UNIVERSIDAD DE EXTREMADURA

Tesis Doctoral

Lenguaje Específico del Dominio para el Diseño y Ejecución de Simulaciones de Entornos IoT

José Ángel Barriga Corchero

Programa de Doctorado en Tecnologías Informáticas (TIN)

Con la conformidad del director
Dr. Pedro José Clemente Martín

Esta tesis cuenta con la autorización del director de la misma y de la Comisión Académica del programa. Dichas autorizaciones constan en el Servicio de la Escuela Internacional de Doctorado de la Universidad de Extremadura.

2024

UNIVERSIDAD DE EXTREMADURA

Tesis Doctoral

Lenguaje Específico del Dominio para el Diseño y Ejecución de Simulaciones de Entornos IoT

José Ángel Barriga Corchero

Programa de Doctorado en Tecnologías Informáticas (TIN)

2024

A Victoria, mi mujer, por enseñarme a subir a las estrellas y tirarme de cabeza.

A Pepi y a Teofilo, mis padres, por darme la vida y su amor incondicional.

A Arturo, mi hermano, por inspirarme y dar sentido a lo que hago.

A Ángela, mi abuela, por enseñarme qué es lo importante en esta vida.

A Felisa, mi tía, y a Carlota, mi prima, por dar un color especial a mis días.

A Sergio y Maria José, mis suegros, por tratarme como a un hijo.

Acknowledgments

First of all, I would like to express my deepest gratitude to my doctoral advisor, Professor Pedro José Clemente, whose unwavering support, guidance, and expertise have been invaluable throughout my doctoral journey. His dedication to fostering intellectual curiosity, encouraging critical thinking, and pushing the boundaries of research excellence has profoundly shaped my academic growth and personal development. I am indebted to him, his mentorship has been excellent, and I am truly fortunate to have had the opportunity to work under his guidance.

I wish to extend my heartfelt appreciation to the members of the Quercus research group, whose collaborative spirit, camaraderie, and dedication have enriched my doctoral experience immeasurably. I am deeply grateful to my brother, Arturo Barriga, whose support, encouragement, and shared passion for research have been a constant source of motivation and productive discussions. I am also thankful to my esteemed colleagues, Manuel Jesús, Pablo Alonso, Fernando Tejero, and Pedro Carmona. I am truly fortunate to have had the opportunity to collaborate with such talented and dedicated individuals, and I am sincerely grateful for their friendship and support throughout this journey.

I would also like to extend my sincere appreciation to the remaining members of the Quercus research group, whose dedication to scholarship and pursuit of excellence have created a vibrant environment for developing my thesis. While our interactions may have been less frequent, I am deeply grateful for sharing with me their passion for research, intellectual curiosity, and willingness to share their expertise.

Furthermore, I want to express my deep appreciation to José Moreira and Sónia Gouveia, my mentors during my research stay in Aveiro. I am sincerely grateful to them for not only making me feel at home but also for generously sharing their knowledge, providing insightful feedback, and inspiring me with their passion for research. Their invaluable contributions significantly contributed to the success of my thesis. I consider myself truly fortunate to have had the opportunity to learn from such esteemed mentors, and I am thankful for the profound impact they have had on my academic, professional, and personal development.

In the following paragraphs, I will express my gratitude for the support received from my closest friends and family, so I will switch to Spanish.

Quiero agradecer a todos mis amigos del pueblo, Luis, Eloy, Javi, Isma, Enrique, Joni y Miguel, a mis amigos de Madrid, País Vasco y Cataluña, Alex, Dani, Javi, Ander, Miguel, Iván, Ángela y Lidia, así como a todas sus parejas, Carlos, Anyelina, Sergio, Yaiza, Estela, Lidia y Julia, por todos los buenos momentos compartidos e inolvidables que guardamos juntos en nuestras memorias, por los que nos quedan, y por haber contribuido a ser lo que soy hoy día. Muchas gracias a todos y a todas.

Por último, quiero dar el mayor de mis agradecimientos a mi familia, las personas más importantes en mi vida. En primer lugar, a mi mujer, Victoria, por tanto amor incondicional, por su constante apoyo en cualquier proyecto que emprendo, y por ser como es, el amor de mi vida. En segundo lugar, quiero agradecer a mis padres, Pepi y Teófilo, por haberme guiádo cada día de mi vida para ser la persona que soy hoy, pese al esfuerzo que sé, les ha costado. Pero sobre todo, quiero agradecerles el haberme criado con tanto amor. También quiero agradecer a mi hermano Arturo, quién a pesar de nuestras discusiones, ocupa y ha ocupado siempre un lugar especial en mi corazón, y es que es mi hermano pequeño y a pesar de ello, es para mí un ejemplo a seguir. Otra persona muy importante a la que transmitir mis agradecimientos es a mi abuela, quién ya no puede comprender estos párrafos, pero no importa, porque ella sabe y me hace saber siempre lo orgullosa que se siente de mí, y lo mucho que me quiere. También están mi tía Felisa, Ángel, y mi prima, Carlota, quienes hacen de mis días, días especiales cuando los compartimos. Siempre han estado ahí, y no puedo estarles más agradecidos por ser conmigo como son. Por último, no me olvido de agradecer a mis suegros, Sergio y María José, quienes me acogieron en su familia como a un hijo, y me hacen sentir siempre que estamos juntos como tal.

Muchas gracias a todos, de corazón.

Abstract

The Internet of Things (IoT) is rapidly evolving and wide-spreading among several sectors and application areas such as Smart Cities, Agriculture, Transport (Internet of Vehicles or IoV), and Industry in general terms (Industrial IoT or IIoT). The broad domains and applications of the IoT make it a paradigm where a vast amount of heterogeneous technologies may collaborate and comprise the same system. In addition, the ever-evolving nature of the IoT, leads to the ongoing introduction of new technologies, increasing even more this diversity of devices. Furthermore, IoT systems are often designed without adhering to specific guidelines or methodologies, as there are no universally accepted standards.

This scenario of technological heterogeneity, ever-evolving nature, and lack of standards, makes IoT a complex paradigm whose development could face significant challenges. For this reason, testing processes are imperative to ensure that IoT systems perform as expected prior to deployment and production. Testing IoT systems is a costly and time-consuming process since involves device acquisition, configuration, and deployment among other tasks. To avoid these shortcomings, IoT simulators can be employed. However, IoT simulators often tackle simulations from a low level of abstraction, focusing on low-level details instead of on the high-level IoT concepts and their relationships. This leads to simulators with a prominent learning curve to overcome, low agility in designing and simulating IoT systems, and other difficulties that hinder comprehensively testing them.

Model-driven development (MDD) is a software development methodology that focuses on creating and using models to design and implement software systems. It abstracts the domain of a specific system by capturing its main concepts and relationships in a Metamodel. This Metamodel serves

as a high-level representation of the system, enabling users to focus on its core concepts without concerns about low-level details. Users can then create various models conform to this Metamodel, representing different systems. Subsequently, code generation tools can automatically produce code and other artifacts directly from these models, reducing the manual effort and error proneness associated with software development. Moreover, note that MDD is supported by tools such as graphical editors for designing models and validators for ensuring the integrity of these models. Thus, MDD significantly enhances productivity, maintainability, and consistency in software development by providing a Domain Specific Language (DSL) for modeling, code generation, and facilitating adaptation to evolving requirements.

To address the aforementioned challenges, this thesis dissertation introduces SimulateIoT, a novel MDD-based simulator designed to streamline the testing of IoT systems. SimulateIoT leverages the principles of MDD to offer a high-level abstraction framework for simulating complex IoT ecosystems. By employing a metamodel that facilitates the modeling and validation of IoT systems, along with model-to-text transformations for generating simulation code, configuration files, and deployment scripts, SimulateIoT significantly reduces the learning curve and enhances the agility of the design and simulation process. This approach not only simplifies the initial engagement with the tool but also provides the flexibility needed for redesign and testing, crucial for achieving optimal IoT system designs. Moreover, a methodology to support this process has been also developed.

SimulateIoT encompasses a comprehensive range of IoT components and simulation capabilities, including foundational elements, such as sensors, actuators, fog and cloud nodes; FIWARE architectures, including several components of the FIWARE catalog; mobile devices, together with the entire architecture to support them; and task-scheduling nodes and processes, to define environments where users can test and assess their task-scheduling proposals. Moreover, these components and simulation capabilities have been validated through several use cases, such as simulating an IoT environment deployed at the School of Technology of the University of Extremadura, a Smart Agricultural IoT system, an Animal Tracking system, a Smart City focused on managing mobile personal mobility devices, and on an Industrial IoT system (IIoT) based on the predictive maintenance of engines among others.

Note that the novelty of these contributions not only relies on providing a simulator capable of bridging the gaps often present in current IoT simulators but also in highlighting the potential of MDD techniques in managing the complexity of IoT systems for simulation purposes.

Thus, SimulateIoT represents an advancement in the field of IoT simulations, offering a versatile and user-friendly platform for the high-level abstraction modeling, validation, and testing of IoT systems through simulations. By addressing the primary challenges associated with traditional IoT testing, SimulateIoT paves the way for more efficient, cost-effective, and accessible development of IoT technologies, demonstrating the extensive applicability of MDD in overcoming the complexities of IoT ecosystems.

Resumen

El Internet de las Cosas (IoT) está evolucionando rápidamente y extendiéndose por varios sectores y áreas de aplicación como las Ciudades Inteligentes, la Agricultura, el sector del Transporte (Internet de los Vehículos o IoV) y la industria en general (IoT Industrial o IIoT). Los dominios y aplicaciones del IoT son muy diversos, convirtiéndolo así en un paradigma donde una gran cantidad de tecnologías heterogéneas colaboran entre sí y forman parte de un mismo sistema. Además, la naturaleza cambiante y evolutiva del IoT tiene como consecuencia la introducción constante de nuevas tecnologías, aumentando aún más su diversidad. Por otro lado, los sistemas IoT a menudo se diseñan sin adherirse a pautas o metodologías específicas, ya que no existen estándares universalmente aceptados.

Este escenario de heterogeneidad tecnológica, naturaleza cambiante y falta de estándares claramente definidos y aceptados, hace del IoT un paradigma complejo, siendo así el desarrollo de estos sistemas un desafío significativo. Por esta razón, los procesos de prueba son imprescindibles para asegurar que los sistemas IoT funcionan como se espera antes de su despliegue y puesta en producción. No obstante, comprobar el funcionamiento de estos sistemas es un proceso costoso y que consume tiempo, ya que implica la adquisición de dispositivos, su configuración y despliegue, entre otras tareas. Para evitar estas cuestiones, se pueden emplear simuladores IoT. Sin embargo, estos simuladores a menudo abordan las simulaciones desde un bajo nivel de abstracción, centrándose en detalles de bajo nivel en lugar de en los conceptos de alto nivel y sus relaciones. Esto conduce a simuladores con una curva de aprendizaje difícil de superar, a una baja agilidad en el diseño y simulación de los sistemas IoT, así como a otras dificultades que obstaculizan la prueba de éstos de manera integral.

El desarrollo dirigido por modelos (MDD) es una metodología de desarrollo de software que se enfoca en la creación y uso de modelos para diseñar e implementar sistemas de software. El MDD abstrae el dominio de un sistema específico capturando sus conceptos principales y relaciones en un Metamodelo. Este Metamodelo es una representación de alto nivel del sistema, permitiendo a los usuarios centrarse en sus conceptos principales sin tener que preocuparse por detalles de bajo nivel. Los usuarios pueden entonces crear varios modelos conforme a este Metamodelo, representando diferentes sistemas. Posteriormente, herramientas de generación de código pueden generar código y otros componentes de forma automática a partir de estos modelos, reduciendo así el esfuerzo manual y la propensión a errores relativos al desarrollo de software. Además, cabe destacar que el MDD integra herramientas como editores gráficos para diseñar modelos y validadores para garantizar la integridad de estos modelos. Así, el MDD puede mejorar significativamente la productividad, mantenibilidad y consistencia en el desarrollo de software al proporcionar un Lenguaje Específico del Dominio (DSL) para modelar, generar código y facilitar la adaptación a requisitos cambiantes.

Para abordar estos desafíos, esta tesis presenta SimulateIoT, un novedoso simulador basado en el MDD diseñado para agilizar las pruebas y validación de sistemas IoT. SimulateIoT aprovecha los principios del MDD para ofrecer un marco de abstracción de alto nivel con el objetivo de testear, mediante simulaciones, complejos sistemas IoT. Mediante el empleo de un metamodelo que facilita el modelado y validación de sistemas IoT, junto con transformaciones de modelo a texto para generar código de simulación, archivos de configuración y scripts de despliegue, SimulateIoT reduce significativamente la curva de aprendizaje y mejora la agilidad del proceso de diseño y simulación de éstos sistemas. Este enfoque no solo simplifica el uso inicial de la herramienta, sino que también proporciona la flexibilidad necesaria para diseñar y probar sistemas IoT, aspectos clave para implementar sistemas IoT óptimos. Además, se ha desarrollado una metodología que da soporte a todo este proceso.

En cuanto a sus capacidades, SimulateIoT abarca una amplia gama de componentes y posibilidades de simulación IoT, incluyendo elementos fundamentales, como sensores, actuadores, nodos Fog y Cloud; Arquitecturas basadas en la plataforma IoT FIWARE, incluyendo varios componentes de su catálogoE; Dispositivos móviles, junto con toda la arquitectura para brindar soporte a la movilidad de éstos; Y nodos y procesos enfocados en la planificación de tareas, para definir entornos donde los usuarios pueden probar y evaluar sus propuestas de planificación de tareas. Además, estos componentes y capacidades de simulación han sido validados a través de varios casos de uso, como un entorno IoT desplegado en la Escuela Politécnica de la Universidad de Extremadura, un sistema IoT agrícola inteligente, un sistema de monitorización de animales, una ciudad inteligente centrada en la gestión de dispositivos de movilidad personal móviles, y un sistema de IoT industrial (IIoT) basado en el mantenimiento predictivo de motores, entre otros.

Cabe destacar que la novedad de estas contribuciones no solo se basan en proporcionar un simulador capaz de solventar la problemática que suelen presentar los simuladores IoT actuales, sino también en resaltar el potencial de las técnicas MDD en la gestión de la complejidad de los sistemas IoT para fines de simulación.

Así, SimulateIoT representa un avance significativo en el campo de las simulaciones IoT, ofreciendo una plataforma versátil y fácil de usar para el modelado de alto nivel de abstracción, validación y pruebas de sistemas IoT a través de simulaciones. Al abordar los desafíos asociados a los procesos de prueba tradicionales de sistemas IoT, SimulateIoT allana el camino para el desarrollo eficiente, rentable y accesible de tecnologías IoT, demostrando la aplicabilidad del MDD para superar estos desafíos.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"We ourselves feel that what we are doing is just a drop in the ocean. But the ocean would be less because of that missing drop"

Work unknown (Year unknown)
Attributted to Mother Teresa

The Internet of Things is a paradigm where a vast amount of heterogeneous technologies collaborate to deliver a wide spectrum of services [6, 7], Smart Homes [8], Smart Cities [9], Healthcare (Internet of Medical Things) [10], Industry (Industrial IoT) [11], Agriculture [12], Smart Grids and Energy Management [13], Connected Vehicles and Transportation (Internet of Vehicles) [14], are some of the most popular application domains of the IoT.

The landscape of technological heterogeneity, constantly evolving nature and absence of standardized guidelines make IoT a complex paradigm, thus presenting significant challenges. Consequently, rigorous testing processes are imperative to ensure the expected performance of IoT systems prior to deployment and production. In this respect, testing these systems is costly and time-consuming, as it involves activities such as device procurement, configuration, deployment, and more. However, simulations can be performed to test these systems before putting them into production [15].

Simulations avoid device acquisition, simplify configuration and deployment processes, and facilitate the assessment of the system [15]. However, given the inherent complexity and technological heterogeneity of the IoT, IoT simulators often focus on low-level details, such as specific device configurations or network intricacies [16]. This not only heightens the learning curve for users but also restricts the agility in designing and deploying IoT simulations. Thus, leading to increased time, costs, and efforts to carry out simulations [16].

Model-driven Development [17] can help to overcome the aforementioned scenario by elevating the level of abstraction at which IoT simulations are tackled. Focusing on high-level concepts and their relationships, MDD enables users to focus on the core aspects of their systems, rather than dealing with their low-level details, which can be managed and included in the simulations seamlessly to the user. As a result, software code can be generated for a particular technological platform, enhancing technological independence and reducing susceptibility to errors.

In the context of the IoT simulations, these features are highly appealing. This is mainly because of the inherent complexity of the IoT, characterized by the high technological heterogeneity it encompasses. With the MDD, IoT simulations can be tackled from a high level of abstraction, focusing on its high-level concepts, rather on low-level details. Moreover, model-to-text transformations have a key role since they can generate all the simulation's artifacts for different target technologies, thus decreasing error proneness and increasing user productivity in terms of carrying out simulations [17].

Thereby, in this Ph.D. Thesis, an MDD-based IoT simulator called SimulateIoT has been developed. This work aims to explore to what extent MDD techniques are suitable to simulate IoT systems considering several critical aspects of the current IoT simulators, such as the agility to design and simulate IoT systems. This simulator allows users 1) to design their systems through graphical models; 2) to validate their models to verify the correctness of their designs; 3) to generate the code related to the system simulation from the modeled system; and 4) to simulate and assess the system. Note that each stage operates at a high level of abstraction, thus streamlining the testing process of IoT systems.

Next, an overview of the research conducted in this Ph.D. Thesis is provided. Section 1.1 details the research context of the thesis. The problem statement is articulated in Section 1.2. The aims and specific research questions guiding this doctoral research are presented in Section 1.3. A summary of the principal contributions of this Ph.D. Thesis is provided in Section 1.5. Finally, the structure of the thesis is delineated in Section 1.6.

1.1 Research Context

This thesis is focused on the exploration of the extent to which MDD techniques are suitable to simulate IoT systems, considering several critical aspects of the current IoT simulators, such as the agility to design and simulate IoT systems. Thus, the research context encompasses the MDD foundations and several IoT concepts. In this respect, note that the IoT is a wide domain, being not feasible to capture its whole domain in a single MDD approach. Therefore, the IoT concepts covered have been selected to achieve the primary aim of this Ph.D. Thesis, which is stated in Section 1.3. These concepts are listed below:

- The foundation of IoT systems, such as the multi-layered computing architecture of these systems (Cloud, Fog, Edge, and Mist layers) and its devices, such as Cloud and Fog nodes, or sensors and actuators [18].

- IoT mobility, concept included to provide IoT devices with the ability to move and communicate in various environments seamlessly, supporting continuous connectivity and interaction regardless of location [19]. This includes mobile devices and the architecture to support them.

- Task scheduling, concept that refers to the process of organizing, allocating, and managing tasks across a network of interconnected devices and computing nodes. This involves the coordination between various nodes capable of generating and offloading tasks (such as IoT devices or applications deployed on computing nodes) and computing nodes (including Edge, Fog, and Cloud computing resources) that are responsible for scheduling and processing these tasks [20]. Note that task scheduling aims to optimize resource utilization, improve system performance, and ensure timely task execution by considering factors like computational load, network bandwidth, and latency [20].

- FIWARE, an open-source IoT platform that offers a standardized framework, based on a catalog of components [21], aimed at simplifying the development of IoT systems across various domains like smart cities and smart agriculture [22].

Thereby, the following sections provide a detailed description of the aforementioned research context aspects addressed in this Ph.D. Thesis. Section 1.1.1 introduces MDD techniques. Then, Section 1.1.2 delves into the multi-layered computing architecture in which current IoT systems are grounded. Subsequently, Section 1.1.3 addresses the mobility concepts that have to be taken into account in nowadays mobile IoT systems. Later on, Section 1.1.4 outlines the role of task scheduling within the IoT. Lastly, the IoT FIWARE platform is presented in Section 1.1.5.

1.1.1 Model-Driven Development

MDD is an evolving area within software engineering that focuses on the creation of software guided by models. To do so, MDD employs the Metamodeling technique [17, 23]. Metamodeling is delineated through four distinct model layers, as illustrated in Figure 1.1.

At this hierarchy, a Model (M1) conforms to a Metamodel (M2), while a Metamodel, in turn, conforms to a MetaMetaModel (M3), which possesses reflexive properties [24]. The MetaMetaModel level encompasses well-established standards and specifications, such as the Meta-Object Facilities (MOF) [25], and ECore in the Eclipse Modeling Framework (EMF) among others [26]. A Metamodel is designed to define domain concepts and relationships within a specific domain, thereby capturing a segment of reality. A Model (M1) then delineates a concrete system in accordance with the Metamodel. From these models, it is feasible to generate application code (M0 - code) either wholly or partially through model-to-text transformations. This process allows high-level definitions (models) to be translated via model-to-text transformations into specific technologies (target technology) [27].

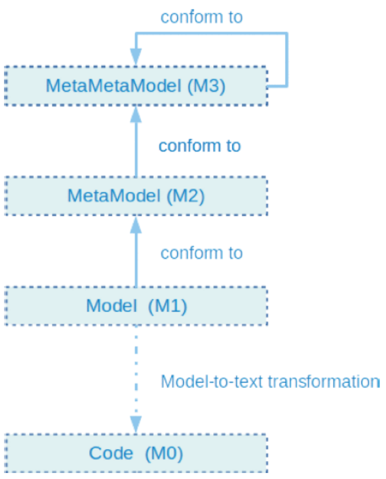


Figure 1.1: Model-driven development. Four layers of metamodeling.

As a result, software code can be generated for a particular technological platform, enhancing technological independence and reducing susceptibility to errors.

In summary, MDD makes it possible to increase the abstraction level where the software is developed, focusing on the domain concepts and their relationships rather than on low-level aspects. These domain concepts and their relationships are defined by a model (M1), conform to a metamodel (M2), which can be analyzed and validated using MDD techniques. Furthermore, the IoT system code, including all the artifacts needed, can be generated from a model (M1) using model-to-text transformations, decreasing error proneness and increasing the user's productivity.

The subsequent sections will delve into the Internet of Things (IoT), the field of application where the previously described MDD technique has been employed in this thesis dissertation.

1.1.2 The Internet of Things From a Multi-Layered Computing Perspective

The IoT concept was coined in 1999 by Kevin Ashton, a British technology pioneer [28]. At that time, he used the term to describe a system where objects are connected to the internet through devices, such as sensors, allowing them to be tracked and managed automatically. His vision was that objects could be controlled and communicated with over the internet, leading to an integration of the physical world into computer-based systems, resulting in improved efficiency, accuracy, and economic benefit [28].

Since the advent of the IoT, many technologies and developments have contributed to what we now know as IoT [7, 29]. Currently, the IoT embodies a paradigm structured through multiple computing layers, wherein a vast amount of heterogeneous devices, services, and applications collaboratively function to meet specific user requirements [30]. To delve into the finer aspects of the IoT paradigm, an analysis is conducted from a multi-layered computing perspective of each computing layer that forms the foundation of the IoT architecture. The Cloud, Fog, Edge, and Mist layers. This analysis includes an examination of the types of devices that form each layer and the role of the aforementioned layers within the broader IoT ecosystem. Figure 1.2 shows an outline of these layers and the devices that comprise them. To address this analysis in an ordered manner, a logical and chronological order is followed. First, the Mist layer, which encompasses the end devices of IoT systems, is presented. Subsequently, the Cloud layer, integrated into the IoT to provide end devices with high computing and storage capabilities, is described. Then, the layers conceived to bring the Cloud close to end devices, i.e. the Fog and Edge layers, are outlined. The analysis can be found below.

