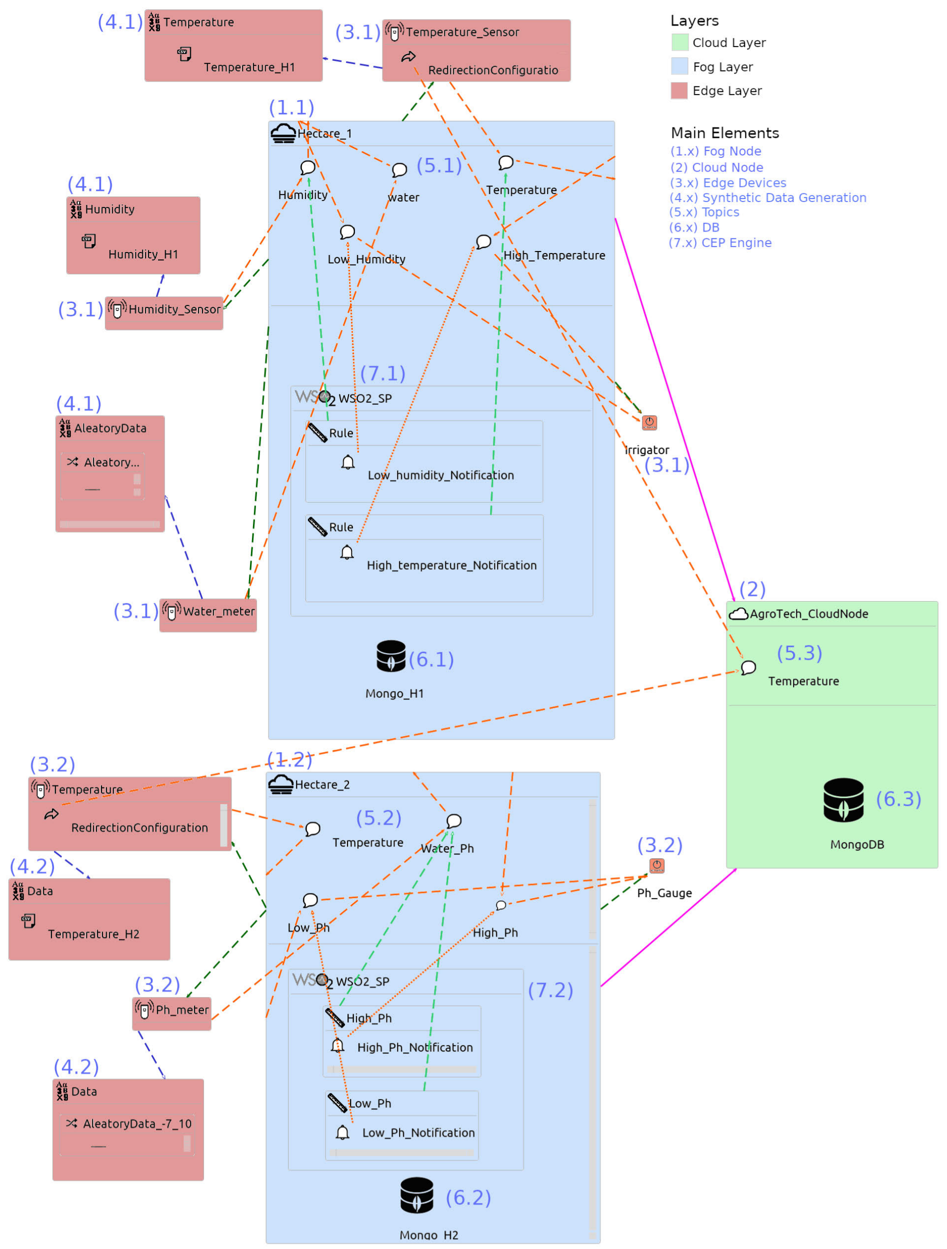


FIGURE 11. Case 02. AgroTech model conforming to the SimulateIoT metamodel.

- The *ESP* characteristic defined at *ProcessNode* deploys an event stream processor to process high amounts of messages in real-time. As can be observed in Figure 12 a WSO2 engine (Figure 12, references 6.1, 6.2 is deployed on each *FogNode*. The WSO2 engine processes the *Rule* elements associated with it.
- The *EdgeNode* elements including sensors (Figure 12, references 4.1, 4.2) and actuators (Figure 12, references 9.1, 9.2) defined in the model are suitably deployed in Docker containers.