Firstly, end devices also referred to as *Things*, constitute what is known as the Mist layer [30]. These devices predominantly fall into two categories: sensors and actuators. Sensors are designed to sense real-world phenomena; for instance, thermometers measure temperature, while hygrometers gauge humidity. Conversely, actuators are devices designed to interact with the real world; for instance, a heater can elevate the temperature within a designated space, while a dehumidifier can reduce ambient moisture levels. Standalone sensors and actuators often can not process data nor connect to the Internet themselves. So, they are attached to computing modules that not only facilitate operational logic for these devices (e.g. turn on/off depending on a specific input) but also serve to connect them to the internet. Thus, enabling communication between these devices and the rest of the IoT system [31, 32]. Note that the union between a sensor or an actuator,

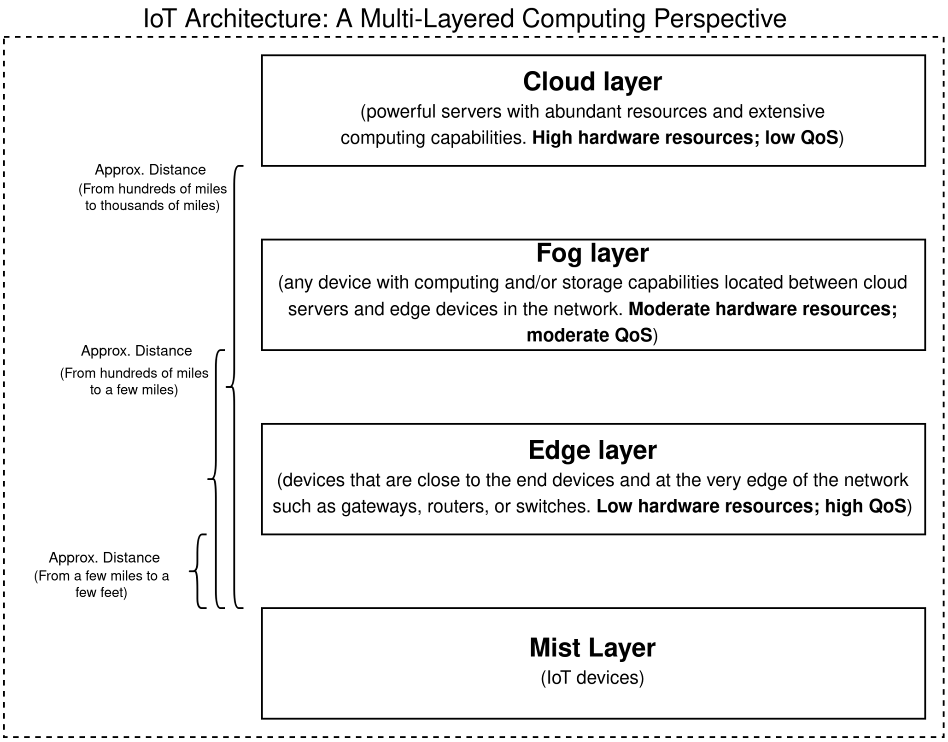

IoT Architecture: A Multi-Layered Computing Perspective
Cloud layer
(powerful servers with abundant resources and extensive computing capabilities. **High hardware resources; low QoS**)
Fog layer
(any device with computing and/or storage capabilities located between cloud servers and edge devices in the network. **Moderate hardware resources; moderate QoS**)
Edge layer
(devices that are close to the end devices and at the very edge of the network such as gateways, routers, or switches. **Low hardware resources; high QoS**)
Mist Layer
(IoT devices)
Approx. Distance (From hundreds of miles to thousands of miles)
Approx. Distance (From hundreds of miles to a few miles)
Approx. Distance (From a few miles to a few feet)

Figure 1.2: IoT architecture from a multi-layered computing perspective.

or a set of these, with a computing module conforms to what is known as an IoT device. A device with internet connectivity that can collect, transmit, and sometimes process data, exemplified by devices like smartwatches or connected vehicles [31, 32].

On the other side, in the hierarchical layered structure depicted in Figure 1.2, the uppermost tier is represented by the Cloud layer. This layer is comprised of powerful servers with abundant resources and extensive computing capabilities. Indeed, this layer represents the integration of Cloud computing within the IoT ecosystem, a cornerstone in current IoT systems [33]. Typically, IoT devices are not equipped with powerful hardware, constrained by factors such as battery consumption or the cost of the device. Consequently, the Cloud layer is leveraged to undertake processing tasks beyond the capabilities of the IoT devices. Thus, enabling IoT devices to adhere to the aforementioned constraints [34].

However, devices' constraints represent only one aspect of the requirements that must be addressed in an IoT system. Current IoT systems also encompass applications, which frequently tend to be distributed rather than monolithic, typically structured using a microservices architecture (MSA) [35]. In this context, the Quality of Service (QoS) refers to the set of performance metrics that govern the overall performance and reliability of a service or application [36]. QoS encompasses various parameters such as latency, bandwidth, throughput, error rates, availability, cost, or battery consumption. QoS is particularly important in scenarios where real-time data processing and response are critical, such as the Industrial IoT (IIoT) [37] and the Internet of Vehicles (IoV) [38]. For instance, in the IoV, minimal response times are imperative since any delay could have potentially fatal consequences for the driver or others [38]. Furthermore, in the context of electric vehicles, maintaining low costs and efficient battery consumption is essential, not only to ensure the vehicle's competitiveness in terms of price but also to enable significant travel distances without the need for frequent recharging [39]. Consequently, the services that form part of an IoT system together with the infrastructure that supports them may be required to adhere to stringent QoS constraints, which can vary depending on the specific operational context of each service.

In this scenario, relying only on the Cloud layer may prove insufficient for meeting some QoS requirements [40]. This limitation arises because Cloud providers, such as Google or Amazon, do not uniformly distribute their Cloud servers across all potential IoT deployment locations. In some instances, there may not even be a Cloud server within the same country as the deployed IoT system. For example, considering Google's Cloud infrastructure in Spain, the only Cloud servers are located in Madrid. Consequently, for regions like Extremadura, which is approximately 297 kilometers from Cáceres to Madrid, Madrid's data center is the nearest [41]. This distance could lead to an insufficient QoS (e.g. response time) for certain IoT applications. So, further solutions have been explored to bridge this gap.

Among these solutions, Fog computing, proposed by CISCO in 2012, emerged as a prominent approach [42]. According to CISCO, Fog computing can be understood as a decentralized computing infrastructure that extends the Cloud through the placement of end nodes (IoT devices). Therefore, devices involved in Fog computing can be those with computing and storage capabilities, strategically positioned between the Cloud and IoT devices in the network. So, Fog computing puts data, compute, storage, and applications nearer to the user or IoT devices where the data needs processing, thus creating a Fog outside the centralized Cloud, and reducing the data transfer times necessary to process data [43]. Consequently, despite Fog computing offering limited computing and storage resources compared to Cloud computing, it delivers better QoS for a wide range of applications and services. Furthermore, it is noteworthy that Fog computing, as an extension of the Cloud to end devices, allows for reducing the traffic sent to Cloud servers. Thus, improving network congestion and therefore the quality of service of the Cloud layer [42]. As a result, Fog computing has merged with the IoT paradigm through the establishment of the Fog layer, illustrated in Figure 1.2 between the Cloud and Edge layers. Considering said advantages, nowadays, the Fog layer is an indispensable component within the architectural framework of IoT systems [44, 45].

However, as previously described, the Fog layer consists of devices positioned between the Cloud layer and the Mist layer. Consequently, a Fog device might be located closer to the end devices or nearer to the Cloud servers. This variability in spatial distribution can lead to a scenario where certain Fog devices, particularly those situated at greater distances from the end devices, may not be capable of fulfilling the most stringent QoS requirements. In this scenario, Edge computing plays a key role [46], constituting a fundamental element in critical IoT systems such as IIoT or IoV [47, 48]. Since Edge computing is primarily aimed at fulfilling the most stringent QoS requirements, in the architecture of IoT systems, the Edge layer is positioned between the Fog layer and the Mist layer, thus being exclusively composed of devices that are so close to the end devices [46]. Likewise, Edge devices reside at the very Edge of the network, marking the boundary beyond which the Fog layer extends. Devices such as gateways, routers, or switches could be Edge devices. The positioning of these devices is key for facilitating immediate data processing and decision-making, thus meeting the most stringent QoS requirements of critical IoT systems. Finally, it is important to note that, analogous to the relationship between the Fog and Cloud layers, while the Edge layer offers better QoS than the Fog layer, the Edge layer typically has more limited computing and storage capabilities compared to the Fog layer [46].

In summary, the IoT is a multi-layered computing paradigm, characterized by a diverse array of devices, services, and applications working together to fulfill specific user needs [30]. Typically, the lower layers, such as the Mist and Edge layers, where data originates, are equipped with more limited computing and storage capacities. However, they provide better QoS due to their proximity to end devices [42, 43, 46]. In contrast, the higher layers, such as the Cloud and the Fog layers, present greater computational and storage resources but may deliver comparatively lower QoS, attributable to their increased distance from these end devices [40, 33].

This section has provided an overview of the foundational aspects of the IoT paradigm from a multi-layered computing perspective. Subsequent sections will delve into more specific areas within the wide IoT scope. The forthcoming section will focus on the aspect of mobility within IoT systems.

1.1.3 Mobility in the IoT

In an IoT system, not all devices are stationary, but some might move or even need to move to perform their functions, a characteristic that should be taken into account when designing these systems. In this scenario, the concept of IoT mobility refers to the ability of IoT systems to support mobile devices, applications, and services [49]. This involves not just the physical movement of devices but also the seamless transfer of data and consumption/provision of services across different networks and environments [50].

IoT applications exhibit a wide range of diversity, each with its unique set of requirements and needs. IoT mobility requirement arises in specific contexts where device mobility is essential for operational purposes. For instance, personal mobility devices (PMD's) such as bicycles and scooters, available for rent in urban settings, exemplify this need [51]. Similarly, GPS sensors attached to animals on extensive farms serve as another example [52]. Likewise, manufacturing and industrial processes may necessitate the deployment of mobile IoT devices throughout a factory setting. Currently, there does not exist a specific standardized architecture particularly tailored to support mobile devices within the IoT. Instead, there exists a plethora of technologies, standards, and protocol stacks to use, depending on the application domain of the IoT system [53]. Therefore, this section is dedicated to discussing some of the key aspects to take into account when dealing with IoT mobility.

A pivotal aspect of IoT mobility lies in the wireless connectivity that interconnects the different devices of an IoT ecosystem. This connectivity not only aims at maintaining constant communication but also ensures that such communication is resilient and adaptable to a variety of scenarios and conditions that arise from mobility. As mobile devices move between different locations and network coverage areas, the ability to seamlessly switch connections, maintain data integrity, and ensure minimal service disruption is crucial. This requires architectures, handover strategies, and protocols that can manage these transitions smoothly [54].

On the other hand, energy consumption is a critical factor in any IoT system. This is because of the devices' constraints previously discussed in Section 1.1.2. However, it often becomes more pronounced and challenging in mobile IoT systems compared to stationary IoT systems [55]. For these reasons, the optimization of data transmission frequency, the processing power of the device, and the activity level of sensors, particularly in persistent monitoring applications, play a significant role in minimizing energy consumption. Furthermore, network stability is another critical factor as unreliable connectivity can lead to increased power usage due to frequent signal searches, connections, and the need for data retransmission. So, the longevity and effectiveness of IoT systems are significantly influenced by their ability to manage these aspects, thus conserving power efficiently [55].

In addition, security is another essential characteristic in the IoT landscape, with the mobility of devices introducing additional layers of complexity that necessitate advanced security measures. In mobile IoT scenarios, devices frequently transition across different network environments. If not properly managed, this mobility can expose them to several network vulnerabilities and potential security threats [53]. In this regard, the deployment of secure authentication protocols is vital. These protocols ensure the verification of devices' identities, thereby preventing unauthorized access and ensuring that only authenticated devices are granted access to the IoT system [53, 56].

In summary, supporting mobile devices and seamless data communication across networks faces several challenges such as the aforementioned, i.e. wireless connectivity management, energy consumption and security. Thus, to overcome these challenges, aspects such as robust connectivity, energy awareness, and ensuring security are key.

1.1.4 Task Scheduling in the Cloud-to-Thing Continuum Paradigm

As described in Section 1.1.2, the architecture of IoT systems can be conceptualized from a multi-layered computing standpoint. From this perspective, IoT systems are structured into several computing layers: the Mist, Edge, Fog, and Cloud layers. Each of these layers presents different characteristics, thus providing a diverse range of possibilities in terms of computing and storage resources, QoS, and other relevant aspects [30]. However, one of the most interesting features of this multi-layered computing architecture that has not been covered in Section 1.1.2, is the potential for federation among the nodes comprising each of these layers. Federations enable the nodes of an IoT system to collaborate and act as a single entity rather than isolated nodes. Moreover, federations can include nodes that belong to different layers, constituting a Cloud-Fog-Edge-Mist heterogeneous federation [57]. The Cloud-to-Thing Continuum paradigm emerges [58].

The Cloud-to-Thing Continuum paradigm can be defined as the orchestrated coordination of services and resources across the various computing layers within an IoT system. This paradigm facilitates the seamless flow of data through Cloud data centers, intermediary nodes such as Fog or Edge nodes, and ultimately to end devices. Consequently, the Cloud-to-Thing Continuum paradigm is a holistic approach that encompasses the entire spectrum of IoT architecture, from the Cloud layer to the Mist layer, thus enhancing data processing, decision-making, and system responsiveness by strategically leveraging resources in closer proximity to the data source when necessary [58].

As the infrastructure of the IoT has continued to evolve and advance, there has been parallel progress in the optimization of resource management within these systems. A fundamental strategy in this regard is task scheduling, a well-known technique extensively utilized in distributed computing environments similar to IoT systems structured around the Cloud-to-Thing Continuum paradigm [59]. In such environments, services are typically broken down into a series of tasks, each representing an individual unit of work or a specific job to be executed. These tasks can range from data collection and processing to control commands and other computational processes. Within the context of IoT, task scheduling aims to strategically allocate the execution of these tasks across the computing nodes of these systems. The overarching goal is to enhance system efficiency by optimizing the use of available resources from various angles [60]. For example, some task scheduling proposals focus on reducing the makespan, which is the total time required to complete a given task [61], others prioritize minimizing system energy consumption [62] or reducing the costs in terms of money associated with task processing [63], while others aim for several objectives, i.e., multi-objective approaches that seek to achieve a balance between two or more of the aforementioned aspects [64]. Thus, by effectively scheduling tasks, these proposals aim to optimize overall system performance, addressing key operational aspects such as time, energy, and cost among others.

In summary, the Cloud-to-Thing Continuum paradigm and task scheduling approaches are highly synergistic, thus achieving optimal utilization of the resources of IoT systems and the effective execution of tasks throughout all their layers, from Cloud to Things.

1.1.5 IoT Platforms: FIWARE, an IoT Platform for Developing and Deploying IoT Environments

Given the complexity of IoT systems, outlined through sections 1.1.2, 1.1.3 and 1.1.4, to facilitate their development, multiple technologies are available from configuring a specific sensor to analyzing a vast amount of data in real-time. In this context, IoT platforms are tools designed to support and manage IoT devices and applications. These platforms provide a suite of tools and services that help in connecting devices and accessing and managing data. There are several IoT platform providers, such as Microsoft Azure IoT Hub [65], ThingSpeak IoT Platform [66], Thingworx IIoT Platform [67], Things Network [68], and FIWARE [22] among others. Each of these IoT platforms presents distinct characteristics and mechanisms for defining devices, establishing connections, storing and analyzing data, as well as generating notifications [69].

Specifically, FIWARE is an open-source project that defines and implements a universal set of standards for context data management intending to optimize the development of IoT environments in different fields, such as Smart Cities, Smart Buildings, Smart Agro, Smart Energy, Smart Industry, etc.

Within the FIWARE framework, *context* refers to the state of the IoT environment at any given moment. Therefore, context elements or data are those that provide meaning to the environment [70]. For instance, they can define environmental characteristics such as temperature or wind speed, or architectural aspects like the position of an element or its movement speed. Thus, FIWARE makes IoT simpler by driving key standards for breaking the information silos, transforming Big Data into knowledge, enabling data economy, and ensuring sovereignty over data [71].

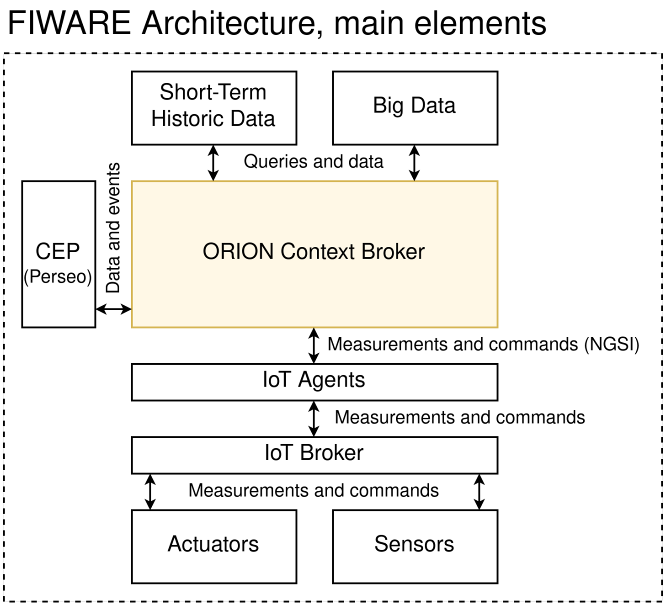


Figure 1.3: Main elements of FIWARE architecture.

Figure 1.3 shows the main elements that constitute the FIWARE architecture [21]. Each of these elements is addressed below: A) The core and indispensable component of any FIWARE solution is the Orion Context Broker. The Orion Context Broker manages the entire data lifecycle within an IoT system powered by FIWARE. This includes the creation and registration of context elements in the system and their subsequent management [71]. Context elements could be sensors, actuators, and Complex Event Processing (CEP) engines [72] among others; B) On the other hand, the Short-Term Historic Data and Big Data elements provide the FIWARE architecture with data persistence, thus enabling the Orion Context Broker to maintain an ongoing awareness of the IoT system's context at any given moment, which facilitates the effective management of the data lifecycle of the system [71]; C) Sensors and actuators have the same role that in any other IoT system, serving as fundamental components for data collection and action initiation; D) Additionally, the IoT Broker, a central element in publish/subscribe protocols commonly employed in IoT environments, such as MQTT [73], is employed by FIWARE. This broker is used for aggregating and dispatching data to and from sensors and actuators, thereby interconnecting them with the rest of the FIWARE ecosystem; E) Meanwhile, the IoT Agents serve as interoperability components that bridge the Orion Context Broker, which utilizes the NGSI protocol [74], and the different IoT brokers that FIWARE offers users to integrate into their IoT systems [75]. It is noteworthy that, although not depicted in Figure 1.3, FIWARE supports not only publish/subscribe protocols but also protocols such as LoRaWAN [76] or Sigfox [77]. Accordingly, FIWARE provides IoT agents that facilitate the integration of these protocols with the Orion Context Broker; F) Finally, the CEP component facilitates real-time data processing [72]. This is achieved by applying users' pre-defined rules to the data, which, when met, trigger corresponding events and notifications. For instance, to automatize and control the operation of actuators.

In summary, the development and management of IoT systems can be partially addressed by using IoT platforms like FIWARE. As depicted in Figure 1.3, FIWARE offers several solutions in terms of functionality, interoperability, data governance, etc. by means of its catalog of components [21]. Thus, facilitating users in the building and managing of IoT systems, from device configuration to advanced data processing.

1.2 Problem Statement

Systems based on the IoT are experiencing continuous growth and are being increasingly applied in numerous domains, including smart cities [78], home environments [79], buildings [80], agriculture [81], and industry [82]. However, as discussed in Section 1.1, IoT systems are complex ecosystems that integrate a wide array of heterogeneous technologies, encompassing a vast amount of devices, various protocols, optimization techniques, services, and applications among others [7]. Furthermore, the continuous evolution of the IoT leads to the emergence of even more technologies, thereby expanding the possibilities but also adding to the complexity of IoT ecosystems. Additionally, the field of the IoT is characterized by the absence of universally accepted standards in terms of, for instance, system deployment, the development of IoT software artifacts, interoperability, and the orchestration of the various services and applications involved in the system [83]. This landscape, characterized by a lack of standardization, diverse technologies and solutions, coupled with the ever-evolving and dynamic nature of IoT, poses a formidable challenge within the field.

On the one hand, during the design stage of an IoT system, users are required to make several key decisions concerning the system's infrastructure. For instance, some of these decisions could include: A) Pertaining to the Mist layer, selecting the appropriate types and quantities of sensors and actuators, as well as their optimal placement [84]; B) For the Edge and Fog layers, the choice of appropriate hardware and devices to constitute these layers, the strategic deployment locations for each Edge and Fog node, and determining the computing power and storage capacity that these nodes should have to meet the requirement of the system (e.g. desired applications' and services' QoS) [85]; C) In the context of the Cloud layer, selecting the most suitable Cloud service providers for the IoT system, taking into account factors such as the cost, QoS, availability, elasticity, and other relevant parameters related to the services offered by Cloud providers [85].

Moreover, pivotal decisions related to system deployment and functioning are also essential. For instance, some of these decisions could include: A) In terms of service and application deployment, determining the most suitable nodes within the various computing layers for deploying each service and application, with a focus on meeting the diverse QoS requirements considered essential for the system's correct functioning [86, 87, 88]; B) Regarding software components such as CEP engines or IoT brokers that could be part of the system, selecting the most suitable IoT platform; C) For sensors and actuators in general, the frequency of data gathering and publishing, respectively [89]. For mobile devices, defining and predicting the routes that mobile devices should or could take [90]; D) Additionally, in the context of task execution, it is imperative to identify the most effective strategies for federating the system, which includes determining how many different federations could co-exist within the system, selecting those task scheduling proposals that best align with the system's needs across different potential scenarios, etc. [62, 91].

These represent some of the key design decisions that users may deal with. While these decisions are high-level in terms of abstraction, numerous additional low-level decisions are essential for the effective design, deployment, and performance of an IoT system. These include internal device configuration, the development of ad-hoc software solutions, and the management of low-level networking aspects, among others.

Given the complexity of IoT systems, the wide spectrum of devices that could be involved, the lack of standards, as well as the numerous decisions that users have to face during the design stage, comprehensive testing is essential before deploying and putting the designed IoT system into production [92]. Thus, allowing users to assess the system's performance and identify unexpected behaviors. To conduct a thorough assessment of the system, these tests must encompass a variety of scenarios, including stress testing under peak loads to evaluate system performance and stability [93]; compatibility checks across different devices and platforms to ensure seamless operation [94]; QoS assesments to guarantee services' and applications' reliability and responsiveness by measuring their latency, throughput, packet loss, and error rates under diverse network conditions [95, 96]; assessments about the different devices of the system during any scenario that the system could face, not only to ensure correct operation during peak loads but also to optimally select the computing power and storage capacity of each device [97]; for battery-operated devices, power consumption and battery life testing should be also undertaken to ensure that these devices will perform correctly and that the power consumption of the overall system is as expected [98, 99]; failover and recovery testing to evaluate the system's resilience [100, 101]; and load-balancing tests for systems distributing tasks across multiple nodes, thus assessing if the task distribution is as expected during system functioning [102].

These are some of the tests that users should conduct prior to deploying their designed IoT systems into production. By conducting these tests, users can gain knowledge and insights that would enable them to verify the correct design of their systems and reduce the risk of unforeseen scenarios that could result in system malfunctions or partial failures.

However, carrying out these tests involves a significant investment of money, time, and effort in acquiring devices, their configuration, deployment, etc. Moreover, the continuously evolving nature of the IoT, which leads to increasing complexity over time, not only escalates the costs associated with these tests but also amplifies their necessity.

In an effort to bridge this gap, several tools have been proposed in the literature. Specifically, simulation and emulation tools are widely recognized and adopted to conduct IoT system tests [103, 104, 105]. These tools often enable users to design, test, and validate their IoT systems through simulations, thereby avoiding the aforementioned costs typically associated with such processes. For instance, acquiring IoT devices is not yet required with most of these simulation tools. However, there is a wide spectrum of IoT simulators, each of them with different features to both 1) design simulations, i.e. how the simulator allows users to design the simulations; and 2) simulate, i.e. kind of IoT system or elements of an IoT system that allows to simulate.

Given this scenario and the challenges described throughout this section, the desired characteristics for an IoT simulator to simulate IoT systems comprehensively and in an effortless and cost-efficient manner could be summarized in: 1) Adaptive integration capability, 2) User-friendly learning curve, 3) Efficient design agility, and 4) Cost-efficiency. These features are described below:

- Adaptive integration capability: Capability to integrate new elements to the simulations that are not covered by default by the simulator, such as a new kind of device, or a task scheduling proposal that a user has developed and wants to test. This is crucial for the simulator's usability and to cover wide scopes within the IoT. This feature is an enabler to simulating comprehensive IoT systems tailored to the specific simulation needs of the stakeholders, i.e. the IoT systems or IoT elements that users want to simulate. This characteristic is also especially appealing due to the ever-evolving nature of the IoT, where technologies are constantly being updated and new ones are being included, as is the case of certain users' proposals, such as recently published task scheduling algorithms [106, 107, 108].

- User-friendly learning curve: This is a critical aspect since stakeholders often have limited resources regarding time and money to overcome the learning curve of an IoT simulator.
- Efficient design agility: A key capability to enable users to design and re-design IoT simulations in a fast and effective manner.
- Cost-efficiency: Finally, IoT simulators should be cost-efficient, as carrying out IoT simulations should require a reasonable amount of time, money, and effort. This characteristic is closely related to the three aforementioned since it could be mostly fulfilled by meeting them.