Later on, the execution information can be audited by querying the MongoDB database or using the monitoring tool available on each *ProcessNode*. Moreover, each Docker is generating log information during the IoT execution. Finally, the nodes deployed are accessible from a dashboard tool which gathers the available endpoints of each element, for example, to query a MongoDB database or to show information about a Mosquitto broker.

VI. DISCUSSION

Model-driven development can be used to model complex IoT environments using domain concepts. They could not be tied to specific technology, but rather a model-to-text transformation makes it possible to generate the code needed to deploy and simulate the systems. Besides, the system deployed is gathering continuous data which can be analysed later on.

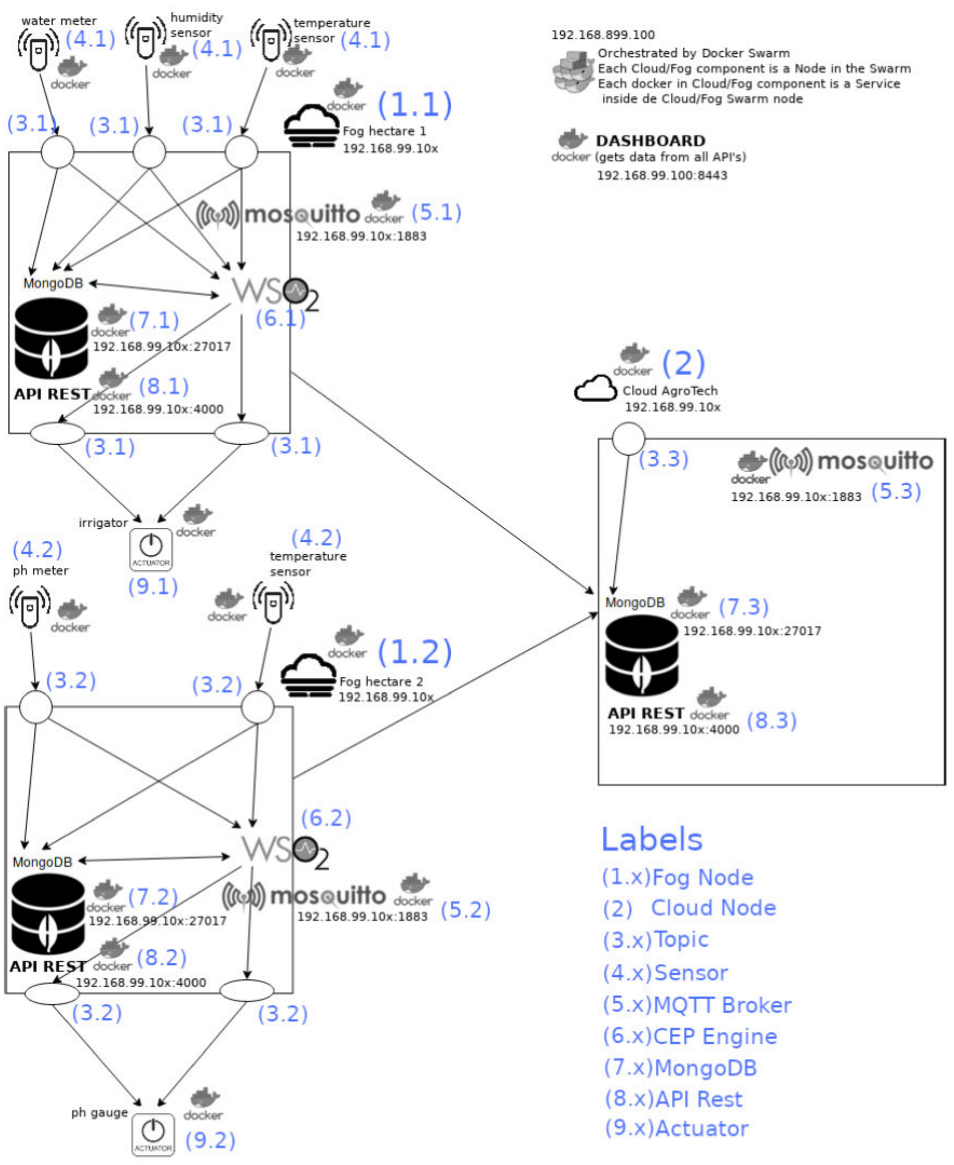


FIGURE 12. Case 02. AgroTech deployment architecture.

Simulate IoT makes it possible to define models which could include a large amount of Node elements. Then, the code generated from models allows to create an scalable deployment based on well-known software architecture patterns such as publish-subscribe and Docker containers among others.

The technology used as a target, such as micro-services (Thorntail), containers (Dockers), message-oriented middleware and MQTT (Mosquitto broker) or a container orchestrator (Docker Swarm) can be quickly replaced by other suitable technology if needed. In order to change the target technology, a model-to-text transformation should be implemented. However, the domain concepts used to model the IoT environment are fixed. As a consequence, the models help users to understand the IoT system, their relationships and constraints. Besides, the code generated can be analysed later on.

On the other hand, the target users could be both: a) professional users and b) students. Professional users can use the methodology and tools presented in this work to define and analyse complex IoT environments where finally heterogeneous technology is used. Besides, our approach can be used for teaching purposes because it makes it possible for students to learn about IoT concepts and relationships. In addition, they can deploy the IoT simulation, and they can study the code generated to learn the technology used to deploy the IoT system. Thus, they can understand edge technology and integration patterns such as data patterns, IoT characteristics, publish-subscribe communication protocols, MQTT (Message Queuing Telemetry Transport) communication protocol, containers, NoSQL databases, distributed systems and so on.

An IoT environment where the nodes are moving throughout the system can be partially modelled. These kinds of nodes are needed to define more complex IoT simulation environments such as wearables, people on the move, etc. Modeling complex node behaviours could be managed by means of dynamic behaviours and self-adaptation characteristics which could be defined in order to offer additional mechanisms for simulation purposes. For example, currently we are working on *Topics* elements which could be discovered by using a service discovery or using an introspection mechanism over the MQTT broker. The node service discovery is a new service deployed on Fog and Cloud Node elements able to offer by an API information about the Topics available, making possible that the IoT Nodes can connect to, send to and receive data from not fixed IoT nodes.

The proposal that we are implementing to manage Node mobility includes the following aspects:

- It is possible to model a *route generation*, taking advantage of the geolocation that is already modellable. In this way, *Node* elements that require mobility to perform their functions can be moved through the IoT environment in such a way that the user who has modelled the environment requires it.
- The route generation solves the problem that arises from the need for mobility of devices in an environment. However, it is also necessary to define the coverage of the different Brokers in the environment, so that the different devices are able to make the decision to disconnect from one broker and connect to another. To solve this problem, it is proposed again the use of geolocation. In this way, the user who models the environment can define a radius of coverage of the different Brokers deployed, so that the devices, taking into account their own geolocation, can determine which Brokers are within reach and which are not. Thus, the different mobile devices in the environment can analyse which Brokers to connect to and which to disconnect from.
- The *Topic Discovery Mechanism* is a service that makes it possible to dynamically re-configure the *Node* elements in order to publish or subscribe on compatible Topics. To do this, *Node* elements publish a broadcast package to the network following Topic Services available and compatible with a concrete Topic. To answer the broadcast, each *Node Processing* element implements a *Topic Discovery Node* which answers it with the list of Topics available and compatible. Currently, the Topic compatibility is based on the Topic Data interchanged, Topic's name or Topic's Tags.

Initial results of this approach to manage node mobility show that IoT nodes can dynamically reconfigure their connections to send or receive data.

Finally, using the IoT simulation environment, users can propose and compare several policies before implementing them. Consequently, they can carry out several stress tests on the IoT architecture, obtaining valuable data. For instance, users can detect if a *ProcessNode* is running out of RAM. In addition, the bottlenecks in the IoT system could be detected by analysing the data gathered, producing valuable data that helps users to consider different IoT architectural alternatives.