However, most existing simulators present significant variations regarding the desired characteristics listed above. Therefore, efforts should be aimed at providing IoT simulation tools that meet the above-described features. Bridging this gap would enable users to conduct tests regarding their IoT systems in a comprehensive, user-friendly, agile, and cost-efficient manner.

1.3 Aims and Research Questions

Accordingly to the research context and the stated problem presented in sections 1.1 and 1.2, respectively, ***the primary aim of this Ph.D. Thesis is to provide the IoT community with a tool that enables them to comprehensively test and enhance their IoT system designs without the cost, time, and effort often associated with this process.***

To fulfill this objective, a set of four Research Questions (RQs) has been formulated. By addressing these RQs, this Ph.D. Thesis intends to achieve the aforementioned aim. The RQs are as follows:

- **RQ1. To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?** As outlined in Section 1.1.1, MDD increases the abstraction level where the software is developed, focusing on the domain concepts and their relationships rather than on low-level aspects. Consequently, by increasing the abstraction level in the context of IoT systems, MDD could contribute significantly to addressing the technological heterogeneity of these systems. Thus, managing the broad spectrum of technologies and low-level details, such as hardware configurations or device communications features that these systems may incorporate. Therefore, this RQ seeks to ascertain the suitability of applying MDD in this context.

- **RQ2. To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?** Even if MDD techniques are capable of handling and managing the complexity of IoT systems by increasing the abstraction level from which they are addressed, it remains unclear whether these techniques can be effectively used to develop tools for simulating IoT systems. However, as outlined in Section 1.1.1, and as can be seen in Figure 1.1, MDD allows to generate code (M0) from models (M1) modeled conforms to a metamodel (M2). This process suggests the potential feasibility of generating the code related to the simulation of a modeled IoT system. Nevertheless, given that at the beginning of the development of this Thesis there were no IoT simulators based on the MDD, is not possible to conclusively determine whether code for simulating IoT systems can be effectively generated from models using MDD. Given that this would be key in developing an MDD-based IoT simulation tool, this RQ seeks to explore this possibility.

- **RQ3. In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve**

but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them? Even though MDD-based simulators are feasible, it would remain unclear whether these simulation tools would exhibit adaptive integration capabilities, a user-friendly learning curve, agility in designing IoT systems, and cost-efficiency in testing and validating them. Crucial characteristics that an IoT simulation tool should meet as discussed in Section 1.2. Consequently, this RQ aims to resolve this dilemma.

- **RQ4. To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?** If the aforementioned RQs yield affirmative results, the development of a formal and methodological approach for testing and refining IoT systems until an optimal design is achieved, may be feasible. Consequently, this particular RQ is dedicated to investigating methodologies capable of integrating a testing and refinement process with an MDD-based IoT simulator. The goal is to establish a formal and methodological approach for testing and refining IoT systems in a manner that is not only agile and straightforward but also cost-efficient.

1.4 Research Methodology

Over the years, Design Science (DS) research [1] has gained prominence in the field of Information Systems (IS), with a focus on developing successful artifacts (models, methods, software components, or tools). DS research has been applied in the creation of SimulateIoT, the MDD-based IoT simulator proposed in this Ph.D. Thesis, along with all the artifacts associated with it. However, several DS research approaches could have been applied for this purpose. Among others, it is possible to highlight Systems Development Research Methodology [109], DS Research Process Model [110], Action Design Research [111], Soft Design Science Methodology [112], Participatory Action Design Research [113] and Design Science Research Methodology (DSRM) [114].

In the course of this Ph.D. Thesis, each of these methodologies has been carefully studied aiming to select the most appropriate for the development of the IS artifacts involved in this research. Moreover, frameworks intended to assist in this selection process were also explored. Specifically, the framework proposed in [115] by some of the most renowned researchers in the field of DS research, such as John R. Venable, Jan Pries-Heje, and Richard L. Baskerville, was applied. After applying this framework, DSRM was identified as the most suitable for the development of SimulateIoT.

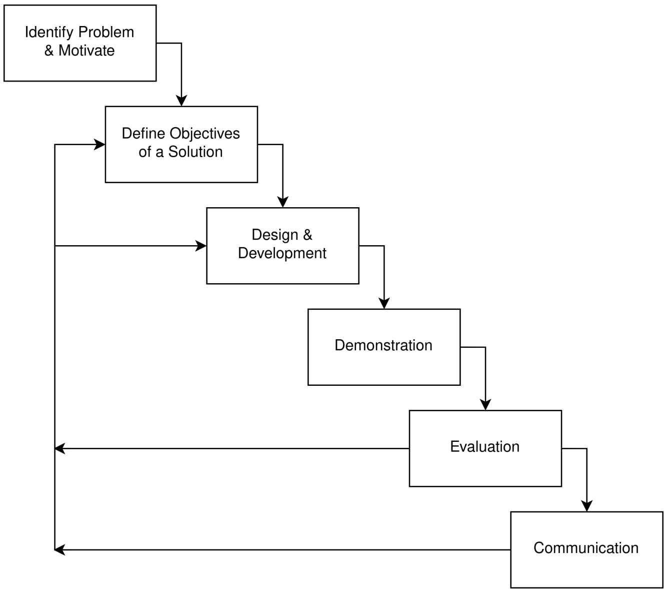


Figure 1.4: Design Science Research Methodology stages [1].

DSRM is a methodological approach focused on the creation and evaluation of artifacts designed to address specific problems. DSRM is structured in several activities or stages as can be seen in Figure 1.4. These stages are: 1) DSRM starts with the identification of a problem; 2) followed by setting clear objectives for a solution; 3) the core of DSRM is the design and development phase, where an artifact is created and iteratively refined; 4) this artifact is then demonstrated in a relevant environment, such as a use case, to illustrate its applicability in solving the identified problem; 5) after that, follows a rigorous evaluation of the artifact, assessing how well it meets the set objectives and addresses the problem; 6) the methodology finishes in the communication stage, where the research process and findings, including the efficacy of the artifact, are documented and shared. For instance, publishing an article in a journal or a congress.

In addition, it is noteworthy that, as can be seen in Figure 1.4, cycling back to earlier stages is a key aspect of the DSRM. Particularly, cycling back from the Evaluation (5) or Communication (6) stages to the Define Objectives (2) or Design & Development (3) stages. The iterative nature of the DSRM is crucial for refining the developed artifact. For instance, if the Evaluation stage reveals that the artifact does not meet its intended aims or effectively solve the identified problem in the first stage of the DSRM, it could be necessary to revisit the Define Objectives (2) stage to reassess and redefine the goals. Similarly, during the Communication stage (6), feedback from peers and users could reveal aspects of the artifact that require further development or refinement, prompting a return to the Design & Development stage.

As for the selection of DSRM, it was grounded in the alignment of SimulateIoT with the specific criteria and guidelines defined in [115] to choose the most suitable DS research methodology for the development of ISs. The main rationale for this selection, based on the aforementioned guidelines, is detailed below.

- Philosophy (Paradigm, Objectives, Domain): DSRM aligns with an objectivist and positivist paradigm, which is appropriate for projects focused on creating software artifacts with measurable outcomes. Since this Ph.D. Thesis aims to develop an IoT simulator that encompasses several software artifacts with measurable outcomes for simulation and validation purposes, DSRM fits suitably in this regard.

- Scope (DS Research Activities): DSRM covers essential DS research activities such as problem identification, defining objectives, design and development, and evaluation. This methodology is ideal for SimulateIoT, ensuring that all critical aspects of the simulator's development, from conceptualization to evaluation, are systematically addressed.

- Output: The primary output of DSRM is the *artifact*, which in the case of this Ph.D. Thesis is SimulateIoT, an MDD-based IoT simulator.
- Practice (Background, User Base, Participants): DSRM's academic orientation suits the research nature of this Ph.D. Thesis. Its widespread use and recognition in academic circles provide a solid foundation for this research.

In summary, the selection of the DSRM for the development of SimulateIoT in this Ph.D. Thesis represents a well-founded choice. This decision is based on thoroughly examining various DS research methodologies and applying the framework proposed by John R. Venable, Jan Pries-Heje, and Richard L. Baskerville in [115], which shows the alignment between this Ph.D. Thesis and the DSRM. Furthermore, the iterative nature of DSRM, as illustrated in Figure 1.4, ensures a flexible and adaptive approach to the development of SimulateIoT, allowing for continuous refinement and improvement based on evaluation and feedback. Moreover, note that this methodology not only guides the systematic development of the IoT simulator but also enhances the academic rigor of the research.

1.5 Summary of Contributions

This section presents a summary of the primary contributions of this Ph.D. Thesis, which are focused on providing insights and answers to the RQs stated in Section 1.3. All these primary contributions can be summarized in SimulateIoT, the main artifact developed during this Ph.D Thesis. This artifact has been developed iteratively following the DSRM. As a result, each version of SimulateIoT together with its particular artifacts and contributions has been communicated through a journal article and several related conferences, leading to the publication of four Journal Citation Report (JCR) journal articles and six conferences (four of them international conferences). Moreover, note that there is another manuscript under development and a national conference paper under review. Therefore, it can be stated that SimulateIoT is founded on four main contributions, i.e. each journal article and its related conferences.

Regarding these four main contributions, they are: A) SimulateIoT [2], a simulator focused on simulate the foundation of IoT systems; B) SimulateIoT-FIWARE [71], a version of SimulateIoT where the target technology is FIWARE; C) SimulateIoT-Mobile [4], an extension of SimulateIoT that includes the concept of IoT mobility to the simulator; D) And a version of SimulateIoT that includes task-scheduling capabilities [5].

To present these main contributions in an organized and systematic manner, this section is divided into four subsections, each one corresponding to one of the four main contributions. Within these subsections, the structure is as follows: 1) *Title* that summarizes the contribution; 2) *Ph.D. Thesis Context* providing an overview of the relevant time frame and objectives related to the thesis development; 3) *Problem Statement* that summarizes the problem identified; 4) *Contributions* that summarizes the particular contributions made to address the problem identified; 5) *RQs Addressed* which links the contributions made to the specific RQs they attempt to address; 6) *Related Artifacts* that describes each developed artifact to implement each contribution, respectively; and lastly 7) *Communications* that offers an overview of the dissemination efforts related to the contributions.

1.5.1 Simulating the Foundation of the IoT from a High Level of Abstraction

Ph.D. Thesis Context

The initial main contribution of this Ph.D. Thesis was developed during its first year, immediately after stating the four RQs that this Ph.D. Thesis aims to address. Consequently, this first contribution involves a preliminary attempt to approach these RQs.

Problem Statement

The complexity and low level of abstraction of IoT simulators lead to significant costs in time, money, and effort due to steep learning curves, and reduced agility in designing, validating, simulating, and improving IoT systems. Moreover, the lack of formal methodologies can lead to ad-hoc development of these systems, an error-prone practice that further increases costs [2].

Contributions

To deal with the problem identified, adopting a higher level of abstraction to address IoT simulations is key. This approach simplifies development by focusing on essential aspects rather than specific details, thereby enhancing the design, validation, and improvement processes of IoT systems, and reducing the need to integrate numerous devices, concepts, and ad-hoc solutions.

To materialize this contribution, MDD techniques, introduced in Section 1.1.1, emerged as a potential solution. Thus, an MDD-based simulator called SimulateIoT, capable of tackling IoT simulations from a high level of abstraction, was developed. This proposal, allows users to graphically design, formally validate, and deploy their IoT system simulations by means of its three main components: 1) a metamodel that captures the main concepts of IoT systems and their interrelationships while allowing their formal modeling and validation; 2) a graphical concrete syntax for graphically and user-friendly modeling IoT systems; and 3) a model-to-text transformation for generating the code of IoT system simulations from their models.

In addition, a methodology that defines each step required to model and validate the IoT system simulations with SimulateIoT has been proposed. Thus, users can use this methodology while designing, validating, and testing their IoT systems through simulations with SimulateIoT.

Furthermore, the validation of SimulateIoT and the proposed methodology has been carried out by conducting two use cases. The first examines a smart building scenario at the University of Extremadura's School of Technology, encompassing six distinct buildings equipped with sensors, actuators, and data analysis processes. The second case study explores an IoT-based irrigation management system aimed at enhancing tomato crop production across ten hectares, incorporating real-time monitoring sensors and irrigation actuators, supported by fog nodes and mosquito brokers.

RQs Addressed

The four RQs defined in Section 1.3 have been addressed to some extent. Through the development of SimulateIoT, the proposed methodology and the two use cases conducted, RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*), RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate* *an IoT system?*) and RQ3 (*To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?*) are partially addressed, while RQ4 (*To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?*) is fully answered. This is because, although this initial version of SimulateIoT involves several technologies, it does not cover the entire IoT ecosystem and its associated concepts. In addition, it mainly includes components exclusively developed for it. Consequently, a positive answer to RQ1, RQ2, and RQ3 is feasible, but only within the boundaries of the IoT context captured by this first version of SimulateIoT. As for RQ4, it is fully addressed since its resolution is independent of the specific IoT context captured. Instead, the proposed methodology relies on the MDD workflow, i.e. modelization, validation, and model-to-text transformations. Therefore, the proposed methodology is applicable to any version of SimulateIoT, as long as this simulator is based on MDD. Detailed discussions regarding these outcomes are reserved for Section 8.1, which is dedicated to the discussion of the results achieved.

Related Artifacts

The key artifacts developed to implement these contributions include SimulateIoT, the proposed methodology for modeling, validating, generating, and simulating IoT systems through SimulateIoT, and the two use cases developed to show the feasibility of both SimulateIoT and the methodology. Additionally, it should be highlighted the three main components that constitute SimulateIoT: 1) the metamodel, which captures the main concepts of IoT systems and their interrelationships while also allowing the formal modeling and validation of these systems; 2) the graphical concrete syntax for graphically and user-friendly modeling IoT systems; and 3) the model-to-text transformations for generating the code of IoT system simulations from the modeled IoT system.

Communications

The contributions achieved in this respect have been disseminated through a JCR journal article and presentations at three international conferences and one national conference. Moreover, there is another journal paper under development. The details of these communications are listed below chronologically.

- **IEEE Acess 2021** [2]: Barriga, J. A., Clemente, P. J., Sosa-Sánchez, E., & Prieto, Á. E. (2021). SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments. IEEE Access, 9, 92531-92552. **JCR Q2 (Computer Science, Information Systems) IF: 3.476**.

- **Jornadas de Investigación Predoctoral en Ingeniería Informática - Doctoral Consortium in Computer Science (JIPII) 2022** [71]: Barriga, J. A., & Pedro, J. (2022). SimulateIoT: A model-driven approach to simulate IoT systems. Predoctoral en Ingenierıa Informática, 29. **International Conference**.

- **Iberian Conference on Information Systems and Technologies (CISTI) 2022** [116]: Barriga, J. A., & Clemente, P. J. (2022, June). Designing and simulating IoT environments by using a model-driven approach. In 2022 17th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1-6). IEEE. **International Conference**.

- **Jornadas de Ciencia e Ingeniería de Servicios (JCIS) 2022** [117]: Barriga Corchero, J. Á., Clemente Martín, P. J., Sosa Sánchez, E., Prieto Ramos, Á. E.: SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments. In: Navarro, E. (ed.) Actas de las XVII Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2022). Sistedes (2022). https://hdl.handle.net/11705/JCIS/2022/006. **National Conference**.

- **International Conference on Service-Oriented Computing (ICSOC) 2022** [118]: Barriga, J. A. (2022, November). Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment. In International Conference on Service-Oriented Computing (pp. 314-319). Cham: Springer Nature Switzerland. **International Conference**.

- **Under development**: Barriga, J. A., Alonso, P., Sosa-Sánchez, E., Perez-Toledano, M. A., & Pedro, J. (2023). Integrating Hardware in the Loop into IoT Systems Simulations: A Model-Driven Development Approach.

1.5.2 Simulating the foundation of the IoT Powered by FIWARE

Ph.D. Thesis Context

The second contribution of this Ph.D. Thesis is aligned with the first, being an extension of it. This contribution was developed after completing the first one and after exploring several possibilities to further address the RQs defined in Section 1.3, taking into consideration the partial answers provided by the first contribution.

Problem Statement

The first approach to SimulateIoT mainly relies on custom-developed components. To comprehensively address RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*) and RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*), the integration of various third-party IoT components into SimulateIoT has been considered. Since third-party components are not specifically tailored for SimulateIoT and can have very different purposes, their successful integration would show that MDD techniques can effectively manage virtually any IoT component and its inherent complexity. This would provide complete answers to both RQ1 and RQ2, as well as further answer RQ3 (*In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?*).

Contributions

To tackle the identified problem, the open-source IoT platform FIWARE was explored. Since this platform offers a wide array of IoT components that have very different purposes, to address the identified problem, the integration of several FIWARE components into SimulateIoT was carried out.

Thus, an extension of SimulateIoT toward the FIWARE IoT platform was developed. This enhancement involves expanding one of the primary components of SimulateIoT: its model-to-text transformations. Consequently, users are able to graphically model IoT systems, validate them conforming to the metamodel, and automatically generate the entire IoT system. In this respect, the new model-to-text transformations incorporate FIWARE as target technology, including components such as the ORION Context Broker, the MongoDB database provided by FIWARE, and the CEP engine Perseo among others. This process also includes generating the custom SimulateIoT components and integrating them with the selected FIWARE components, all performed transparently to the user. Lastly, users can deploy the system and simulate it in the same way as in the first version of SimulateIoT.

Moreover, note that SimulateIoT-FIWARE was validated by carrying out two use cases. Note that these use cases are the same as those included in the first contribution (see Section 1.5.1), differing in that the target technology in this case is the FIWARE platform.

RQs Addressed

This second contribution fully addresses RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*) and RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*). This is because this second contribution successfully integrates diverse third-party IoT components from the FIWARE platform into SimulateIoT. Moreover, it demonstrates the effectiveness of MDD in managing the complexity of these components through two use cases.

Related Artifacts

The artifacts delivered through this contribution are the three main components of SimulateIoT, initially developed in the first contribution: the metamodel, the concrete syntax, and the model-to-text transformations. However, in this second contribution, these artifacts are extended toward the FIWARE IoT platform, thus enabling users to generate and simulate IoT systems with FIWARE components.

Communications

The contributions achieved in this respect have been disseminated through a JCR journal article and a presentation at one national conference. The details of these communications are listed below chronologically.

- **IEEE Acess 2022** [3]: J. A. Barriga, P. J. Clemente, J. Hernández and M. A. Pérez-Toledano, ”SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE,” in IEEE Access, vol. 10, pp. 7800-7822, 2022, doi: 10.1109/ACCESS.2022.3142894. **JCR Q2 (Computer Science, Information Systems) IF: 3.9**.
- **Jornadas de Ingeniería del Software y Bases de Datos (JISBD) 2023** [119]: Barriga Corchero, J. Á., Clemente Martín, P. J., Hernández Núñez, J. M., Pérez Toledano, M. Á.: SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE. In: Durán Toro, A. (ed.) Actas de las XXVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2023). Sistedes (2023). https://hdl.handle.net/11705/JISBD/2023/8183. **National Conference**.

1.5.3 IoT Simulations Toward Mobility Assessments: Mobile-Driven Design and Functionality

Ph.D. Thesis Context

This third contribution was developed during the second year of development of this Ph.D. Thesis. Taking into account the stage of the Ph.D. Thesis development, it was deemed to further verify and validate the answers given to RQs before proceeding further in the development of the Ph.D. Thesis.

Problem Statement

SimulateIoT lacks a crucial feature often present in nowadays IoT systems: the mobility of their devices. Since several users could have to test the mobility of their devices within their IoT systems, and given that this extension would verify and validate further the previous results achieved, it has been considered appropriate to incorporate the mobility concept into SimulateIoT. Note that this contribution verifies the answers already given to RQs as it involves the addition of new IoT concepts to

SimulateIoT. In the context of this contribution, within the boundaries of IoT mobility. Thus, it verifies further the applicability of MDD in managing the complexity of IoT systems with mobile devices (RQ1 *To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*). Moreover, its suitability as a technique to develop a simulation tool to test IoT systems with mobile devices in an agile, user-friendly, and cost-efficient manner, together with the possibility of validating the system design before its simulation, is also further verified (RQ2 *To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*; RQ3 *In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?*).

Contributions

This third contribution aims to integrate the mobility concept, which includes mobile devices and their supporting architecture, into SimulateIoT. This contribution is aligned with the first contribution developed but not with the second one. This is because the second contribution was mainly undertaken to further answer RQ1 and RQ2, for which several components developed for SimulateIoT were replaced by FIWARE components. Conversely, the first contribution that released SimulateIoT was designed to be able to model, validate, and simulate the foundation of IoT environments with components developed by the author of this Ph.D. Thesis. For this reason, it has been considered appropriate to develop the mobility concept on top of SimulateIoT's first approach.

Thus, to develop this contribution, SimulateIoT-Mobile, an extension of SimulateIoT toward IoT mobility, was developed. This enhancement involves expanding the three primary components of SimulateIoT: its metamodel, concrete syntax, and model-to-text transformations. Consequently, with this extension, users can graphically model, validate, and simulate IoT systems incorporating mobile devices and the architecture that supports their mobility. Namely, users can simulate the mobile sensors and actuators, and design the routes that they will perform during the simulation. Note that these devices integrate several functionalities developed in the framework of this contribution to be able to move, such as all the clients and the logic required to interact with the architecture that enables them to move suitably throughout the IoT system. Regarding the architecture to support mobile devices, components such as the *Broker Discovery Service* and the *Topic Discovery Service*, among others, have been included. These components facilitate mobile devices to discover brokers and the range of topics each broker offers, respectively. Thus, enabling mobile devices to request the necessary information to move effectively throughout the entire IoT system while performing their functions.

This extension was validated through two use cases. The first simulates an IoT system for tracking animal movements with GPS. Data collection is facilitated by Fog nodes at strategic locations like lagoons, which then relay data to a Cloud node for behavioral analysis. The second case study simulates a smart Personal Mobility Device (PMD) system in urban settings, enhancing the management and safety of bicycles and electric scooters with sensors for real-time monitoring and actuators for user alerts on issues. Fog nodes across the city collect and forward PMD data to Cloud nodes for analysis and enable automatic notifications regarding device status, ensuring efficiency and safety.

In short, this contribution is focused on simulating IoT mobility and its related concepts, with the aim of furnishing users with valuable insights regarding the design of their IoT systems with mobile devices. Additionally, this extension also serves to further verify and validate the answers provided by the preceding contributions to the defined RQs.

RQs Addressed

This contribution is related to all RQs, as this extension involves the addition to SimulateIoT of several concepts and components related to the IoT. Namely, to the IoT mobility. While this contribution does not conclusively answer RQ3, the sole RQ that is not completely answered to this point, it further answers all RQs, verifying and validating the research outcomes achieved by the previous contributions.

Related Artifacts

The main artifacts delivered through this contribution are the three primary components of SimulateIoT, initially developed in the first contribution: the metamodel, the concrete syntax, and the model-to-text transformations. However, in this third contribution, these artifacts are extended toward IoT mobility, thus enabling users to include mobile devices together with the mobility-supporting architecture in their simulations.

Communications

The contributions achieved have been disseminated through a journal article. Moreover, there is another paper related to this contribution already submitted to a conference and under review. The details of these communications are shown below.

- **Pervasive and Mobile Computing (PMC) 2023** [4]: Barriga, J. A., Clemente, P. J., Pérez-Toledano, M. A., Jurado-Málaga, E., Hernández, J. (2023). Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile. Pervasive and Mobile Computing, 89, 101751. **JCR Q2 (Computer Science, Information Systems) IF: 4.3**.

- **Jornadas de Ingeniería del Software y Bases de Datos (JISBD) 2024 (Under review)**: Barriga, J. A., Clemente, P. J., Pérez-Toledano, M. A., Jurado-Málaga, E., Hernández, J., Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile*. Sistedes (2024). **National Conference**.

1.5.4 IoT Simulations Toward Task-Scheduling Assessments: Task-Driven Design and Functionality

Ph.D. Thesis Context

This contribution was developed in the third year of the Ph.D. Thesis, following the complete addressing and validation of RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*), RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*), and RQ4 (*To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?*) through the three prior contributions. RQ3 (*In what measure are MDD* *techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?*) remains the only RQ not fully addressed at this stage. Therefore, this contribution is focused on answering RQ3 further while simultaneously expanding the simulation capabilities of SimulateIoT.

Problem Statement

RQ3 is the only RQ that has not been comprehensively addressed. Specifically, the question *To what extent are MDD techniques effective in developing IoT simulation tools that possess adaptive integration capabilities?* related to RQ3 has to be answered to fully address this RQ. So, this contribution is focused on resolving this aspect of RQ3.