A. LIMITATIONS

Although the domain-specific language and tools presented offer a wide expressiveness, they have several limitations to take into account:

- *Node mobility* has been partially developed following the approach that has been described before by defining the Topic Discovery Node (TDN). In this sense, on one hand, the route for nodes can be defined, and, on the other hand, the TDN makes it possible re-configuring dynamically the WSN deployed.
- This current version of our simulator IoT environment, for the sake of simplicity, allows defining connected nodes by TCP/IP, and we assume that connectivity is guaranteed.
- It is possible to simulate IoT environments defined using a high-level domain-specific language. However, the hardware simulation is only managed by the *size* attribute at *ProcessNode* which implies several constraints to avoid creating specific software elements (see Table 2). Obviously, it could be considered a simplistic approach to tackle this complex problem but in the end, it helps users to model the IoT environments thinking about the hardware restrictions.

VII. CONCLUSION

Model-driven development techniques are a suitable way to tackle the complexity of domains where heterogeneous technologies are integrated. Initially, they focus on modelling the domain by using the well-known four-layer metamodel architecture. Then, by using model-to-text transformations the code for specific technology could be generated. Thus, in this paper, we are tackling the IoT simulation domain allowing users to define and validate models conforming to the *SimulateIoT* metamodel. Then, a model-to-text transformation generates code to deploy the IoT simulation model defined.

The IoT simulation methodology and tools proposed in this work help users to think about the IoT system, to propose several IoT alternatives and policies in order to achieve a suitable IoT architecture. Finally, the IoT systems modelled can be deployed and analysed.

Future works include new concepts taking into account the role of connections among devices and brokers which could be simulated specifying the type of connection or distance among devices. Obviously, the *SimulateIoT* metamodel will be improved by applying these new concepts, although it will require that users define more accurately the IoT simulation model. Additionally, dynamic behaviours and self-adaptation characteristics could be defined in order to offer additional mechanisms for simulation purposes. For example, *Topics* elements could be discovered by using a service discovery or using introspection mechanism over the MQTT broker. Finally, another interesting further work includes the definition and generation of new types of data generation patterns. Again, these model extensions will improve the IoT simulation.

REFERENCES

[1] K. Alwasel, R. N. Calheiros, S. Garg, R. Buyya, M. Pathan, D. Georgakopoulos, and R. Ranjan, "Bigdatasdnsim: A simulator for analyzing big data applications in software-defined cloud data centers," *Softw. Pract. Exper.*, vol. 51, no. 5, pp. 893–920, 2020. [Online]. Available: https://onlinelibrary.wiley.com/action/showCitFormats?doi=10.1002%2Fspe.2917

[2] C. Atkinson and T. Kuhne, "Model-driven development: A metamodeling foundation," *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, Sep. 2003.

[3] P. Baldwin, S. Kohli, A. Edward Lee, X. Liu, and Y. Zhao, "Modeling of sensor nets in ptolemy II," in *Proc. 3rd Int. Symp. Inf. Process. Sensor Netw. (IPSN)*, New York, NY, USA, 2004, pp. 359–368.

[4] Bevywise. (2018). *Bevywise IoT Simulator*. [Online]. Available: https://www.bevywise.com/iot-simulator/

[5] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2008-37, 2008.

[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[7] E. Cheong, E. A. Lee, and Y. Zhao, "Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks," in *Proc. SenSys*, vol. 5, 2005, p. 302.

[8] J. P. Clemente, M. J. Conejero, J. Hernández, and L. Sánchez, "Haais-DSL: DSL to develop home automation and ambient intelligence systems," in *Proc. 2nd Workshop Isolation Integr. Embedded Syst. (IIES)*, New York, NY, USA, 2009, pp. 13–18.

[9] P. Clemente and A. Lozano-Tello, "Model driven development applied to complex event processing for near real-time open data," *Sensors*, vol. 18, no. 12, p. 4125, Nov. 2018.

[10] (2018). *MongoDB Compass*. [Online]. Available: https://www.mongodb.com/products/compass

[11] A. G. D. Prado, G. Ortiz, J. Hernández, and E. Moguel, "Generación de datos sintéticos para arquitecturas de procesamiento de datos del Internet de las cosas," *Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*, 2018. [Online]. Available: http://hdl.handle.net/11705/jcis/2018/007

[12] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity–the ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[13] EsperTech. (Nov. 2016). *Esper CEP*. [Online]. Available: http://www.espertech.com/esper/

[14] EsperTech. (Jul. 2019). *Esper EPL Language*. [Online]. Available: http://esper.espertech.com/release-5.2.0/esper-reference/html/epl_clauses.html

[15] C. M. de Farias, I. C. Brito, L. Pirmez, F. C. Delicato, P. F. Pires, T. C. Rodrigues, I. L. dos Santos, L. F. R. C. Carmo, and T. Batista, "COMFIT: A development environment for the Internet of Things," *Future Gener. Comput. Syst.*, vol. 75, pp. 128–144, Oct. 2017.