In this respect, it has been considered to extend SimulateIoT toward task scheduling. Task scheduling is a well-known and widespread technique to optimize task execution within distributed systems like IoT systems. However, there is a lack of IoT simulators that include modeling task scheduling in line with current IoT systems that integrate the Cloud-To-Things Continuum paradigm. Additionally, existing task-scheduling simulators do not easily allow users to test their own task-scheduling algorithms, a gap that this contribution aims to fill while answering RQ3.

Contributions

This fourth contribution integrates task scheduling capabilities into SimulateIoT for simulation purposes. To implement it, SimulateIoT was extended toward several task-scheduling concepts, which involved expanding the three primary components of SimulateIoT: its metamodel, concrete syntax, and model-to-text transformations. Therefore, with this extension, users can graphically model, validate, and simulate IoT systems with task scheduling capabilities. Namely, users can model: 1) *Tasks* by means of *Workflows* [120, 121]; 2) the nodes that will generate these tasks during simulation execution, such as *TaskNodes* and *TaskApps*; 3) the *Task Scheduler* that will schedule these tasks; 4) the task processors of the system, which will process the scheduled tasks. Often they will be Fog and Cloud nodes, although Mist/Edge nodes such as the *Task Nodes* can also incorporate task processing capabilities; 5) several federations between nodes; and 6) relevant task-scheduling networking aspects such as the bandwidth and delay between the links of each federated node.

So, within this task-scheduling environment, as users can integrate their task-scheduling algorithms in the *Task Scheduler* node, users can test their own task-scheduling proposals, assess and validate them, and compare them with other existing in the literature.

Note that this contribution has been validated by carrying out two use cases.

RQs Addressed

From the perspective of RQs, the most relevant component released in this contribution is the *Task Scheduler*. This is because it provides the required answers to completely address RQ3. In this respect, the *Task Scheduler* component is capable of integrating users' task-scheduling algorithms and using them within the defined simulations. Thus, users can use SimulateIoT to asses their own task-scheduling proposals, measure their performance, compare them with the existing ones in the literature, etc. So, by developing this contribution, it is feasible to fully answer RQ3 by responding to the question: *To what extent are MDD techniques effective in developing IoT simulation tools that possess adaptive integration capabilities?* A part of RQ3 that previous contributions lacked an answer.

Moreover, note that as task scheduling includes new IoT concepts, functionalities, and components to SimulateIoT, the rest of the RQs are also further validated.

Related Artifacts

The main artifacts delivered through this contribution are the three primary components of SimulateIoT: the metamodel, the concrete syntax, and the model-to-text transformations. However, in this fourth contribution, these artifacts are extended toward IoT task scheduling, thus enabling users to test their task-scheduling algorithms within a wide IoT ecosystem by means of simulations.

Communications

This contribution has been disseminated through a conference paper and a journal article. The details of these communications are shown below chronologically.

- **International Workshop on Model-Driven Engineering for Smart IoT Systems (MeSS) 2022** [122]: Barriga, J. A., &

Clemente, P. J. (2022). SimulateIoT-Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations. **International Conference**.

- **Journal of Object Technology (JOT) 2023** [5]: Barriga, J. A., Chaves-González, J. M., Barriga, A., Alonso, P., & Clemente, P. J. Simulate IoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments. **JCR Q4 (Computer Science, Software Engineering) IF: 0.8**.

1.5.5 Collaboration: SimulateIoT toward Big Data assessments

Ph.D. Thesis Context

In the concluding year of this Ph.D. Thesis, a collaborative stay was undertaken with a research group at the *Universidade de Aveiro* in Aveiro, Portugal. This research stay was focused on extending SimulateIoT toward Big Data systems.

It is important to acknowledge that this contribution is currently unpublished as it remains under development. Specifically, the completion of a use case and the writing of the corresponding paper are still pending. In this respect, note that due to the absence of definitive results at this stage, this contribution is mentioned in this summary but will not be included in the chapter dedicated to presenting the results of this Ph.D. Thesis (Chapter 2).

Problem Statement

Some IoT simulators are designed to simulate the Big Data segment of an IoT system, which is responsible for providing storage and query services [123, 104, 124]. These simulators primarily assess the performance of these services in the context of the IoT. For instance, by measuring the time that these systems take to process a particular request in a specific timeframe, when various devices may be executing concurrent requests.

Despite their value in analyzing different aspects of Big Data systems in IoT environments, these simulators have several limitations. Primarily, they tend to assess the Hadoop File System [123, 104], neglecting the need to test a more diverse range of Big Data systems. In addition, these simulators often lack integration adaptability. For instance, users could face challenges in integrating their custom requests into simulations. Thus, hindering a more accurate assessment of users' system performance. Besides, these simulators often focus on the Big Data system and neglect the rest of the IoT system, thus capturing a small domain regarding the IoT [124]. Note that the IoT domain is critical in these kinds of simulators, as it enables an accurate assessment of the IoT environment's influence on the Big Data system. Moreover, these simulators often adopt a low level of abstraction in their simulation approach, which can lead to increased time, cost, and effort in the system testing process.

In short, the predominant focus on the Hadoop File System, coupled with the identified shortcomings in integration adaptability, IoT scope covered, and abstraction level used to tackle simulations, underscores the need for improvements in the design and functionality of these kinds of simulators.

Contributions

This contribution is based on the integration of Big Data capabilities into SimulateIoT. For this purpose, SimulateIoT has been extended toward several Big Data concepts, which involved expanding the three primary components of SimulateIoT: its metamodel, concrete syntax, and model-to-text transformations. Therefore, with this extension, users can graphically model, validate, generate, and simulate IoT systems with Big Data capabilities.

This simulator offers users several capabilities, including: 1) Configuring and deploying a Big Data system based on MongoDB, covering elements such as *shards*, *replicaset* nodes, *query router* nodes, *config server* nodes, and the MongoDB schema with its databases and collections; 2) Simulating workloads, which constitute the set of requests generated by devices and applications during the simulation. These workloads can incorporate user-defined requests, thus enhancing the customization and adaptability of simulations to meet specific user requirements. Note that these workloads will be sent to the defined database system for their processing; 3) Generating enhanced synthetic data for sensors, since the simulator is primarily data-centric. This is because the key assessment that can be carried out with this tool is to analyze the performance of the Big Data system regarding request execution. So, as requests involve data, and as more complex data enables more complex and varied requests, the synthetic data generation has been improved; 4) data aggregators that aggregate published data on user-selected topics and send it to the database for their storing.

These functionalities represent the main aspects available for simulation. However, the simulator also allows extensive customization through various configurable parameters, with the aim of mirroring real-world systems and obtaining results applicable in practical scenarios. In addition, the simulator incorporates features such as a real-time simulation analyzer, which generates multiple graphs that provide valuable information such as each request generated and processed together with their processing time among other insights.

Thereby, this comprehensive set of features and customizable options make the simulator a versatile tool for accurately modeling and analyzing Big Data systems in the context of IoT, avoiding the shortcomings identified in the problem statement section of this contribution.

RQs Addressed

This contribution addresses the four RQs defined. Regarding RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*), this contribution extends the simulator toward Big Data systems within the scope of the IoT. Thus, verifying the suitability of the application of MDD to tackle effectively the complexity of IoT systems, in this context, with Big Data capabilities. Concerning RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*), this contribution extends the model-to-text transformations of SimulateIoT, generating the entire architecture of a Big Data system and the IoT environment that interacts with this system. Thus, validating that MDD is adequately suited for generating the simulation code necessary to simulate IoT systems, in this respect, with Big Data capabilities. As for RQ3 (*In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?*), this simulator provides high adaptability in terms of integrating user-defined requests and user-defined sensor data generation. Moreover, as with the other releases of SimulateIoT, this contribution integrates a concrete graphical syntax to facilitate system modeling for users. It also offers the possibility to validate models according to the metamodel. Furthermore, allows for testing these systems in a cost-efficient manner. In this respect, the costs associated with these simulations are the efforts made to design the system simulation, mitigated by the low learning curve of SimulateIoT, and the energy consumption that the simulation could require to be executed. Lastly, regarding (*To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?*), the methodology developed in the first contribution has been further validated by using it to carry out the use case conducted in this contribution.

Thereby, this final contribution has been designed to both help users test their Big Data systems within their IoT environments and to finally verify and validate the answers given to each RQ. Thus, conclusively answering them.

Related Artifacts

The main artifacts delivered through this contribution are the three primary components of SimulateIoT: the metamodel, the concrete syntax, and the model-to-text transformations. However, in this last contribution, these artifacts are extended toward Big Data systems, thus enabling users to test their Big Data systems within a wide IoT ecosystem by means of simulations.

Communications

Since this contribution is still under development, there are no publications yet.

1.6 Structure of the Thesis

This Ph.D. Thesis is organized into four main sections. The initial section provides a summary of the thesis, including Chapter 1, which serves as the Introduction, and Chapter 2, which presents the Results. The second section compiles the key research papers that underpin the foundation of this Ph.D. Thesis, including Chapter 3, which presents an overview of these publications, and Chapters 4, 5, 6, and 7, which include the papers that comprise the compendium of publications. The third section is dedicated to the Conclusions, summarizing the insights and implications of the research. The final section encompasses the Appendices, offering supplementary material supporting the research findings.

Chapter 2

Results

"The Road goes ever on and on down from the door where it began. Now far ahead the road has gone, and I must follow, if I can, pursuing it with eager feet, until it joins some larger way where many paths and errands meet. And whither then? I cannot say."

The Fellowship of the Ring (1954)
Tolkien, J. R. R.

This section describes the main contributions of this PhD thesis, which aim to provide insights and answers to the RQs specified in Section 1.3. As mentioned in Section 1.5, the primary artifact and contribution of this Ph.D. thesis, SimulateIoT, is founded on four main contributions. This section elaborates on these four contributions. To this end, first, the problem statement on which each contribution is based, previously outlined in Section 1.5, is revisited. Subsequently, the contributions, also summarized in Section 1.5, are presented and elaborated in detail.

2.1 Simulating the Foundation of the IoT from a High Level of Abstraction

There is a wide diversity among IoT simulators [125]. Some simulators are specialized in simulating specific segments of an IoT system, such as the Cloud Layer or the Fog Layer, while others have been designed to conduct holistic simulations, encompassing components related to each computing layer within an IoT system. Regardless of what layer or set of layers a simulator focuses on, the types of simulations that can be performed, and therefore the insights that can be derived, vary significantly [125]. For instance, certain cloud-focused IoT simulators allow for the simulation of specific node types, like 'Data Center' nodes [103], while other cloud-focused IoT simulators do not include this type of node.

This diversity among simulators stems from the complexity of IoT systems. This complexity makes it unfeasible to capture the heterogeneous and vast array of technologies, such as devices and communication protocols, in a single IoT simulator. For this reason, holistic simulations are too complex to handle by a single simulator, and simulators face difficulties in providing users with a wide spectrum of knowledge and insights. Therefore, users often have to use several IoT simulators in order to comprehensively test their IoT systems [125], which increases the learning curve that users have to overcome to test their systems.

Given this scenario, there is an opportunity to elevate further the level of abstraction at which IoT simulations are addressed. This elevation would enable simulators to focus on the core concepts of IoT systems rather than on low-level details like the kind of nodes to include regarding IoT computing layers. For instance, high-level abstract simulators should include the generic concept of a *Cloud node* and the pertinent parameters associated with these nodes (which could play a crucial role during simulations, such as their computing power), instead of the myriad of existing cloud node types, like the aforementioned 'Data Center' nodes. Likewise, these high-level abstract simulators would allow users to focus on what actually matters instead of on low-level aspects of their specific IoT systems. Thus, reducing the learning curve to use the simulator.

So, elevating further the level of abstraction at which IoT simulations are tackled avoids capturing the wide spectrum of technologies that comprise

IoT systems. Thereby, making it feasible to perform holistic simulations, or at least, allowing for more comprehensive simulations. Moreover, it also helps users to design and test their IoT systems without concern for low-level details, which decreases the learning curve of the simulator.

On the other hand, another problem identified is the lack of methodologies documented in the literature for designing, validating, and simulating IoT systems, which can lead to errors and inaccurate simulation outputs.

In this context, MDD techniques, introduced in Section 1.1.1, emerged as a potential solution. Thus, an MDD-based simulator called SimulateIoT, capable of tackling IoT simulations from a high level of abstraction, was developed. This tool allows users to graphically design, validate, and deploy their IoT system simulations by means of its three main components: 1) a metamodel that captures the main concepts of IoT systems and their interrelationships, while also allowing the modeling and validation of these systems; 2) a graphical concrete syntax for graphically and user-friendly modeling these IoT systems; and 3) a model-to-text transformation for generating the code of the specific IoT system simulation based on the modeled IoT system.

The metamodel of SimulateIoT is shown in Figure 2.1. In this Figure, it can be observed some of the high-level concepts related to IoT systems that have been captured, such as Sensors, Actuators, Fog and Cloud nodes, etc. that are common in any IoT system. The explanation of each of these concepts can be found in the paper included in Chapter 4. Specifically, in Section IV - subsection A, of this paper.

The concrete syntax developed to graphically assist users in modeling IoT system simulations conform to the proposed metamodel, is illustrated in Figure 2.2. This tool, which can be integrated into Eclipse IDEs [126] via plugins, provides a palette featuring all the modelable elements, along with a canvas where users can design their simulations. Further details about this component can be found in the paper included in Chapter 4. Specifically, in Section IV - subsection B, of this paper.

Once an IoT system model is designed and validated, its code can be generated. Figure 2.3 shows a generic architecture regarding the components and their interactions that can be generated through SimulateIoT model-to-text transformations.

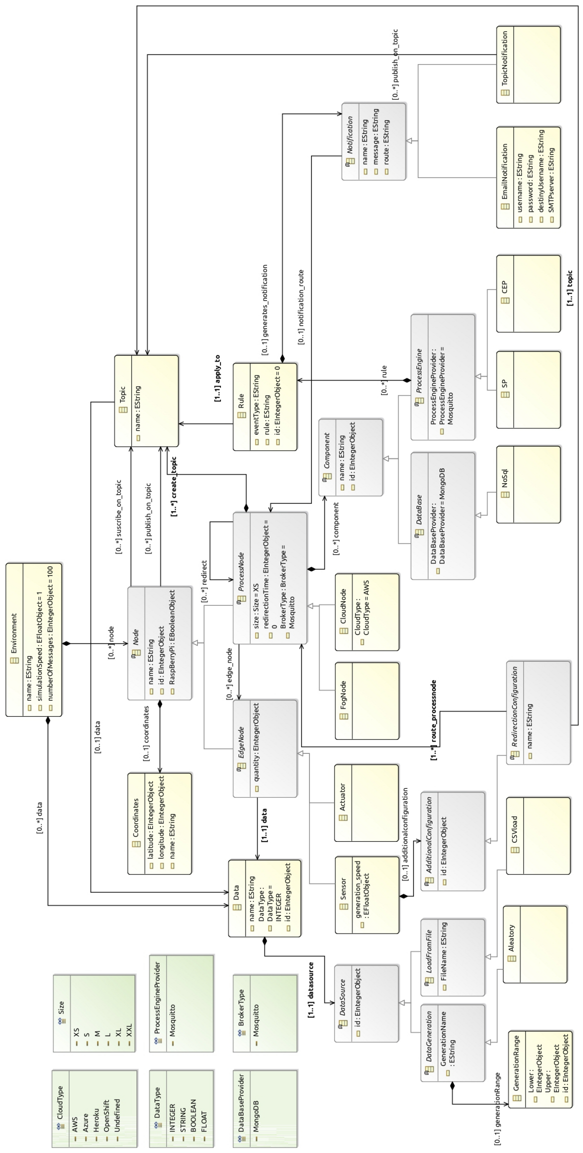
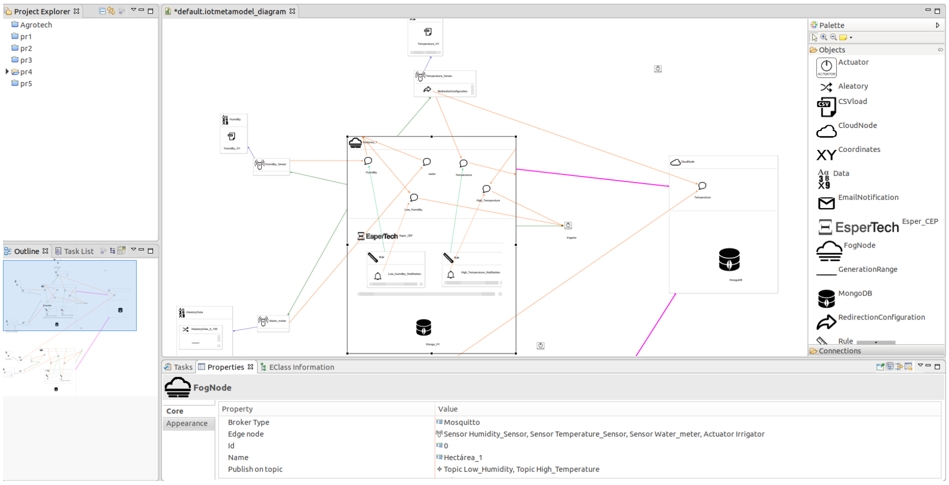


Figure 2.1: SimulateIoT metamodel. Figure source [2].



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 2.2: SimulateIoT concrete syntax. Figure source [2].

Note that the architecture shown in Figure 2.3 is close to a real IoT system rather than a simulated one. Each component is wrapped in a docker container [127], such as a MongoDB Database [128], a Mosquitto Broker [129], and CEP engines such as EsperTech-based ones [130], among other components. This is because SimulateIoT relies on real components to perform simulations. It automatically integrates them into simulations depending on the systems modeled by users. The rationale for this decision is mostly founded on RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*). This RQ seeks to ascertain if MDD is suitable for managing the complexity of IoT systems. Managing the complexity of IoT systems involves both capturing the domain of IoT systems and handling their technological heterogeneity. By capturing the domain of IoT systems, MDD enables users to model IoT systems. By handling their technological heterogeneity, MDD enables users to automatically generate the code of each IoT component that has to be included in the system, the configuration files required for each component, the deployment script to orchestrate the deployment of the system, etc. Therefore, truly handling the

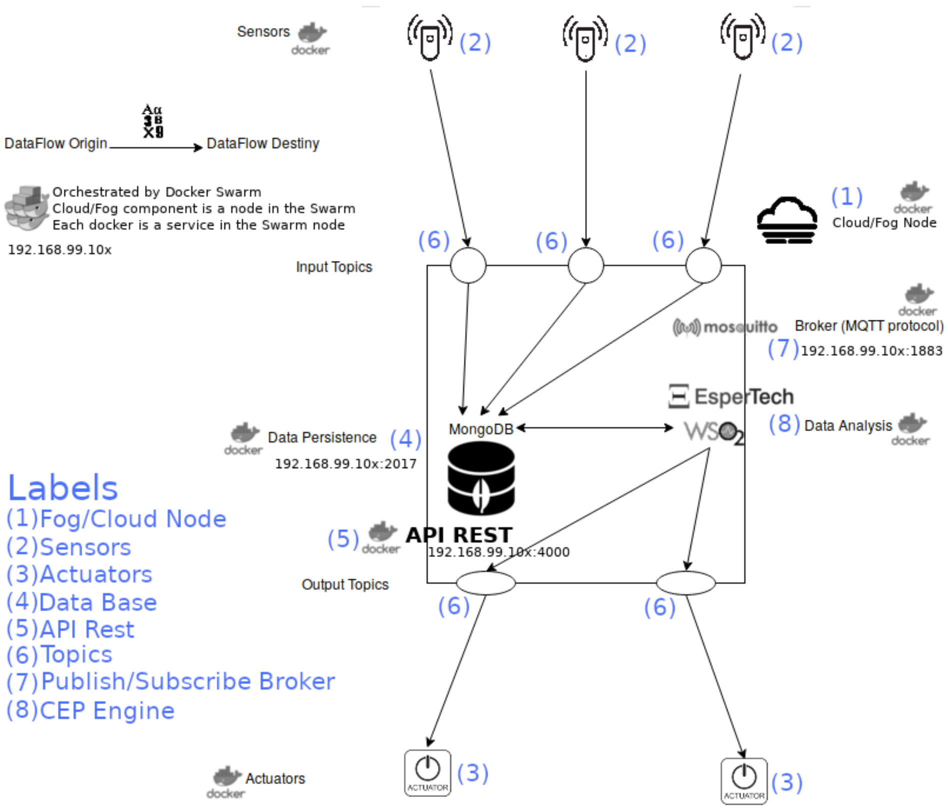


Figure 2.3: Generic IoT architecture generated through SimulateIoT's model-to-text transformations. Figure source [2].

complexity of IoT systems, from modeling to deploying and orchestrating them. Nevertheless, it is noteworthy that although SimulateIoT performs simulations relying on real components, there are also several simulation processes, such as the sensors' synthetic data generation among others.

On the other hand, the second contribution is the development of a methodology (see Figure 2.4) that defines each step required to model, validate, and simulate IoT system simulations with SimulateIoT. Thus, users can use this methodology while designing, validating, and testing

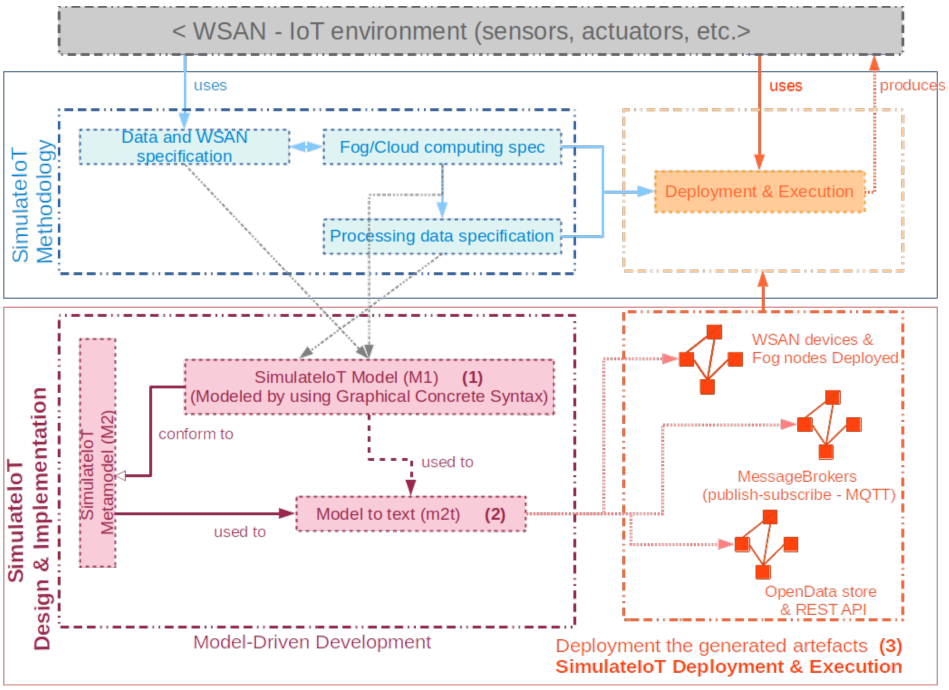

< WSAN - IoT environment (sensors, actuators, etc.>
uses
uses
produces
SimulateIoT Methodology
Data and WSAN specification
Fog/Cloud computing spec
Processing data specification
Deployment & Execution
SimulateIoT Design & Implementation
SimulateIoT Metamodel (M2)
SimulateIoT Model (M1) (1)
(Modeled by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

Figure 2.4: SimulateIoT methodology. Figure source [2].

their IoT systems through simulations with SimulateIoT. In Figure 2.4, the proposed methodology, named *SimulateIoT methodology* (depicted in blue), is displayed along with its integration into the SimulateIoT Design & Implementation stages. These stages essentially mirror the process commonly employed by any tool based on MDD, encompassing modeling, validation, and model-to-text transformations (code generation from models). Additionally, the figure illustrates the outcome of applying this methodology, which results in the deployment of the simulation.

Finally, the third contribution is the validation of SimulateIoT and the proposed methodology by conducting two use cases. The first case study focuses on the simulation of a smart building, specifically, 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 been furnished with a set of sensors, actuators, and analysis information processes. The second case study focuses on simulating an IoT system for managing irrigation and weather data to improve crop production. This case study includes sensors distributed over ten hectares of tomatoes that are monitored in real time. Moreover, actuators to control the irrigation of these tomatoes, together with elements to allow the operation of these sensors, such as fog nodes or mosquito brokers, are also included. The paper included in Chapter 3, delves into the inner details of these use cases in Sections V - A and B respectively.

Before concluding this section, it is important to acknowledge that these contributions have limitations since this first approach to SimulateIoT is focused on simulating some of the key components of IoT systems by applying MDD techniques, such as the simulation of cloud and fog nodes, or sensors and actuators. However, it does not allow the simulation of other key concepts such as the federation between nodes or their mobility. This is because this first approach is mainly focused on initially addressing the RQs stated in Section 1.3, rather than on providing a final IoT simulator.

2.2 Simulating the Foundation of the IoT Powered by FIWARE

Given the partial answers given to RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*) and RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*) by the first contribution, this second contribution primarily aims to answer these two RQs further. In this respect, it is noteworthy that the initial version of SimulateIoT primarily encompasses custom components specifically designed for it. Therefore, to address RQ1 and RQ2 further, the integration of third-party IoT components into SimulateIoT was considered. Third-party components are not specifically designed for SimulateIoT or tools based on MDD. Moreover, third-party components are heterogeneous and can vary significantly, each serving distinct functions within an IoT system. Consequently, if these components could be successfully incorporated into SimulateIoT, it would validate the findings and solutions related to RQ1 and RQ2 achieved through the first contribution. Thus, leading to a conclusive answer for these two RQs.

For this purpose, the open-source IoT platform FIWARE was chosen as the target technology. This platform provides a comprehensive array of IoT components through its catalog that can be integrated into SimulateIoT, along with its paradigm for managing IoT systems, which is centered on context awareness, as elaborated in Section 1.1.5. This approach is not only suitable for addressing the previously stated problem but also enhances the simulation capabilities offered by SimulateIoT to the community. Namely, to the best of our knowledge, there are currently no MDD-based IoT simulators specifically designed for simulating and testing IoT systems that use FIWARE.

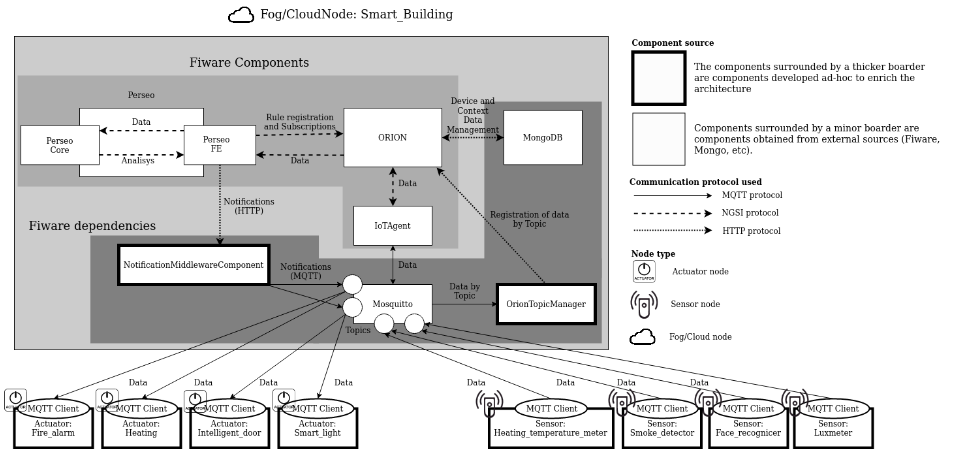

Fog/CloudNode: Smart_Building
Fiware Components
Perseo
Perseo Core
Perseo FE
Data
Analisys
Rule registration and Subscriptions
Data
ORION
Device and Context Data Management
MongoDB
Data
IoTAgent
Notifications (HTTP)
Registration of data by Topic
Fiware dependencies
NotificationMiddlewareComponent
Notifications (MQTT)
Data
Mosquitto
Data by Topic
OrionTopicManager
Topics
Data
MQTT Client
Actuator: Fire_alarm
Actuator: Heating
Actuator: Intelligent_door
Actuator: Smart_light
Sensor: Heating_temperature_meter
Sensor: Smoke_detector
Sensor: Face_recognicer
Sensor: Luxmeter
Component source
The components surrounded by a thicker boarder are components developed ad-hoc to enrich the architecture
Components surrounded by a minor boarder are components obtained from external sources (Fiware, Mongo, etc).
Communication protocol used
MQTT protocol
NGSI protocol
HTTP protocol
Node type
Actuator node
Sensor node
Fog/Cloud node

Figure 2.5: Generic IoT system architecture generated and deployed using SimulateIoT-FIWARE. Figure source [3].

Thus, SimulateIoT has been extended towards FIWARE. For this purpose, only one of its three main components of SimulateIoT has been extended, the model-to-text transformations. The metamodel and the concrete syntax remain unchanged from the first version of SimulateIoT. This is because the metamodel of SimulateIoT-FIWARE does not introduce new IoT concepts. Therefore, the systems that can be modeled with SimulateIoT-FIWARE are identical to those with the original SimulateIoT. However, in this version, all components that were developed for the initial version of SimulateIoT and could be substituted with FIWARE platform components, have been replaced. Moreover, those FIWARE components required in any IoT system powered by FIWARE, such as the core component of FIWARE, the ORION Context Broker, that orchestrates every FIWARE component, have been also included. Furthermore, some components to integrate FIWARE components with SimulateIoT components, have been included likewise. Thus, when generating the code and deploying simulations in this second approach, several FIWARE components are involved. Figure 2.5 illustrates a generic IoT system deployed with SimulateIoT-FIWARE. The FIWARE components included are the *ORION Context Broker*, the *MongoDB* database provided by FIWARE, the *CEP engine Perseo*, and the *IoTAgent*. They are addressed below.

- *ORION Context Broker* [131]: This is the core component of the FIWARE platform. It manages the entire lifecycle of context information including updates, queries, registrations, and subscriptions. It allows the system to perform updates and manage changes in the state of connected entities (such as IoT devices, user-defined entities, etc.), making this information available for other components or third-party applications.

- *MongoDB Database Provided by FIWARE* [132]: MongoDB, in the context of FIWARE, is used as a database to store context information. It's a NoSQL database, which means it can handle large volumes of data and is highly scalable. In FIWARE, MongoDB is often used to store the data managed by the Orion Context Broker, providing a robust and efficient way to handle the data generated by IoT devices and other sources.

- *CEP Engine Perseo* [133]: The Complex Event Processing Engine Perseo is a rule-based system designed to support the definition and notification of events based on specific patterns detected in the data managed by the Orion Context Broker. Essentially, it allows actuators, by notifying them, to perform real-time responses to complex sequences of events, which is crucial for dynamic and responsive IoT applications.

- *IoTAgent* [134]: The IoT Agents in FIWARE are responsible for the communication between IoT devices and the Orion Context Broker. They translate the device-specific protocols into the standard NGSI (Next Generation Service Interfaces) used by the Orion Context Broker. This means that devices using different communication protocols can be integrated into the FIWARE ecosystem, ensuring interoperability and seamless data flow between devices and the Context Broker.

In addition to these components, some components have been developed in order to increase the functionality of the included FIWARE components, such as the *NotificationMiddlewareComponent*, which adapts the HTTP notification of the component *CEP Perseo* to MQTT, and the *ORION-TopicManager* that saves all the published data in each topic in a collection of MongoDB. Note that the standard version of FIWARE saves the data of each sensor in distinct collections.

On the other hand, Table 2.1 presents a mapping between the components of SimulateIoT and the SimulateIoT-FIWARE. This mapping shows the relationship between the metamodel concepts and each component, indicating which components from SimulateIoT have been replaced, by which FIWARE components have been replaced, and which have been preserved. Thus, regarding the *ProcessorNodes*, which can be fog and cloud nodes, the Mosquitto broker and the MQTT client, both deployed on these computing nodes, have been replaced for those provided by FIWARE. Moreover, the Orion Context Broker and the IoT Agent have been included and deployed on these fog and cloud nodes, as they are enabler components for FIWARE allowing it to be integrated with the rest of the system. Note that further description about these components can be found in Section 1.1.5, as well as for the rest of the FIWARE components involved in this Section. Regarding the DataBase, the MongoDB has been replaced with the MongoDB provided by FIWARE. In addition, the MongoDB client developed in the prior version of SimulateIoT has been replaced by the ORION Context Broker, which has been designed to manage all the data published in topics and store them suitably in MongoDB. Lastly, the CEP engine developed with EsperTech

Metamodel Element	***SimulateIoT* Components**	**Description**	**FIWARE Components**	**Description**
ProcessNode	Mosquitto	MQTT Broker	*Mosquitto*	MQTT Broker
			MQTT client	Publish/Subscribe on topics
	MQTT Client (internal component)	Publish/subscribe on topics	*Orion* Context Broker	Device and Context data management
			IoTAgent-Json	Bridge between MQTT-Json and NGSI
Component Database	MongoDB client	MongoDB management	*Orion* Context Broker	*Orion* has a client to interact with MongoDB
	MongoDB	NoSql Database	MongoDB	NoSql Database. A dependency of *Orion* Context Broker. Due to the above fact, the *ComponentDatabase* is no longer an optional component
Component Process Engine	CEP Engine	Apply rules to topics	Perseo	Perseo-FrontEnd: Subscribe to *Orion* context data and Publish notifications (HTTP), Perseo-Core: Apply rules to *Orion* context data
	MQTT client	Subscribe on topics, Publish notifications on topics		

Table 2.1: Relationships among the main metamodel elements with the main target components. Table source [3].

for SimulateIoT has been also replaced by Perseo, a CEP Engine provided by FIWARE. Note that, as mentioned before in this Section, to adapt the behavior of Perseo regarding notifications, as well as how the Orion Context Broker manages and stores data, the *NotificationMiddlewareComponent* and the *OrionTopic Manager* components have been developed and integrated, respectively, in the resulting SimulateIoT-FIWARE architecture, as can be seen in Figure 2.5.

Finally, this contribution was validated by carrying out two use cases. Note that these use cases were the same as those included in the first contribution (see Section 2.1), differing in that the target technology in this case is the FIWARE platform. The paper included in Chapter 4, delves into the inner details of these use cases in Sections VI - A and B respectively.

2.3 IoT Simulations Towards Mobility Assessments: Mobile-Driven Design and Functionality

Having comprehensively addressed RQ1 (*To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?*), RQ2 (*To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?*), RQ4 (*To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?*), and partially RQ3 (*In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?*) through the first and the second contributions, it was deemed appealing to further verify and validate the answers given to these RQs. Thus, ensuring the validity of the results achieved and being able to focus on comprehensively answering RQ3 in the fourth contribution. Note that, increasing the simulation features of SimulateIoT, implies the addition of several and diverse new IoT concepts, components, and functions. Thereby, verifying and validating further the results achieved to answer the RQs. This is especially appealing at this stage, given SimulateIoT's capabilities to simulate the foundation of IoT systems, which makes it possible to target more specific simulation aspects by leveraging these foundational elements. In this respect, taking into account that SimulateIoT lacks a crucial feature often present in current IoT systems, the mobility of their devices, it was considered to extend it toward IoT mobility and its related concepts.

To develop this contribution, the three primary components of SimulateIoT were extended: the metamodel, the concrete syntax, and the model-to-text transformations. Consequently, with this extension, users can graphically model, validate, and simulate IoT systems incorporating mobile devices and the architecture that supports their mobility.

The extension made to the metamodel is illustrated in Figure 2.6. Note that for the sake of clarity, this figure focuses exclusively on the new classes and relationships included on top of the first SimulateIoT metamodel. This extension includes: 1) the ability to model mobile devices capable of publishing and subscribing to topics; 2) the capability to set specific parameters for these devices. Notably, this includes the option to specify a buffer for data storage during periods of disconnection, as well as the signal gain power of these devices, which refers to each device's ability to detect and amplify coverage signals; 3) the ability to model the route that each of these devices will follow during simulation execution; 4) the capability of configuring mobility-relevant parameters for fog and cloud nodes, such as signal coverage power. This parameter pertains to the signaling power of gateways associated with fog and cloud nodes, which enables devices to connect to them and with the rest of the system. 5) the capability of modeling features related to the security of the mobile IoT system, such as a token verification service to prevent unauthorized devices from connecting to the system.

Regarding the concrete syntax, it remains the same tool as in the two previous contributions, but with the required graphical elements to represent the included mobile concepts.

On the other hand, the model-to-text transformations have been also extended. These transformations enable users to generate IoT systems with mobile devices that users can model with the SimulateIoT-Mobile metamodel. Moreover, these transformations also generate the mobility-supporting architecture. In essence, this architecture is based on several components deployed on fog and cloud nodes, capable of supporting the

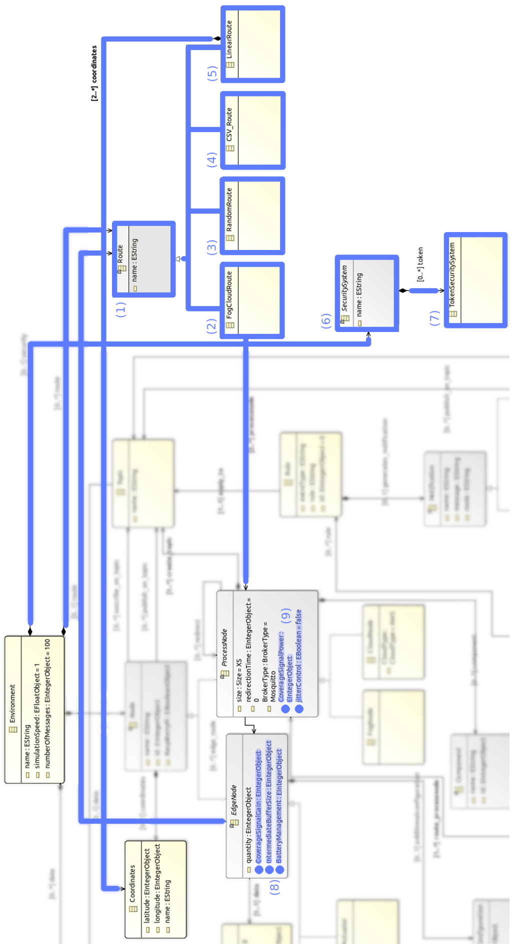

LinearRoute
CSV_Route
RandomRoute
FogCloudRoute
Route
name : EString
SecurityTystem
name : EString
TokenSecuritySystem
[0..*] token
[2..*] coordinates
Environment
name : EString
simulationSpeed : EFloatObject = 1
numberOfMessages : EIntegerObject = 100
ProcessNode
size : Size = XS
redirectionTime : EIntegerObject = 0
BrokerType : BrokerType = Mosquitto
CoverageSignalPower : EIntegerObject
JitterControl : EBoolean = false
EdgeNode
quantity : EIntegerObject
CoverageSignalGain : EIntegerObject
IntermediateBufferSize : EIntegerObject
BatteryManagement : EIntegerObject
Coordinates
latitude : EIntegerObject
longitude : EIntegerObject
name : EString
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)

Figure 2.6: Excerpt of the SimulateIoT-Mobile metamodel focusing on the new classes and relationships included in the metamodel through this contribution. Figure source [4].

movement of mobile devices. For the sake of clarity, firstly, this architecture and its related components are addressed. Subsequently, mobile devices are described.

Figure 2.7 illustrates the components and modules that can comprise

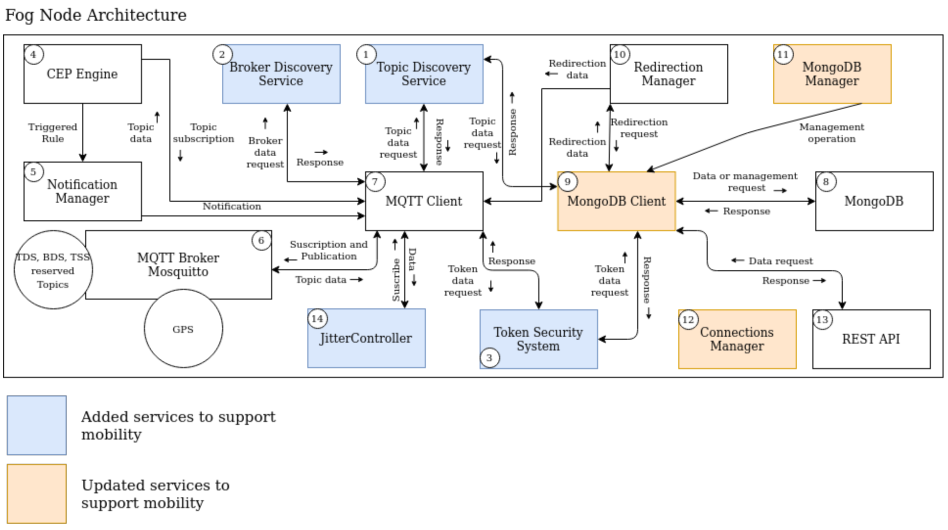


Figure 2.7: Software architecture of a Fog/Cloud node generated. Figure source [4].

the fog and cloud nodes of SimulateIoT-Mobile. Among these components, it can be observed that some of them are highlighted in blue or orange, while others are not highlighted. The components highlighted in blue are those components generated by the extended model-to-text transformations. They have been exclusively developed for this contribution and primarily focus on supporting mobile devices' movement. Among these components, it can be observed: A) *The Broker Discovery Service*, which provides mobile devices with information related to the location of the brokers of the IoT system. With this information, mobile devices know to which brokers they could connect during their routes throughout the system; B) The *Topic Discovery Service*, which is a complementary module to the *Broker Discovery Service* since it provides users with information related to the topics offered by each broker of the system. Thus, mobile devices know whether a specific broker would allow them to publish or subscribe to a specific topic. Information that can be used by mobile devices to determine whether to connect or not to a broker; C) The *Token Security System*, which ensures that untrustworthy devices can not connect to the system.

This security service provides a token to each device at the beginning of the simulation. Thus, each time a device has to publish or subscribe to a broker, it first requests to the *Token Security System*, which will verify its token. So, if the token is valid, the device can continue performing its operations. Conversely, if the token is not valid, the device is disconnected from the system. Note that during simulations, these tokens change over time, and devices are provided by this security system with new tokens; D) Finally, the *Jitter Controller* aims to measure the jitter produced in the exchange of messages between the different devices of the IoT environment. These measurements are simulation outputs that can be used to analyze the behavior of the system in this respect.

As for the components highlighted in orange, there are the *Connections Manager*, the *MongoDB Manager*, and the *MongoDB Client*. The *Connections Manager* is the component that manages the connections of fog and cloud nodes with the rest of the nodes in the IoT environment. The *MongoDB Manager* manages all the operations with the MongoDB Database, and the *MongoDB Client* is the component that allows the *MongoDB Manager* to perform these operations through requests. These components are already included in the first release of SimulateIoT. However, they have been extended to support mobile devices. For instance, the *MongoDB Manager* can make new requests related to the supporting mobility architecture, such as the information of each topic offered in this fog or cloud node. Information that mobile devices could request to determine whether is feasible for them to connect to that broker or not, as previously described.

Concerning the components not highlighted, they were initially developed for the first release of SimulateIoT. Although they are also part of SimulateIoT-Mobile, they did not require any extensions or modifications. Among, these components, it is possible to observe the *MQTT Broker*, the *MQTT Client*, and the *CEP Engine* among others. Note that these components are addressed in Section 2.1, reserved for the results of the first release of SimulateIoT.

Firstly, the components related to the mobility supporting architecture introduced in this contribution include the *Topic Discovery Service Client*, the *Broker Discovery Service Client*, and the *Token Security Service (TSS) Client*. These clients facilitate interaction with their respective services deployed on fog and cloud nodes (see Figure 2.7). Additionally, other

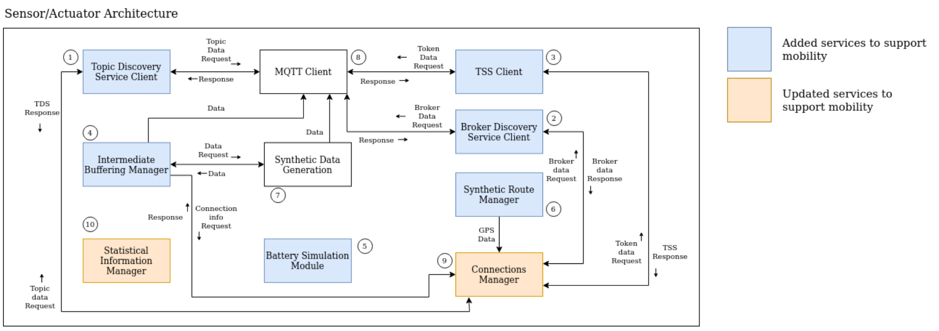


Figure 2.8: Software architecture of a mobile device generated. Figure source [4].

components have been included, such as the *Intermediate Buffering Manager*, which manages the data buffer for mobile devices, crucial for storing data when the device is disconnected and avoiding the loss of data. The *Synthetic Route Manager*, which implements the movement of mobile devices based on user-defined routes. The *Battery Simulation Module*, which tracks power consumption actions related to various device activities, including movement, connection changes, and publication frequency.

The extended components, highlighted in orange, are the *Connections Manager* and the *Statistical Information Manager*. The *Connections Manager* handles connections and disconnections to brokers. The *Statistical Information Manager*, is an enhanced component to comprehensively collect and store device data for post-simulation analysis. For instance, it stores the metrics from the *Battery Simulation Module*.

Lastly, the components retained from the first release of SimulateIoT are the *MQTT Client*, enabling data publication and notification receipt via the MQTT protocol, and the *Synthetic Data Generation* module, responsible for generating simulated data such as temperature or humidity readings.

Furthermore, note that this contribution was validated by carrying out two use cases. The first use case addresses the simulation of an IoT system designed for tracking animal movements through GPS devices. Thus, each animal was modeled with a GPS device. The devices are set to follow predefined routes to mimic flock movements. For data collection and analysis, three Fog nodes are deployed in key locations like lagoons. Note that the data gathering is done when the flock is near and their devices have coverage from the gateways modeled into the fog nodes. After collecting this data from the GPS devices, they send it to a central Cloud node for analysis. This structure enables effective tracking and analysis of animal behavior in their natural habitat. The second case study focuses on simulating a smart Personal Mobility Device (PMD) system within a city, addressing the increasing use of bicycles and electric scooters. To enhance management and user safety, PMDs were modeled with various sensors for real-time monitoring, including GPS for location tracking, pressure sensors for wheels, and timers to track usage duration. Additionally, actuators alert users to issues like inadequate wheel pressure. There are also Fog nodes modeled with gateways deployed throughout the city. These nodes not only handle data from PMDs but also forward it to Cloud node elements for storage and further analysis. As in the previous use case, note that these PMDs offload their data to the different Fog nodes when they receive coverage from their gateways. In addition, Fog nodes assess critical PMD data, such as lease term, battery level, and wheel pressure, enabling automatic notifications to users through the PMD's actuators, thus ensuring both operational efficiency and safety.

With these devices and their components, as well as with the mobility-supporting architecture, this contribution enables users to model, validate, generate, and simulate IoT systems with mobile devices. Thus, providing users with knowledge and insights to improve their designs. In addition, note that this contribution has been validated through carrying out two use cases.

2.4 IoT Simulations Towards Task-Scheduling Assessments: Task-Driven Design and Functionality

At this stage of the research, RQ3 (*In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating* *them?*) is the sole RQ that has not been yet comprehensively addressed. Regarding this RQ, initial findings show that the high level of abstraction facilitated by MDD, coupled with the developed concrete syntax, reduces the learning curve associated with using SimulateIoT and makes it a user-friendly tool. Additionally, SimulateIoT demonstrates a high level of agility in its ability to model, validate, and simulate IoT systems. This agility is attributed to the integration of these three processes within its methodology, proposed in the first contribution of this Ph.D. Thesis (see Section 2.1). In addition to this, SimulateIoT provides a suite of tools designed to efficiently facilitate these processes, thereby enhancing further the agility and effectiveness of its application.

These findings partially address RQ3. Nevertheless, SimulateIoT has not yet proven to have a high adaptive integration capability. Although SimulateIoT enables users to incorporate their own devices into simulations since it works with real IoT components, such as the MQTT protocol, this integration process requires manual intervention. A similar situation is observed with services and applications, which can be manually deployed on the fog and cloud nodes within SimulateIoT, but no facilities have been provided by SimulateIoT to help with this process.

In an effort to comprehensively address RQ3, SimulateIoT has been extended to include task scheduling capabilities. Task scheduling plays a pivotal role in IoT systems, as described in Section 1.1.4. Consequently, researchers are investing significant efforts towards optimizing and innovating task-scheduling algorithms. However, there is a notable scarcity of simulators that facilitate the testing of these task-scheduling algorithms. Moreover, among the available simulators, it is exceedingly uncommon to find one that is both up-to-date with the latest advancements in IoT systems and equipped to allow users to seamlessly integrate and test their task-scheduling proposals by default. So, the extension made to SimulateIoT aims to answer RQ3 by bridging this gap. For this purpose, this contribution places special emphasis on enabling users to integrate their own task-scheduling proposals into the broader simulation environment. This focus aims to explore the adaptive integration capabilities that SimulateIoT can offer in this respect, thus targeting to fully answer RQ3.