[16] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Proc. Future Softw. Eng. (FOSE)*, May 2007, pp. 37–54.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, May 2003.

[18] *MDA Guide Revision*, Object Management Group, Needham, MA, USA, 2014.

[19] L. Gutiérrez-Madroñal, I. Medina-Bulo, and J. J. Domínguez-Jiménez, "IoT–TEG: Test event generator system," *J. Syst. Softw.*, vol. 137, pp. 784–803, Mar. 2018.

[20] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Syst. J.*, vol. 45, no. 3, pp. 451–461, 2006.

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 93–104, Dec. 2000.

[22] D. N. Jha, K. Alwasel, A. Alshoshan, X. Huang, R. K. Naha, S. K. Battula, S. Garg, D. Puthal, P. James, A. Zomaya, S. Dustdar, and R. Ranjan, "IoTSim-edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments," *Softw. Pract. Exper.*, vol. 50, no. 6, pp. 844–867, 2020.

[23] D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige, "Eugenia: Towards disciplined and automated development of GMF-based graphical model editors," *Softw. Syst. Model.*, vol. 16, no. 1, pp. 229–255, 2015.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: Accurate and scalable simulation of entire tinyos applications," in *Proc. 1st Int. Conf. Embedded networked sensor Syst.*, pp. 126–137. ACM, 2003.

[25] D. Luckham. (2006). *What's the Difference Between ESP and CEP?*. [Online]. Available: http://www.complexevents.com/2006/08/01/what%e2%80%99s-the-difference-between-esp-and-cep/

[26] A. Mathew, "Benchmarking of complex event processing engine-esper," Dept. Comput. Sci. Eng., Indian Inst. Technol. Bombay, Maharashtra, India, Tech. Rep. IITB/CSE/2014/April/61, 2014.

[27] K. Mehdi, M. Lounis, A. Bounceur, and T. Kechadi, "CupCarbon: A multiagent and discrete event wireless sensor network design and simulation tool," in *Proc. 7th Int. Conf. Simul. Tools Techn.*, Lisbon, Portugal, 2014, pp. 126–131.

[28] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[29] *Meta Object Facility (MOF) Core Specification Version 2.5.1*, Meta Object Facility (MOF), Milford, MA, USA, Nov. 2016.

[30] N. Mohan and J. Kangasharju, "Edge-fog cloud: A distributed cloud for Internet of Things computations," in *Proc. Cloudification Internet Things (CIoT)*, 2016, pp. 1–6.

[31] MongoDB. (2018). *Mongodb is a Document Database*. [Online]. Available: https://www.mongodb.com/

[32] Mosquitto. (2018). *Eclipse Mosquitto: An Open Source MQTT Broker*. [Online]. Available: https://mosquitto.org/

[33] *Message Queuing Telemetry Transport (MQTT) v5.0 Oasis Standard*, Oasis, Woburn, MA, USA, 2019.

[34] *OMG Object Constraint Language (OCL), Version 2.3.1*, OMG, Milford, MA, USA, Jan. 2012. [Online]. Available: https://www.omg.org/contact.htm

[35] G. Z. Papadopoulos, J. Beaudaux, A. Gallais, T. Noel, and G. Schreiner, "Adding value to WSN simulation using the IoT-LAB experimental platform," in *Proc. IEEE 9th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2013, pp. 485–490.

[36] P. Patel and D. Cassou, "Enabling high-level application development for the Internet of Things," *J. Syst. Softw.*, vol. 103, pp. 62–84, May 2015.

[37] (2018). *WSO2 Stream Processor*. [Online]. Available: https://wso2.com/analytics-and-stream-processing/

[38] (2016). *Acceleo Project*. [Online]. Available: http://www.acceleo.org

[39] A. Ruppen, J. Pasquier, S. Meyer, and A. Rüedlinger, "A component based approach for the Web of things," in *Proc. 6th Int. Workshop Web Things (WoT)*, 2015, pp. 1–6.

[40] D. C. Schmidt, "Model-driven engineering," *IEEE Computer Society*, vol. 39, no. 2, p. 25, Feb. 2006.

[41] K. Schwaber and M. Beedle, *Agile Software Development With Scrum*, vol. 1. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.

[42] A. Sehgal, "Using the Contiki Cooja simulator," Center Adv. Syst. Eng., Comput. Sci., Jacobs Univ. Bremen Campus Ring, Bremen, Germany, Tech. Rep., 2013. [Online]. Available: https://www.researchgate.net/profile/Anuj-Sehgal-4

[43] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003.

[44] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, Sep. 2003.

[45] Siafu. (2007). *An Open Source Context Simulator*. [Online]. Available: http://siafusimulator.org/

[46] E. Siow, T. Tiropanis, and W. Hall, "Analytics for the Internet of Things: A survey," *ACM Comput. Surv.* , vol. 51, no. 4, p. 74, 2018.

[47] D. Soukaras, P. Patel, H. Song, and S. Chaudhary, "Iotsuite: A toolsuite for prototyping Internet of Things applications," in *Proc. 4th Int. Workshop Comput. Netw. Internet Things (ComNet-IoT), 16th Int. Conf. Distrib. Comput. Netw. (ICDCN)*, 2015, p. 6.

[48] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2009.

[49] Thorntail. (2018). *Microprofile for Optimizing Enterprise Java Applications*. [Online]. Available: https://thorntail.io/

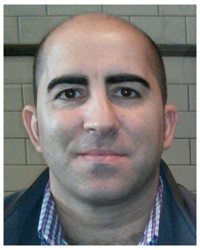
[50] Aqeel-ur-Rehman, A. Z. Abbasi, N. Islam, and Z. A. Shaikh, "A review of wireless sensors and networks' applications in agriculture," *Comput. Standards Interfaces*, vol. 36, no. 2, pp. 263–270, Feb. 2014.

[51] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. 1st Int. Conf. Simulation Tools Techn. Commun., Netw. Syst. Workshops*, 2008, p. 60.

[52] N. Wang, N. Zhang, and M. Wang, "Wireless sensors in agriculture and food industry-recent development and future perspective," *Comput. Electron. Agricult.*, vol. 50, no. 1, pp. 1–14, 2006.

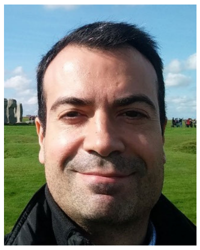
[53] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *J. Syst. Archit.*, vol. 72, pp. 93–107, Jan. 2017.

JOSÉ A. BARRIGA received the degree in computer science from the University of Extremadura, in 2017. He is currently working as a Junior Researcher with the University of Extremadura. He has been working in the IoT and the simulation IoT environments research areas since two years.



PEDRO J. CLEMENTE received the B.Sc. degree in computer science from the University of Extremadura, Spain, in 1998, and the Ph.D. degree in computer science, in 2007. He is currently an Associate Professor with the Computer Science Department, University of Extremadura. He has published numerous peer-reviewed articles in international journals, workshops, and conferences. He is involved in several research projects. His research interests include component-based software development, aspect orientation, service-oriented architectures, business process modeling, and model-driven development. He has participated in many workshops and conferences as a speaker and a member of the program committees.

ENCARNA SOSA-SÁNCHEZ received the B.Sc. degree in computer science from the University of Granada, in 1995. She is currently pursuing the Ph.D. degree with the Computer Science Department, University of Extremadura, Spain. She is also an Assistant Professor with the Computer Science Department, University of Extremadura. She has published several peer-reviewed articles in international journals, workshops, and conferences, and is involved in several research projects. Her research interests include service-oriented architectures, business process modeling, and model-driven development.



ÁLVARO E. PRIETO received the B.Sc. and Ph.D. degrees in computer science from the University of Extremadura, Spain, in 2000 and 2013, respectively. He is currently an Assistant Professor with the University of Extremadura. He is also a member of the Quercus Software Engineering Group. He is involved in various research, development, and innovation projects. His research interests include ontologies, linked open data, data engineering, and predictive analytics.