Therefore, to implement this contribution, the three primary components of SimulateIoT have been extended: the metamodel, the concrete syntax, and the model-to-text transformations. Thus, with this extension, users can graphically model, validate, and simulate IoT systems with task-scheduling capabilities.

The extension made to the metamodel is illustrated in Figure 2.10. Note that for the sake of clarity, this figure focuses exclusively on the new classes and relationships included in this contribution (highlighted in blue). This extended metamodel includes the integration of *Task Nodes* and *Task Apps* through the *TaskNode* and *TaskApp* classes. These components are the elements that generate and offload tasks to the system. These tasks are introduced by the *Workflow* class. This class can be linked with the *TaskNode* and *TaskApp* classes to define the specific workflows to be generated and offloaded by each of these components. A *Workflow* can be defined as a set of tasks and dependencies between tasks. So, to model a *Workflow*, *Task* and *Edge* classes have been added, enabling the representation of *Workflow* nodes and task dependencies. The metamodel's extension also includes hardware specification for each node, with the inclusion of *CPU* and *RAM* classes under the *Hardware_specification* class for detailed hardware resource modeling. The hardware specification is a critical aspect in this context as tasks have to be processed, and the time required for this process highly depends on the hardware of the processing node. Additionally, the new *Federation* class introduces federation capabilities, comprising *Links* and classes like *Delay_specification* and *Bandwidth_specification*' to model link characteristics between federated nodes. Thus, these extensions collectively enhance the metamodel's expressiveness, enabling users to model IoT systems with task-scheduling capabilities.

Regarding the concrete syntax, it remains the same tool as in the three previous contributions, but with the required graphical elements to represent the included task-scheduling concepts.

On the other hand, the model-to-text transformations have been also extended. These transformations enable users to generate IoT systems with task-scheduling devices and functions that users can model with the extended metamodel proposed in this contribution. Figure 2.10 illustrates a generic simulation deployment using the components generated with the extended model-to-text transformations. Components included in this contribution are marked in red while those from the previous versions are in blue.

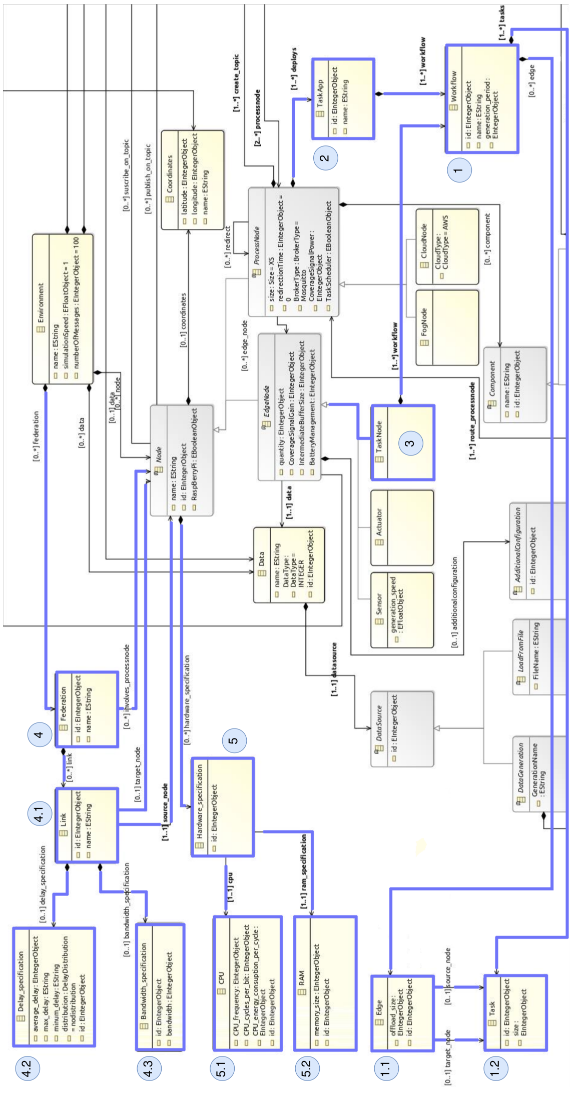


Figure 2.9: Excerpt of the SimulateIoT metamodel including task-scheduling concepts. It focuses on the new classes and relationships included in this contribution. Figure source [5].

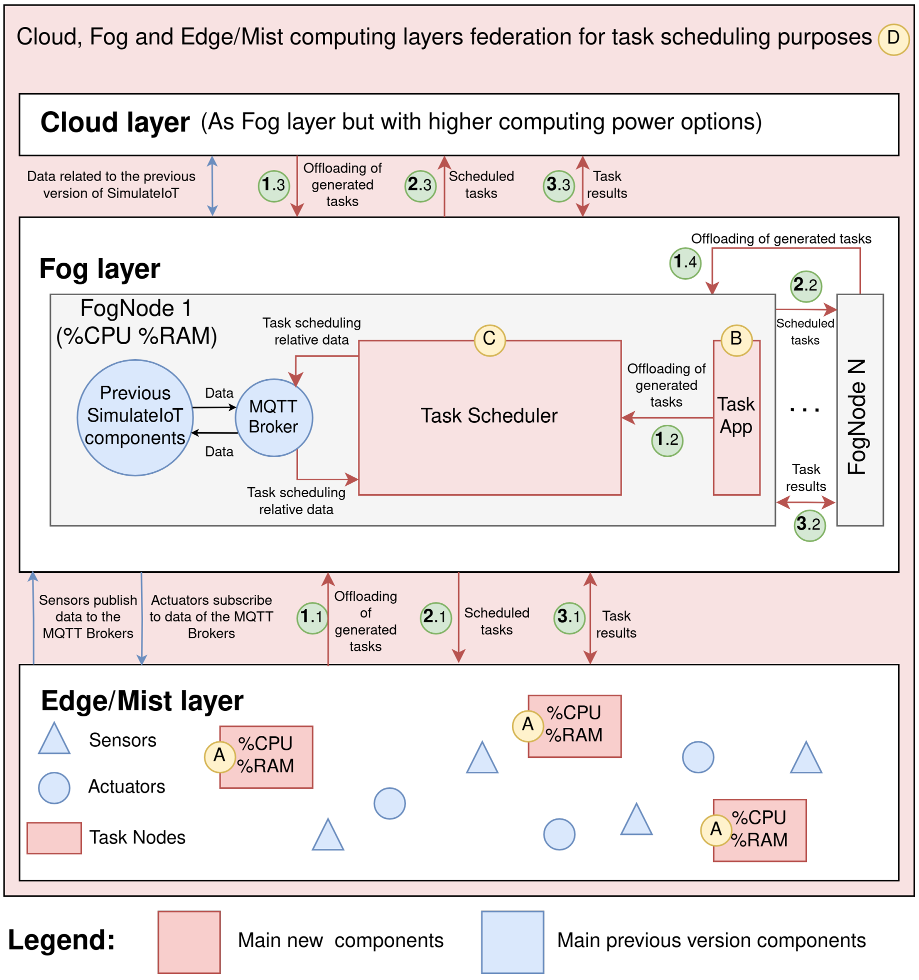

Cloud, Fog and Edge/Mist computing layers federation for task scheduling purposes D
Cloud layer (As Fog layer but with higher computing power options)
Data related to the previous version of SimulateIoT
1.3 Offloading of generated tasks
2.3 Scheduled tasks
3.3 Task results
Fog layer
Offloading of generated tasks
1.4
2.2
Scheduled tasks
FogNode 1 (%CPU %RAM)
Task scheduling relative data
C
B
Previous SimulateIoT components
Data
MQTT Broker
Data
Task Scheduler
Offloading of generated tasks
1.2
Task App
. . .
FogNode N
Task scheduling relative data
Task results
3.2
Sensors publish data to the MQTT Brokers
Actuators subscribe to data of the MQTT Brokers
1.1 Offloading of generated tasks
2.1 Scheduled tasks
3.1 Task results
Edge/Mist layer
Sensors
Actuators
Task Nodes
A %CPU %RAM
A %CPU %RAM
A %CPU %RAM
Legend:
Main new components
Main previous version components

Figure 2.10: A generic simulation generated by using model-to-text transformations from a model defined with the simulator developed in this contribution. Figure source [5].

Regarding the system illustrated in Figure 2.10, *Tasks* are generated and offloaded to the system by *Task Nodes* Ⓐ and *Task Apps* Ⓑ. Task offloading, depicted as (1.1), (1.2), (1.3), and (1.4), is performed to the fog layer.

Then, the *Task Scheduler* Ⓒ, deployed on *FogNode* ①, schedules these tasks and sends them back for processing, as shown by (2.1), (2.2), and (2.3). Each task is then processed by the respective *Task Processor* of fog, cloud, and *Task Nodes*. Note that *Task Processors* are the components with the role of performing the execution of tasks. Consequently, they are deployed in each component capable of processing tasks such as *Task Nodes*, fog, and cloud nodes. Once the processing of a task is performed, the results obtained are returned to the subsequent nodes ((3.1), (3.2), (3.3)). Lastly, note that all these interactions, such as task offloading or the return of the results from processed tasks, are subjected to the network bandwidth and delay modeled by the user. Thus, simulating more realistic task-scheduling IoT environments.

In short, simulations generated with the extended model-to-text transformations involve concurrent generation, offloading, scheduling, and processing of tasks, adhering to the user-defined model.

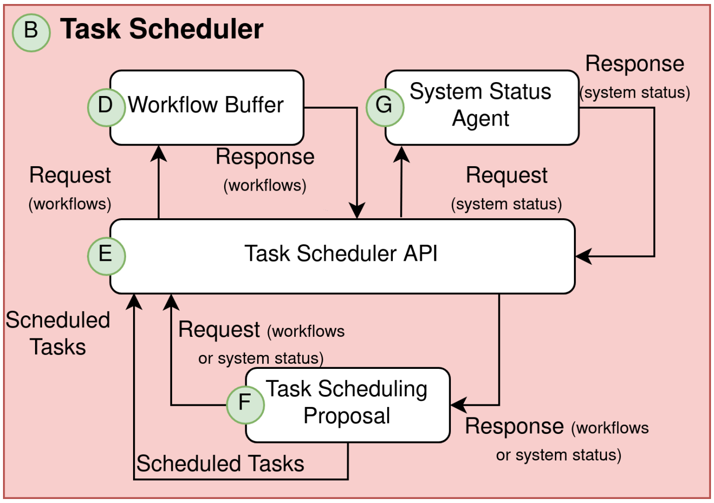


Figure 2.11: Task Scheduler component generated by using the model-to-text transformations developed for this contribution. Figure source [5].

On the other hand, within the context of the RQs, the key component introduced through this contribution is the *Task Scheduler*, depicted in Figure 2.11. This component is particularly significant as it is designed to offer a high integration adaptability, thus addressing a gap identified in previous contributions related to RQ3. Specifically, an aspect of RQ3 aims to probe the extent to which MDD can facilitate the development of highly adaptive IoT simulators. To this end, the *Task Scheduler* is equipped with an API that enables users to retrieve workflows submitted for scheduling. Additionally, it provides access to critical system status parameters, such as the network status, including the delay and bandwidth utilization of each system link, and the hardware resource usage of each processing node within the same federation as the *Task Scheduler*. Note that these parameters are pivotal in task scheduling and are frequently employed in task-scheduling algorithms documented in the literature to schedule tasks [61, 62, 63, 64]. So, by requesting this API, users can attach their task-scheduling algorithms to simulations by feeding them with the information retrieved from the API. Then, the algorithms can interact again with the API to return the tasks already scheduled. Lastly, the *Task Scheduler* redirects the scheduled tasks to the system for their processing. Consequently, users have the capability to incorporate their own algorithms into SimulateIoT, enabling them to evaluate algorithmic performance within the context of the simulated IoT environment. This integration necessitates the only use of an API, which facilitates feeding the algorithms and ensures seamless interaction with the other components of the simulated IoT system.

Chapter 3

Publications Overview

"Even the mightiest warriors experience fears. What makes them a true warrior is the courage that they possess to overcome their fears."

Dragon Ball Z: Majin Buu Saga (1994)
Toriyama, Akira

This section presents the dissemination efforts made to communicate the results achieved through this Ph.D. Thesis. It serves as a summary of all the journal and conference papers published as part of this Ph.D. Thesis, thereby giving an overview of them. These publications are presented in Sections 3.1 and 3.2.

3.1 Core Compendium Publications

This section is dedicated to outlining the compendium of publications that constitute this Ph.D. Thesis. According to the guidelines provided by the University of Extremadura, a Ph.D. Thesis can be structured as a compendium of publications. To fulfill this criterion, the Ph.D. candidate is required to have a minimum of two publications deemed highly relevant and one considered relevant. Publications classified as Q1 or Q2 in the JCR index are categorized as highly relevant. Those ranked as Q3 or Q4 are recognized as relevant. Additionally, presentations at international conferences rated as class 1 or 2 by the Global Guide to Scientific Conferences (GGS) conference rating system are also acknowledged as relevant publications.

This Ph.D. Thesis is composed of three publications classified as highly relevant and one publication deemed relevant. Detailed information regarding these publications is provided in Table 3.2.

Title	**Key Contribution**	**Journal Name (Year)**	**Journal Quality (JCR)**	**Chapter**
SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments	First release of SimulateIoT, which includes the foundational concepts of the IoT	IEEE Access (2021)	Q2	4
SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE	SimulateIoT-FIWARE, an extension of SimulateIoT where the target technology is the FIWARE platform	IEEE Access (2022)	Q2	5
Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile	SimulateIoT-Mobile, an extension of SimulateIoT that includes mobile devices and their supporting architecture	Pervasive and Mobile Computing (PMC, 2023)	Q2	6
SimulateIoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments	A release of SimulateIoT which includes the main task-scheduling concepts related to the IoT	Journal of Object Technology (JOT, 2023)	Q4	7

Table 3.1: Overview of Core Compendium Publications

3.2 Supplementary Publications

This section presents an overview of the supplementary publications that complement the core findings of this Ph.D. Thesis. These additional works, while not forming the core of this Thesis, provide valuable insights and further depth to the primary research. They encompass a range of topics that are tangentially related to the main subject matter, offering broader context and supporting evidence for the arguments and conclusions presented in the main body of the work. Thereby, note that they have not been included in the compendium to maintain a focus on the central findings and contributions of this dissertation.

Title	**Status**	**Conference Name**	**Nature**	**Appendix**
SimulateIoT: A model-driven approach to simulate IoT systems	Published	Doctoral Consortium in Computer Science (JIPII, 2022)	International	A
Designing and simulating IoT environments by using a model-driven approach	Published	Iberian Conference on Information Systems and Technologies (CISTI, 2022)	International	B
SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments	Published	Jornadas de Ciencias e Ingeniería de Servicios (JCIS, 2022)	National	C
Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment	Published	International Conference on Service-Oriented Computing (ICSOC, 2022)	International	D
Integrating Hardware in the Loop into IoT Systems Simulations: A Model-Driven Development Approach	Unpublished	Under development	-	-
SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE	Published	Jornadas de Ingeniería del Software y Bases de Datos (JISBD, 2023)	National	E

* The Table continues on the next page.

SimulateIoT-Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations	Published	International Workshop on MDE for Smart IoT Systems (MeSS)	International	F
Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile	Under review	Jornadas de Ingeniería del Software y Bases de Datos (JISBD, 2024)	National	-

Table 3.2: Overview of Supplementary Publications

Chapter 4

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

"Your focus determines your reality."

Star Wars: Episode I - The Phantom Menace (1999)
Lucas, George

Authors: José A. Barriga, Pedro J. Clemente, Encarna Sosa-Sánchez, Álvaro E. Prieto
Title: SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments
Year: 2021
Journal: IEEE Access
Quality (JCR): Q2

DOI: 10.1109/ACCESS.2021.3092528

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.

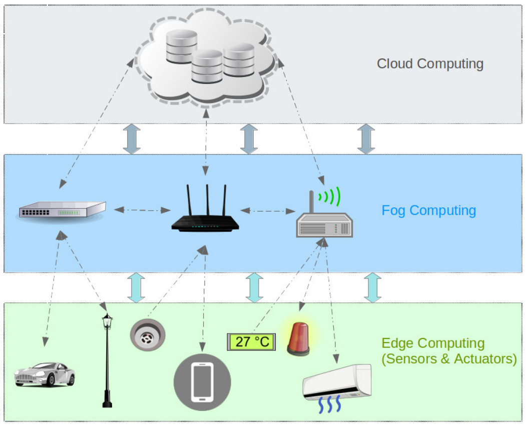


FIGURE 1. IoT architecture: Cloud, Fog and Edge computing.

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

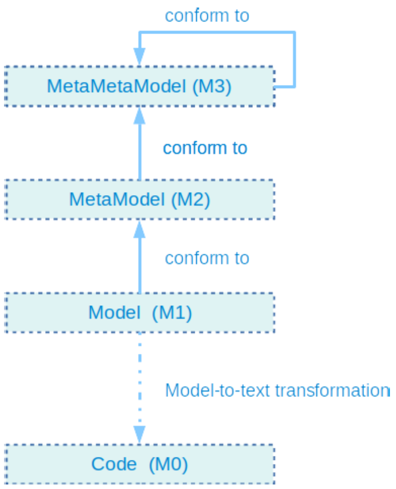


FIGURE 2. Model-driven development. Four layers of metamodeling.

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

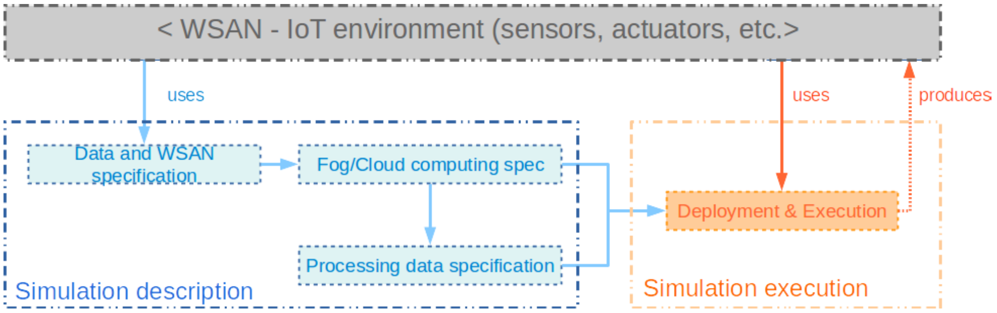

< WSAN - IoT environment (sensors, actuators, etc.>
uses
uses
produces
Data and WSAN specification
Fog/Cloud computing spec
Processing data specification
Deployment & Execution
Simulation description
Simulation execution

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

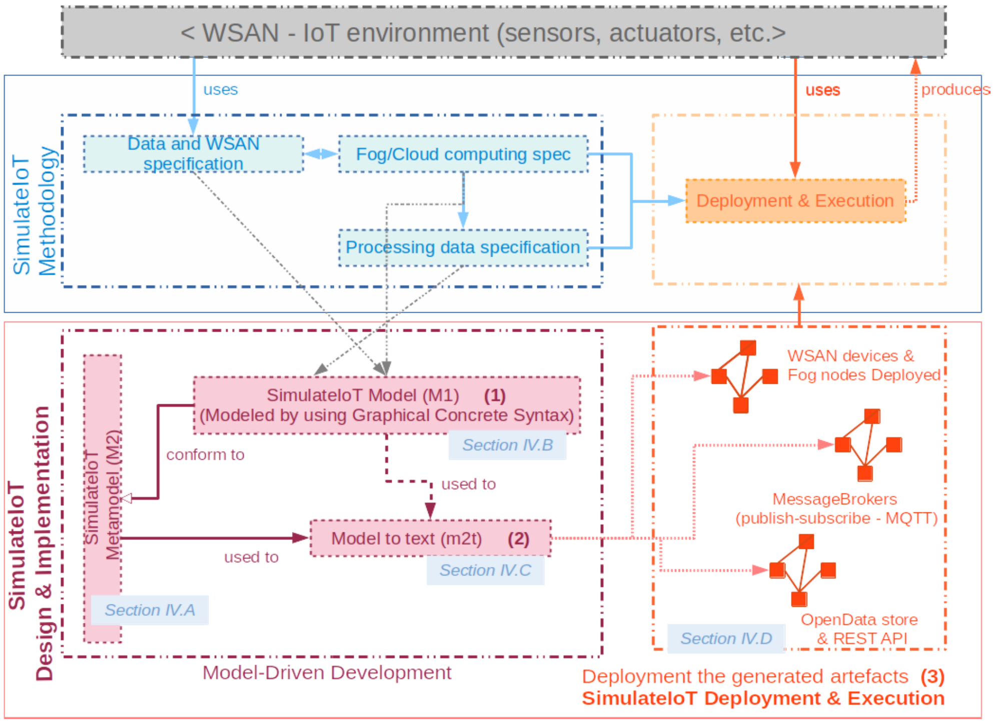

< WSAN - IoT environment (sensors, actuators, etc.>
uses
uses
produces
SimulateIoT Methodology
Data and WSAN specification
Fog/Cloud computing spec
Processing data specification
Deployment & Execution
SimulateIoT Design & Implementation
SimulateIoT Metamodel (M2)
SimulateIoT Model (M1) (1)
(Modeled by using Graphical Concrete Syntax)
Section IV.B
conform to
used to
Model to text (m2t) (2)
used to
Section IV.C
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

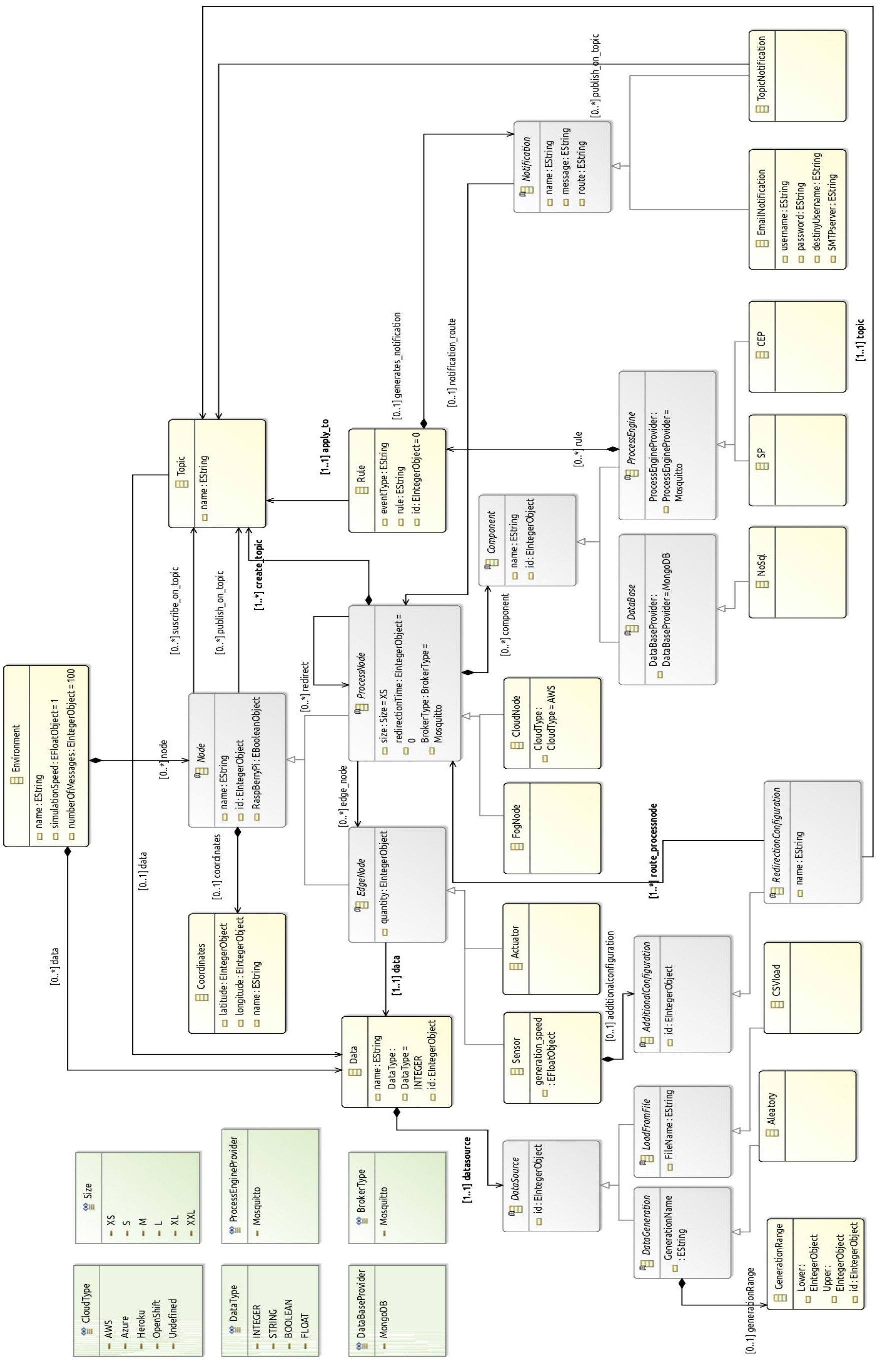


FIGURE 5. **SimulateIoT metamodel.**

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
```

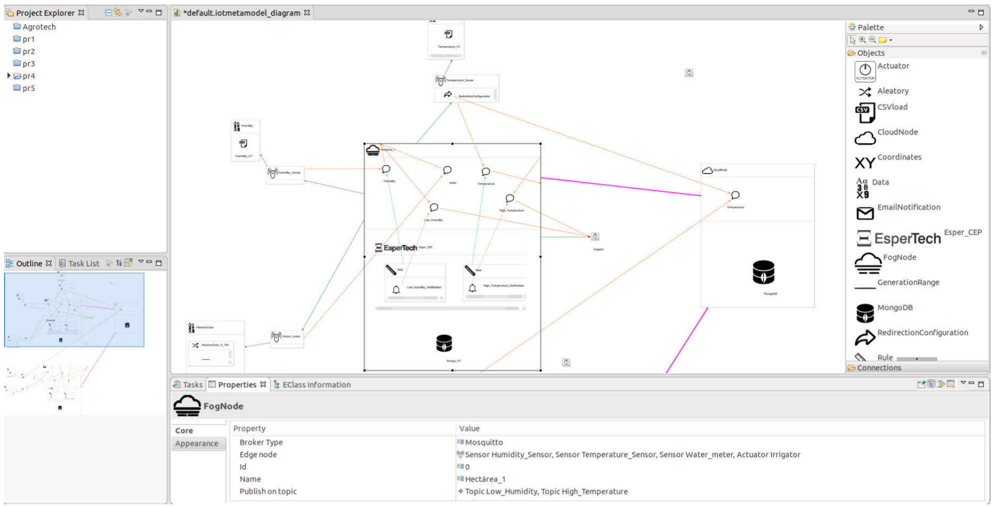
- *TopicNotification* generated by a *Rule* should be published on a *Topic* created by the *FogNode* which analyses data with this *Rule*:

```
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
}
```



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					

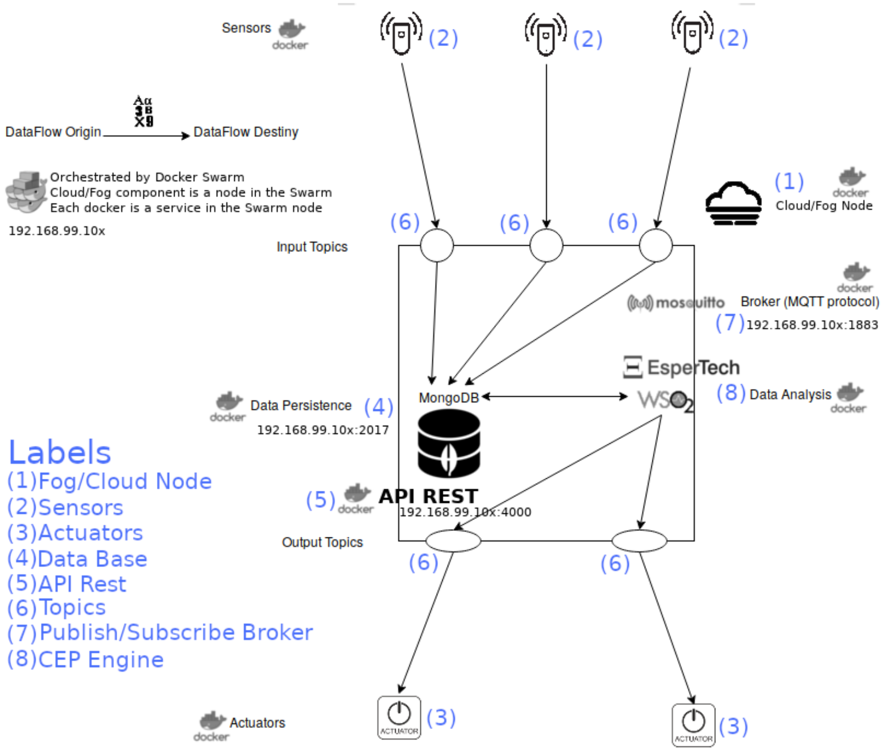


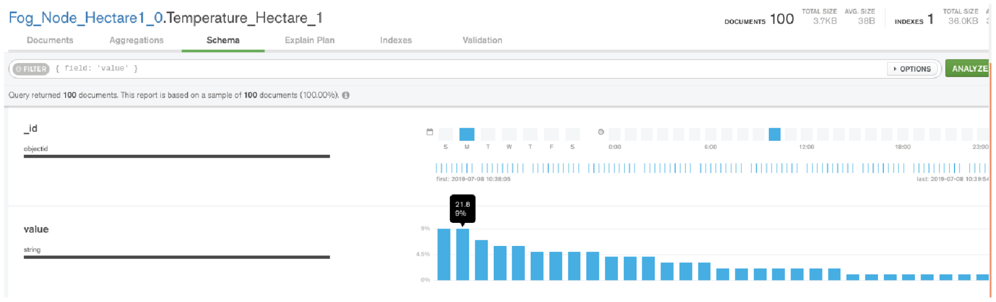
FIGURE 7. Deploy diagram.

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

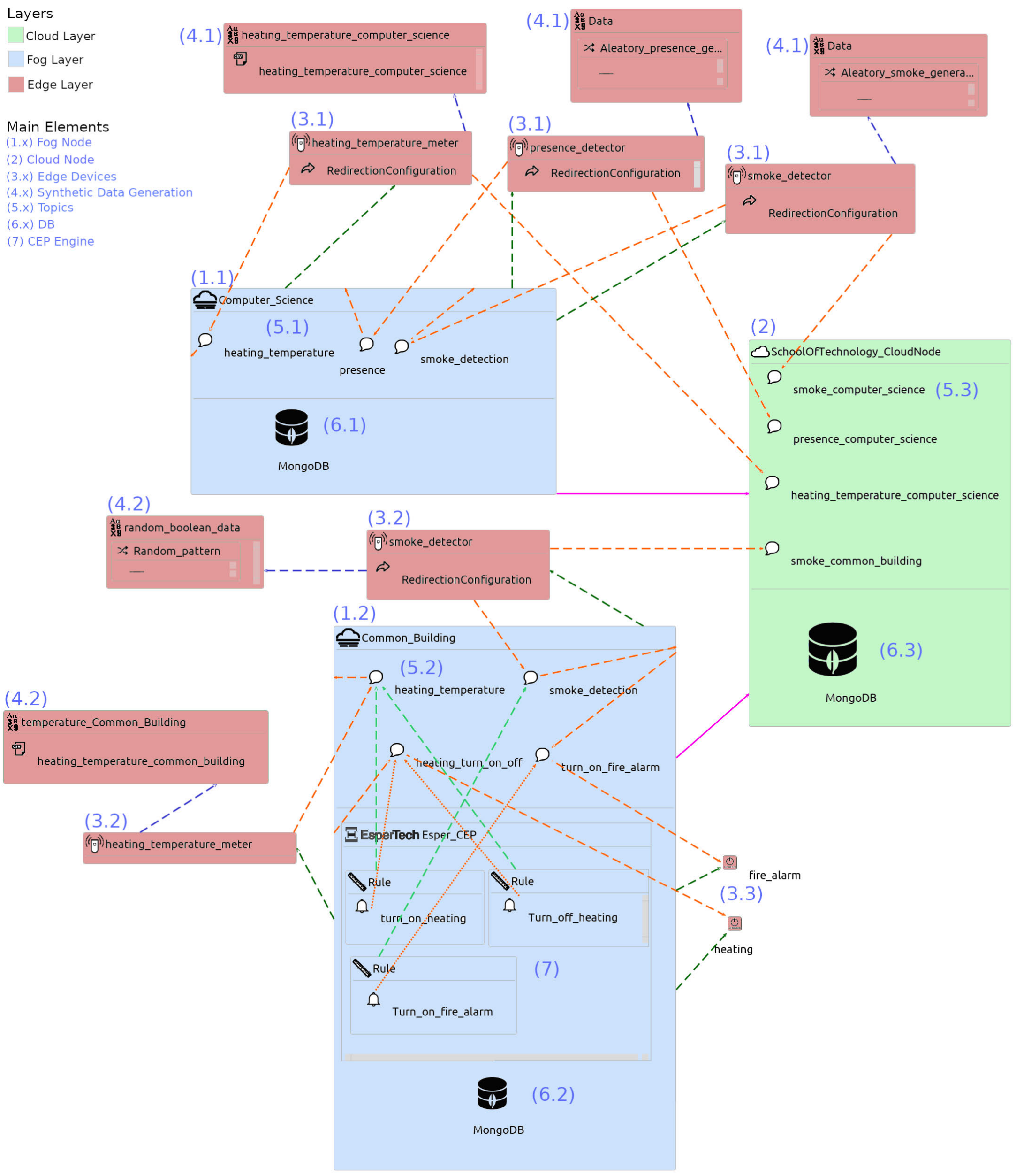


FIGURE 9. Case 01. The school of technology model conforms to the SimulateIoT metamodel.

(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*

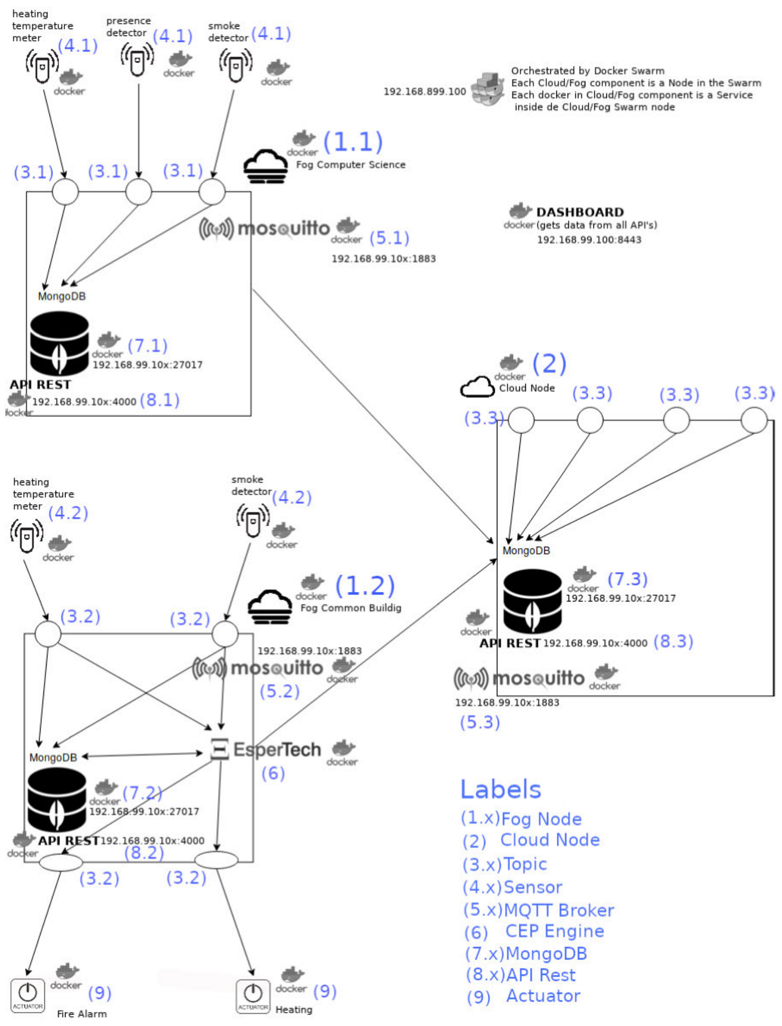

heating temperature meter (4.1)
presence detector (4.1)
smoke detector (4.1)
192.168.899.100
Orchestrated by Docker Swarm
Each Cloud/Fog component is a Node in the Swarm
Each docker in Cloud/Fog component is a Service inside de Cloud/Fog Swarm node
(1.1) Fog Computer Science
(3.1)
mosquitto (5.1) 192.168.99.10x:1883
DASHBOARD docker (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)
(9) Fire Alarm
(9) Heating
ACTUATOR
docker
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.

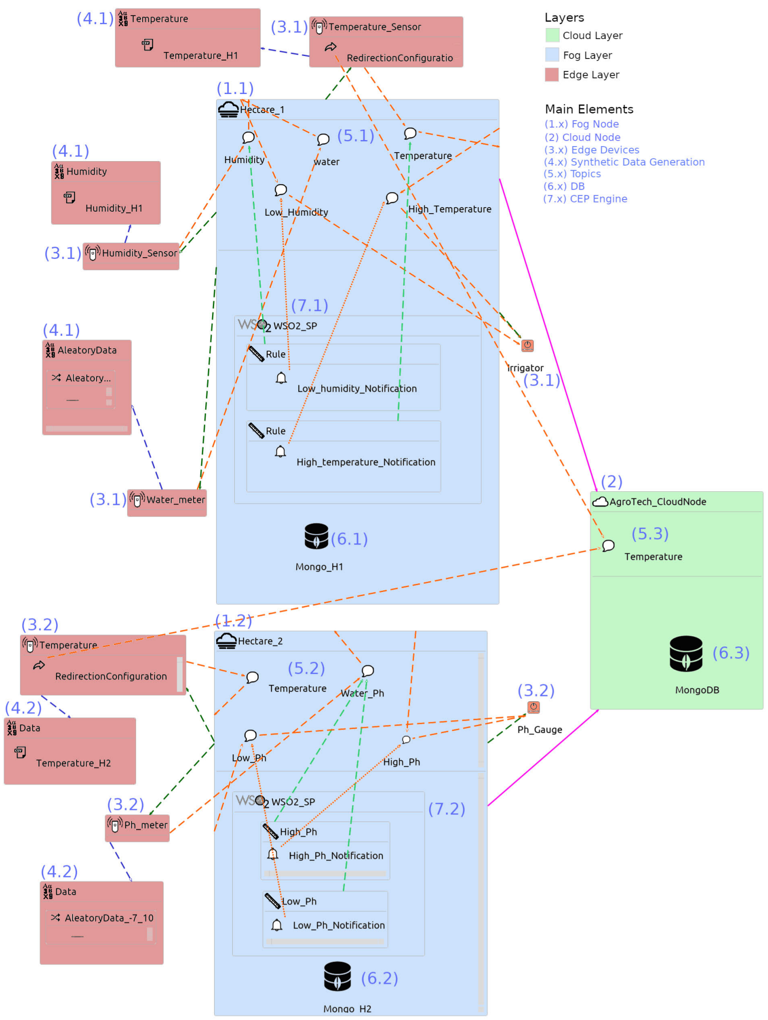


FIGURE 11. Case 02. AgroTech model conforming to the SimulateIoT metamodel.

- 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.

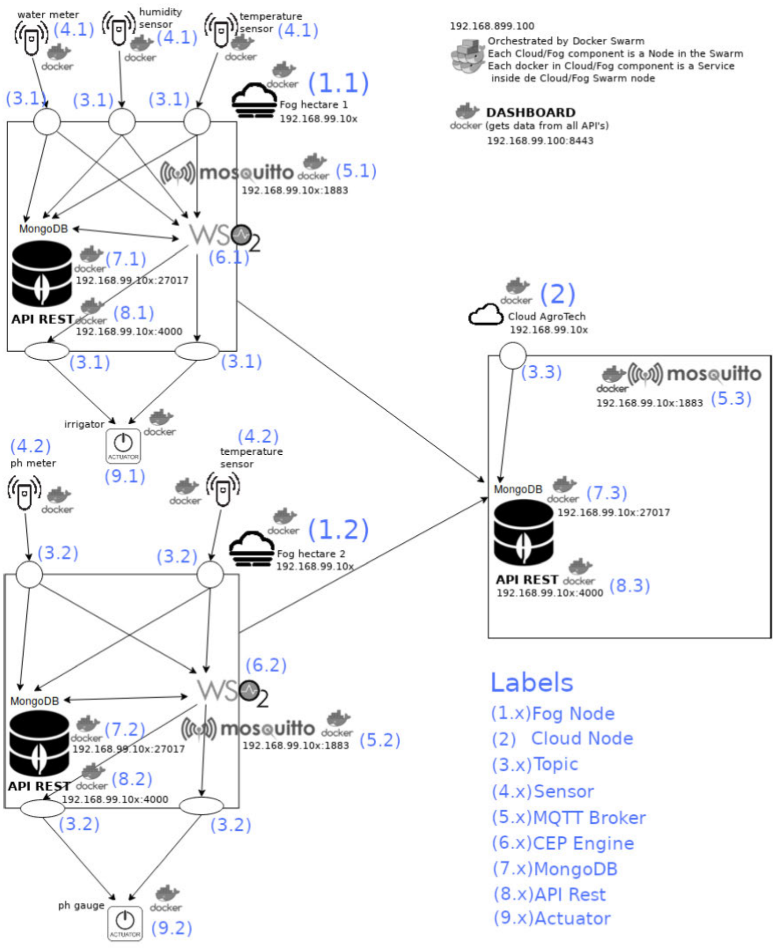

water meter (4.1)
humidity sensor (4.1)
temperature sensor (4.1)
docker
(1.1) Fog hectare 1 192.168.99.10x
(3.1)
mosquitto docker (5.1) 192.168.99.10x:1883
MongoDB
WSO2 (6.1)
(7.1) 192.168.99.10x:27017
API REST (8.1) 192.168.99.10x:4000
irrigator (9.1)
(4.2) ph meter
(4.2) temperature sensor
(1.2) Fog hectare 2 192.168.99.10x
(3.2)
(6.2)
(7.2) 192.168.99.10x:27017
mosquitto (5.2) 192.168.99.10x:1883
API REST (8.2) 192.168.99.10x:4000
ph gauge (9.2)
ACTUATOR
192.168.899.100
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
DASHBOARD
docker (gets data from all API's)
192.168.99.100:8443
(2) Cloud AgroTech 192.168.99.10x
(3.3)
mosquitto 192.168.99.10x:1883 (5.3)
MongoDB (7.3) 192.168.99.10x:27017
API REST (8.3) 192.168.99.10x:4000
Labels
(1.x)Fog Node
(2) Cloud Node
(3.x)Topic
(4.x)Sensor
(5.x)MQTT Broker
(6.x)CEP Engine
(7.x)MongoDB
(8.x)API Rest
(9.x)Actuator

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 multi-agent 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/

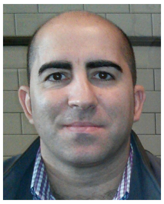
[50] Aqeel-ur-Rehman, A. Z. Abbasi, N. Islam, and Z. A. Shaikh, "A review of wireless sensors and networks' applications in agriculture," *Comput. Standards Interfaces*, vol. 36, no. 2, pp. 263–270, Feb. 2014.

[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.

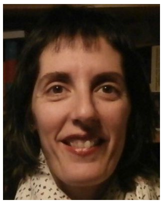
[52] N. Wang, N. Zhang, and M. Wang, "Wireless sensors in agriculture and food industry-recent development and future perspective," *Comput. Electron. Agricult.*, vol. 50, no. 1, pp. 1–14, 2006.

[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.

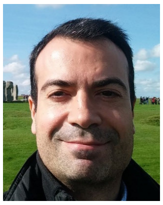
JOSÉ A. BARRIGA received the degree in computer science from the University of Extremadura, in 2017. He is currently working as a Junior Researcher with the University of Extremadura. He has been working in the IoT and the simulation IoT environments research areas since two years.



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.

Chapter 5

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

"Happiness can be found, even in the darkest of times, if one only remembers to turn on the light."

Harry Potter and the Prisoner of Azkaban (1999)
Rowling, J. K.

Authors: José A. Barriga, Pedro J. Clemente, Juan Hernández, Miguel A. Pérez-Toledano
Title: SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE
Year: 2022
Journal: IEEE Access

Quality (JCR): Q2
DOI: 10.1109/ACCESS.2022.3142894

Received December 28, 2021, accepted January 9, 2022, date of publication January 13, 2022, date of current version January 21, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3142894

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

JOSÉ A. BARRIGA, PEDRO J. CLEMENTE, JUAN HERNÁNDEZ, AND MIGUEL A. PÉREZ-TOLEDANO
Quercus Software Engineering Group, Department of Computer Science, Universidad de Extremadura, 10003 Cáceres, Spain

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

This work was supported in part by the Ministerio de Ciencia e Innovación (MCI) through the Agencia Estatal de Investigación (AEI) under Project RTI2018-098652-B-I00; in part by the Government of Extremadura, Council for Economy, Science and Digital Agenda, under Grant GR18112 and Grant IB20058; in part by the European Regional Development Fund (ERDF); and in part by the Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

ABSTRACT Systems based on the Internet of Things (IoT) are continuously growing in many areas such as smart cities, home environments, buildings, agriculture, industry, etc. This system integrates heterogeneous technologies into a complex architecture of interconnected devices capable of communicating, processing, analysing or storing data. There are several IoT platforms that offer several capabilities for the development of these systems. Some of these platforms are Google Cloud's IoT Platform, Microsoft Azure IoT suite, ThingSpeak IoT Platform, Thingworx 8 IoT Platform or FIWARE. However, they are complex IoT platforms where each IoT solution has to be developed ad-hoc and implemented by developers by hand. Consequently, developing IoT solutions is a hard, error-prone and tedious task. Thus, increase the abstraction level from which the IoT systems are designed helps to tackle the underlying technology complexity. In this sense, model-driven development approaches can help to both reduce the IoT application time to market and tackle the technological complexity to develop IoT applications. In this paper, we propose a Domain-Specific Language based on SimulateIoT for the design, code generation and simulation of IoT systems which could be deployed on FIWARE infrastructure (an open-source IoT platform). This implies not only designing the IoT system for a high abstraction level and later on code generation, but also designing and deploying an additional simulation layer to simulate the system on the FIWARE infrastructure before final deployment. The FIWARE IoT environment generated includes the sensors, actuators, fog nodes, cloud nodes and analytical characteristics, which are deployed as microservices on Docker containers and composed suitability to obtain a service-oriented architecture. Finally, two case studies focused on a smart building and an agricultural IoT environment are presented to show the IoT solutions deployed using FIWARE.

INDEX TERMS Model-driven development, Internet of Things, IoT simulation, services-oriented, FIWARE.

I. INTRODUCTION

The Internet of Things (IoT) is widely applied in several areas such as smart cities, home environments, agriculture, industry, intelligent buildings, etc. [45]. In order to build IoT applications, multiple technologies are available from configuring a specific sensor to analysing a vast amount of data in real-time. Thus, data should be, among other actions, stored, communicated, analysed, visualised and notified. For this, multiple IoT cloud platforms have emerged for development such as Google Cloud's IoT Platform [17], Microsoft Azure IoT suite [23], ThingSpeak IoT Platform [47], Thingworx 8 IoT Platform [48] or FIWARE [13].

Each IoT platform has its own characteristics and mechanisms to define devices, connect them, store and analyse data or carry out notifications that provoke the well-known (*vendor lock-in* problem [36]). Likewise, each IoT

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

platform offers different services and QoS which should be managed ad-hoc.

However, it is possible to define IoT solutions independent of the IoT platforms on which they will be deployed. For them, it is necessary to focus on the IoT application domain and not on their specific technological issues. Model-Driven Development(MDD) [43], [51] is able to tackle this heterogeneous technology (*vendor lock-in problem*) increasing the abstraction level where the software is developed, focusing on the domain concepts and their relationships. Thus, the IoT concepts and relationships are defined by a model which can be analysed and validated.

In this sense SimulateIoT [3] is an approach based on Model-Driven Development (MDD) to define IoT environments (Set of components, such as sensors, actuators, Fog or Cloud nodes, etc. that are part of an IoT architecture), generate its code and deploy it. Later on, the IoT environment generated could be simulated. This is because the MDD allows 1) the definition of a Domain Specific Language (DSL), able to model IoT environments, and 2) a *model-to-text (M2T) transformations* needed to code-generation and deploy the IoT environment.

However, SimulateIoT is limited to code generation towards proprietary infrastructure based on microservices. In order to show that an MDD approach is able to generate code on different technological platforms, it is interesting extending the code generation to other technological approaches such as cloud open-source IoT environments such as FIWARE [13]. It is an open-source project that provides a large catalogue of components for the development of IoT environments, including, among other functions and components to manage, analyse or store the data which are generated and shared in an IoT environment [13]. This paper presents the extension of the SimulateIoT MDD platform to deploy the IoT environments modelled in the FIWARE open-source IoT Platform. Consequently, it shows that is possible to model IoT environments independently of technology and deploy them on concrete IoT cloud platforms such as FIWARE. Thus, the IoT concepts and relationships are defined by a model which can be analysed and validated. Besides, the IoT environment code, including all the artefacts needed, can be generated from a model using model to text transformations, decreasing error-proneness and increasing the user's productivity.

The main contributions of this paper include:

- A proposal that shows that Model-Driven Development is a suitable approach to develop tools and languages to tackle the complexity of heterogeneous technology successfully in the context of IoT environments such as sensors, actuator, databases, complex-event processing engines, communication protocols, etc.
- A Model-Driven Development proposal to generate IoT solutions based on FIWARE infrastructure, hiding the complexity of a cloud IoT framework.
- An extended version of *SimulateIoT Domain Specific Language named SimulateIoT-FIWARE* that can be used to define *IoT environments* and generate their implementation based on the components provided by *FIWARE*. That means reusing both *SimulateIoT* Abstract Syntax (Metamodel and OCL constraints) and *SimulateIoT* Concrete Syntax, while the *M2T* transformations have been improved and adapted to generate, configure and deploy FIWARE artefacts.
- Two case studies have been developed following the methodology and tools presented, focusing on different kinds of IoT systems. Note that, these two use cases are the same as those defined in [3], demonstrating that it is possible to deploy these same environments on the FIWARE platform.

The rest of the paper is structured as follows. Section 2 introduces the *FIWARE* architecture and how IoT systems should be implemented on it. Section 3 describes shortly *SimulateIoT* Domain Specific Language. Section 4 presents the integration of *SimulateIoT* DSL with the *FIWARE* architecture and artefacts. Section 5 describes the aspects related to code generation towards *FIWARE* technology from models. Then, Section 6 illustrates the use of the Model-Driven approach presented in two different case studies: Smart Building and Smart Agro. In Section 7 the discussion and limitations of the approach are described, before presenting the related works in Section 8 and the conclusions in Section 9.

II. THE FIWARE ARCHITECTURE

FIWARE is an open-source project that defines and implements a universal set of standards for context data management with the aim of optimising the development of IoT environments in different fields, such as *Smart Cities* [16], [27], *Smart Buildings* [15], *Smart Agro* [25], Smart Energy, Smart Industry [13], etc. *FIWARE* makes IoT simpler by means of driving key standards for breaking the information silos, transforming Big Data into knowledge, enabling data economy and ensuring sovereignty on your data [13]. Consequently, using *FIWARE* to design, develop and manage an IoT environment makes it possible reuse the advantages aforementioned and to reuse the knowledge and tools developed as part of *FIWARE*. In this sense, although *FIWARE* has several components to support the developing of IoT environments in the scenarios aforementioned, the main and only mandatory component of any FIWARE solution is *FIWARE* Context Broker. Mention that the concept of *context* within FIWARE, is the state in which the IoT environment is at a given time. Thus, the context elements or data are those that give context to the environment, i.e., they define a characteristic of the environment, such as climatic data of the environment as temperature or wind speed and also data of the architecture of the environment, such as geoposition of an element or the speed at which it moves.

The *FIWARE* implementation is based on a set of layers and integrated elements such as i) Interface to IoT, Robotics and third party systems, ii) Core Context Management, iii) Context Processing, Analysis and Visualisation and iv) Data/API management and Publication and Monetisation of Context Information. Each layer is supported by several tools such as *Context Broker, Complex Event Processing, IoT Broker or IoT Backend Device Manager* that constitute the architecture that can be seen in Figure 1. These elements communicate among themselves through the *NGSI* protocol [14], although *FIWARE* has a middleware that acts as a bridge between *NGSI* and protocols such as *MQTT* or *HTTP* to communicate with external elements. Next, the most important elements of the *FIWARE* architecture (Figure 1) are defined:

- The Context Broker element named *Orion* (Figure 1-1) is the core of the *FIWARE* architecture. *Orion* allows the management of the complete data lifecycle including updates, queries, registrations and subscriptions. In other words, *Orion* allows creation and registration of the context elements, such as sensors, actuators, CEP engines, etc. and manages them through updates and queries. In addition, devices can subscribe to context information so when some condition occurs these devices receive a notification [10]. Moreover, it should be mentioned that *Orion Context Broker* includes *MongoDB* [30], a *NoSql* database [22] for the data persistence required to perform the above-mentioned functions as well as those of other *FIWARE* elements.
- A *CEP* (Complex Event Processing) (Figure 1-2) [8] element allows more complex analysis techniques than *Orion Context Broker* subscriptions. Thus, in order to develop CEP applications in *FIWARE*, a CEP component named *CEP Perseo* [12] has been included in the *FIWARE* architecture. *CEP Perseo* is a *CEP* software based on the *Esper* language [9], i.e., software that listens for events that come from context information to identify event patterns described by rules, in order to immediately react to them by triggering actions [12]. *CEP Perseo* is composed of two basic elements, *Perseo front-end* and *Perseo core*. *Perseo front-end* stores the event rules (written using *Event Processing Language (EPL)* [37]) on MongoDB and then, processes the incoming events sending them to *Perseo Core*. Next, *Perseo Core* checks incoming events against the event rules and notifies *Perseo front-end* if an action must be executed [11]. Finally, *Perseo front-end* sends notifications to the appropriate devices.
- The *IoT Broker* (Figure 1-3) element allows developers to use a message broker such as *Mosquitto* [31] (based on MQTT protocol) to ensure message exchange among the devices or components defined in an *IoT environment*. It implements a publish/subscribe communication protocol that makes it possible to interconnect the IoT devices and components such as *Sensors*, *Actuators* or other *FIWARE* components.
- The *IoT Backend Device Management* Figure 1-4) consumes data from *Sensors* Figure 1-5) and sends it to the *Actuators* (Figure 1-6). *IoT-Agent* carries out this task. Thus, *IoT-Agent* acts as a bridge between the *NGSI* protocol and other protocols such as *MQTT* or *HTTP*. In this way, the *IoT-Agent* brings a standard interface to all *IoT* interactions at the context information management level (*Orion Context Broker*) allowing each *IoT* device to be able to use it own protocols to communicate with *FIWARE*.

The elements described above are the main components of the *FIWARE* platform which are enough to develop an IoT environment on the FIWARE platform. However, as can be seen in Figure 1, FIWARE offers a larger number of components. These components aim to meet specific needs such as service orchestration, Big Data processing, payment management, etc.

For instance, supposing a general Smart Building case study where several sensors are deployed in the building, these sensors send data to a *FIWARE* instance, which are analysed in real-time by a CEP component in order to notify different event rules detected. Additionally, data processed is stored for later analysis. The system architecture deployed to support this IoT environment can be observed in Figure 2.

Developing this case study includes, among others, the following:

- Define the sensors and actuators into *Orion context broker*.
- Define and configure the messages that should be interchanged from/to devices to *FIWARE* architecture.
- Configure and deploy each node for the *FIWARE* infrastructure, including Orion context broker, CEP perseo, databases, messages brokers, and so on.
- Define the EPL rules and deploy them on the CEP Perseo.
- etc.

Additional issues should be taken into account and they should be resolved by implementing additional ad-hoc modules:

- Components such as *Perseo* notifies event matched by HTTP protocol. Consequently, in order to notify *Actuators* who are subscribed to a specific *Topic*, an HTTP2MQTT converter should be developed. In Figure 2 this module is named *NotificationMiddlewareComponent*.
- Originally, event patterns analysis can't be defined on *Topic* data. So, an additional infrastructure based on Topic data should be registered on *Orion*. In Figure 2 is named *OrionTopicManager*.

Developing an IoT environment by hand involves tedious and error-prone tasks. So, the complexity of the whole process defined previously to implement and deploy the IoT environment using heterogeneous technology should be tackled by increasing the abstraction level of the defined IoT environment. Therefore, the SimulateIoT model-driven approach is extended and used to model and generate IoT environments on FIWARE platform.

III. SimulateIoT: A MODEL-DRIVEN APPROACH TO DEVELOPING IoT SIMULATION ENVIRONMENTS

In a Model-Driven Development approach like *SimulateIoT*, the software development is guided through Models (M1)

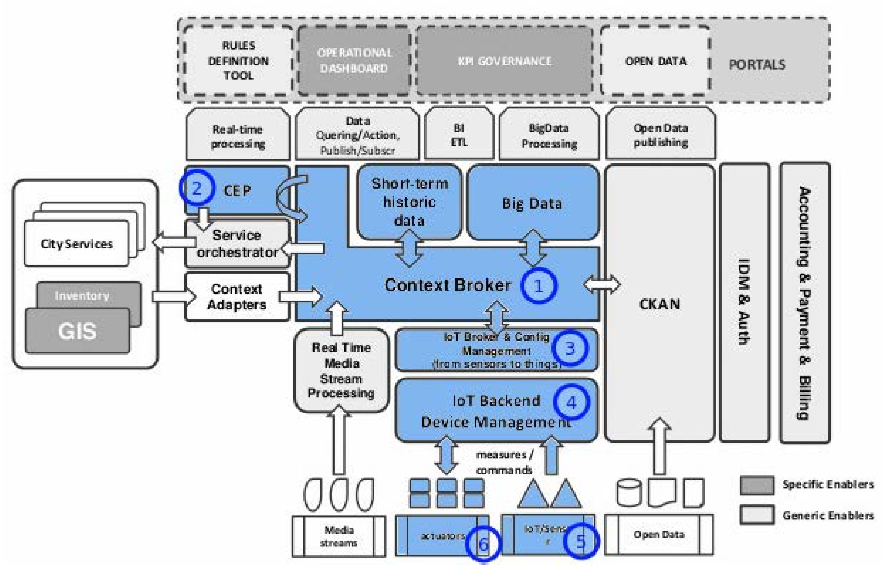


FIGURE 1. FIWARE architecture [13].

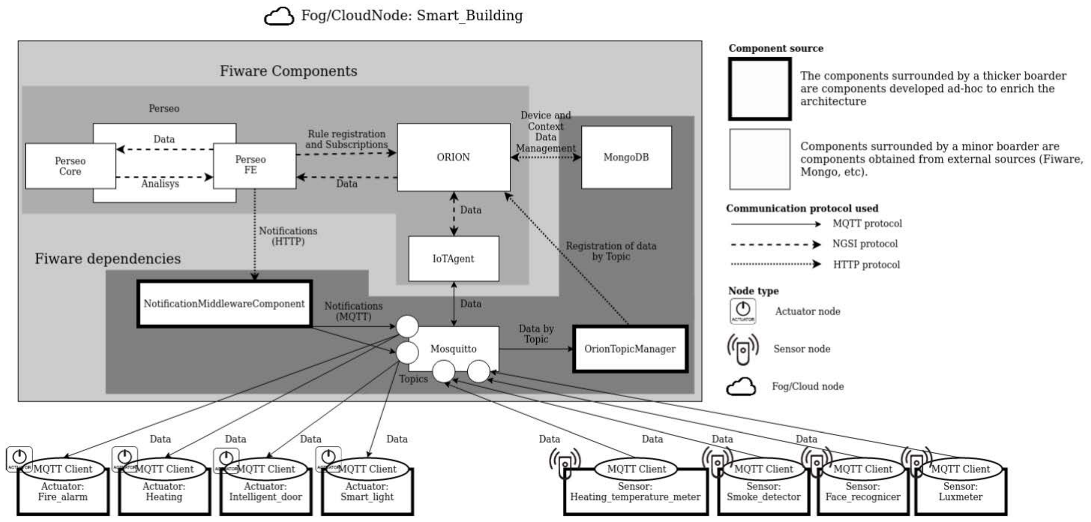

Fog/CloudNode: Smart_Building
Fiware Components
Perseo
Perseo Core
Data
Analisys
Perseo FE
Rule registration and Subscriptions
ORION
Device and Context Data Management
MongoDB
Data
Notifications (HTTP)
IoTAgent
Registration of data by Topic
Fiware dependencies
NotificationMiddlewareComponent
Notifications (MQTT)
Mosquitto
Data by Topic
OrionTopicManager
Topics
MQTT Client
Actuator: Fire_alarm
Actuator: Heating
Actuator: Intelligent_door
Actuator: Smart_light
Sensor: Heating_temperature_meter
Sensor: Smoke_detector
Sensor: Face_recognicer
Sensor: Luxmeter
Component source
The components surrounded by a thicker boarder are components developed ad-hoc to enrich the architecture
Components surrounded by a minor boarder are components obtained from external sources (Fiware, Mongo, etc).
Communication protocol used
MQTT protocol
NGSI protocol
HTTP protocol
Node type
Actuator node
Sensor node
Fog/Cloud node

FIGURE 2. Smart building case study architecture deployed on *FIWARE*.

which 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 [46] and so on. A MetaModel defines the concepts and relationships in a specific domain in order to model partial reality based (conceptual model). Additionally, OCL is formal language used to describe semantic expressions on Metamodels such as UML. These expressions typically specify invariant conditions that must hold for the system being modeled [35]. So, Model conforms to a MetaModel requires to validate with this semantic extensions (OCL invariants). Later on, the validated models are used to generate totally or partially the application code by model-to-text transformations [44]. Thus, the software code can be generated for a specific technological platform, improving the technological independence and decreasing error proneness.

SimulateIoT is a tool that uses model-driven development techniques to manage the IoT environments definition using models, so, the models guide the system description and the code generation. Subsequently, the code generated can be deployed through several hosts or be used to deploy a simulation of the IoT environment. There are three main elements of *SimulateIoT* tools: 1) a Metamodel definition, 2) a Graphical Concrete Syntax definition and 3) the *M2T* transformations to generate the code artefacts needed to deploy, monitor and measure the IoT environment. In order to be a self-contained paper, next the *SimulateIoT* proposal is explained.

Figure 3 defines the domain metamodel including concepts related to *sensors*, *actuators*, databases, fog and cloud nodes, data generation, communication protocols, stream processing, and deployment 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 interchange among the nodes (numberOfMessages).
- *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. Furthermore, the device position can be defined using *latitude* and *longitude* attributes.
- The *EdgeNode* element makes it possible to define simple physical devices such as a sensor or an actuator without process capacities. Each *EdgeNode* could be linked with *ProcessNode* elements by *Topic* 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 in the IoT environment. Additionally, the *CoverageSignalGain* attribute allows users to define the coverage reception capacity (offered by the different *ProcessNodes*) of the device.
- A *Sensor* element extends the *EdgeNode* element and defines a set of characteristics such as *id* or *generation_speed*. 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.
- An *Actuator* element is a device in the IoT environment that can execute an action from a set of inputs. For instance, the inputs could determine that an actuator turns a light on or off; 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. Obviously, the *Topic* element is a flexible concept to manage the data interchange.
- *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 [1], [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 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. This strategy allows specifying the *ProcessNode* element capacity and associating specific constraints, for example, in an XS *ProcessNode* a *ProcessesEngine* such as a Complex Event Processing (CEP) engine cannot be deployed. Hence, granularity labels are used as in a *Scrum* project development [42] to define task complexity. As mentioned, *ProcessNode* can define *Topic* elements, with 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. The attribute *BrokerType* defines the message-oriented broker that currently is established by *Mosquitto*. In addition, the *ProcessNode* element hides the complexity of 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 processing or the complex event processing capabilities help to define when an *Actuator* element should carry out an action. Finally, the *CoverageSignalPower* attribute is used to establish the range of coverage offered by a *ProcessNode* for those mobile devices that want to connect to it.

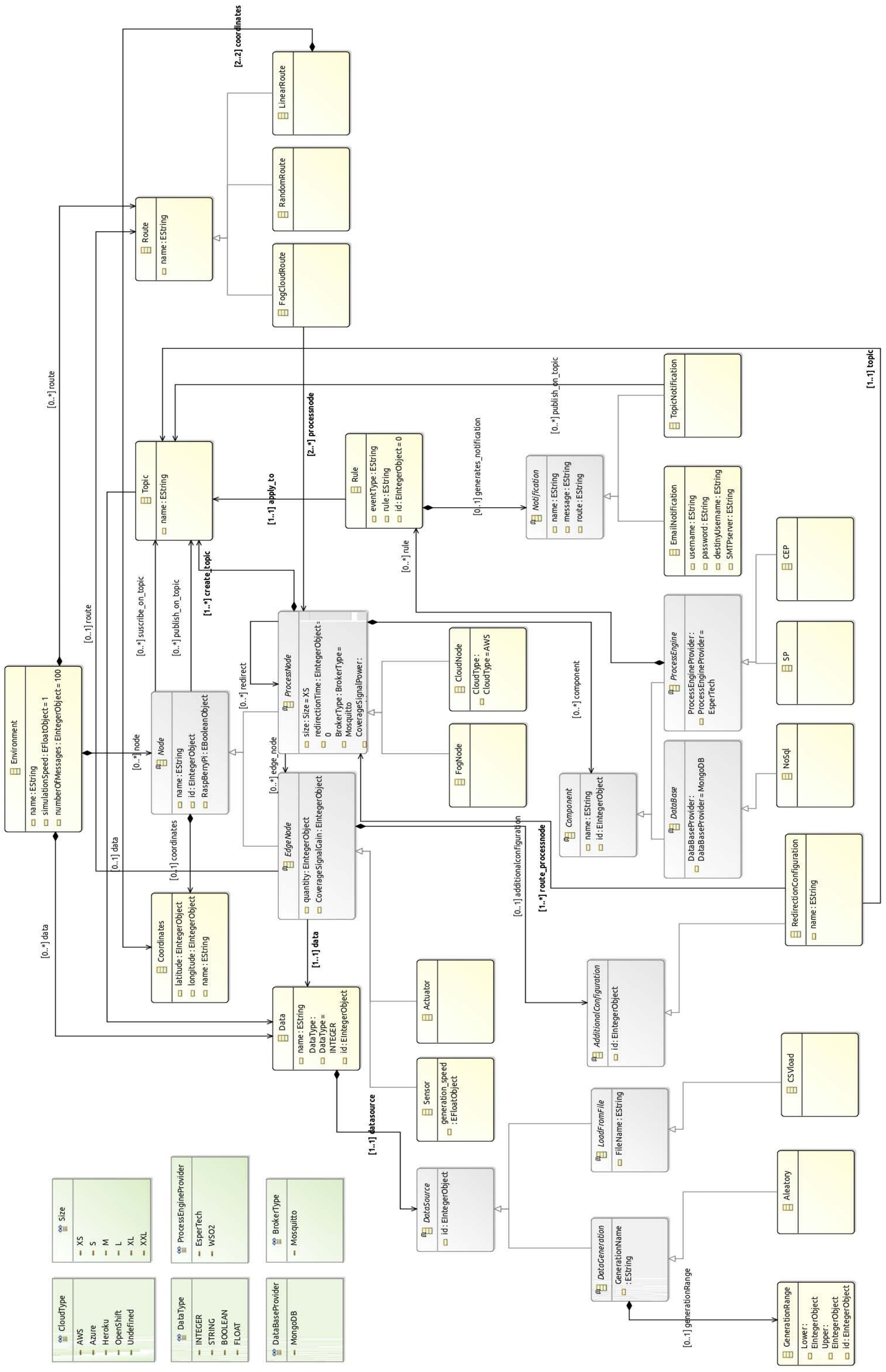


FIGURE 3. SimulateIoT metamodel.

- *FogNode* allows users to describe fog computing instances [5] 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 storage and data processing.
- *CloudNode* extends *ProcessNode* and allows describing a special node deployed in 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. Usually, a CEP (Complex Event Processing) engine has a higher process capacity and lower latency than an ESP (Event Stream Processing) engine [4], [26].
- Rule elements are linked with the *ProcessEngine* elements defined at the *ProcessNode* element. *Rule* elements can be defined using the EPL language defined for a concrete *ProcessEngine* kind.
- *Notification* elements make it possible to throw alerts by using several notification kinds: *TopicNotification* or *eMailNotification*. Obviously, the *notification* hierarchy could be extended in further metamodel versions.
- *Route* element allows to define the route throughout the coordinates by which the mobile device must move. For this purpose, 3 different methods have been included for their generation 1) *Fog/Cloud Route*, 2) *Linear Route*, 3) *Random Route*. *Fog/Cloud Route* allows the user to establish a route from the selection of several Fog/Cloud nodes so that the mobile device will move sequentially among the selected nodes. *Linear Route* allows the user to define 2 coordinates, in this way the mobile device will move in a linear way between these coordinates. Finally, *Random Route* generates a random route at run time.

Later on, the SimulateIoT models can be created by using a Graphical Concrete Syntax (Graphical editor) defined by using Eugenia [24] from the *SimulateIoT* metamodel. Figure 4 shows an excerpt from this graphical editor. It helps users to improve their productivity allowing not only defining models conforming to *SimulateIoT metamodel* but also their validation using OCL constraints [35].

Once the models have been defined and validated conforming to the *SimulateIoT metamodel*, several artefacts can be generated using an *M2T* transformation defined using Acceleo [40]. The generated software artefacts include an MQTT messaging broker, device infrastructure, databases, a graphical analysis platform, a stream processing engine, a docker container, etc.

IV. USING A MODEL-DRIVEN DEVELOPMENT APPROACH TO GENERATE IoT APPLICATIONS WHERE FIWARE IS A TARGET TECHNOLOGY

This section describes how to apply a Model-Driven Development approach (based on SimulateIoT) to generate IoT applications based on *FIWARE*. For this purpose, the tools previously described (*SimulateIoT* and *FIWARE*) have been integrated. In this way, an extended version of *SimulateIoT* can define *IoT environments* and carry out M2T transformations based on the components provided by *FIWARE*. That means reusing both *SimulateIoT* Abstract Syntax (Metamodel and OCL constraints) and *SimulateIoT* Concrete Syntax while the *M2T* transformations have been improved and adapted to generate, configure and deploy *FIWARE* artefacts. Next, *SimulateIoT-FIWARE* components and the *SimulateIoT* components are compared, identifying the main *FIWARE* components that should be integrated, configured and deployed through the new *M2T* transformations based on *SimulateIoT-FIWARE* components.

A. SimulateIoT VS SimulateIoT-FIWARE

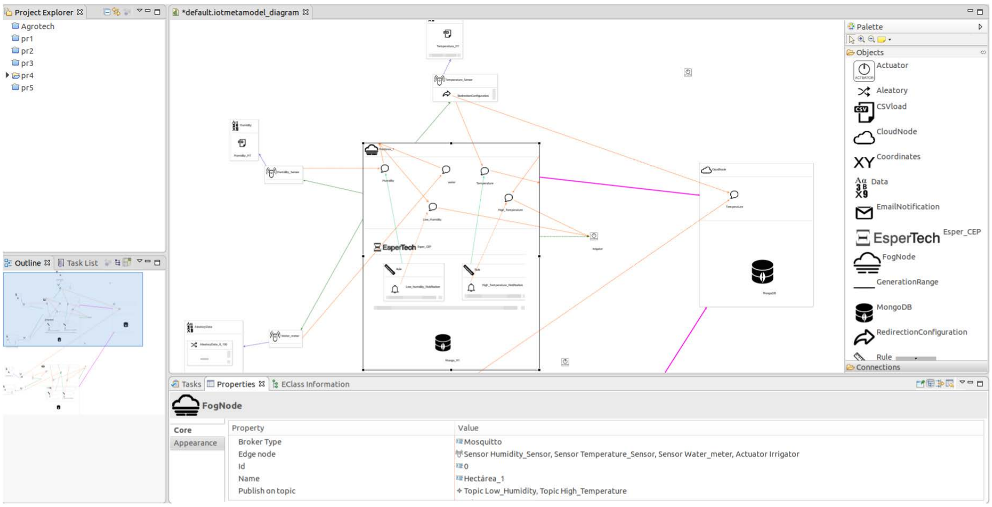
This section shows the differences between *SimulateIoT* and *SimulateIoT-FIWARE* version of *SimulateIoT*. Below are the metamodel classes whose components or functions have been modified after integration with (*SimulateIoT-FIWARE*).

From the *FIWARE* point of view, *Sensors* and *Actuators* (see Figure 1-(5 and 6)) are external elements which the *FIWARE* architecture is interconnecting. Consequently, the code generation for several concepts defined on the *SimulateIoT* models such as *Sensors* or *Actuators* among others do not have direct mapping to *FIWARE* components as they are external to *FIWARE*. Thus, their logic has been updated.

Next, Table 1 compares for each main *SimulateIoT* metamodel element (ProcessNode, Component Database or Component Process Engine) how it is implemented in both *SimulateIoT* and *SimulateIoT-FIWARE*, including the description of each component.

In addition to the mapping defined in Table 1, two components have been specifically developed for *SimulateIoT-FIWARE*: *NotificationMiddleware* and *OrionTopicManager*. Besides, as has been previously mentioned, the *Sensors* behaviour has been modified.

- **Sensors**. *Sensors* have the same components in *SimulateIoT* and in the *SimulateIoT-FIWARE*, however, in *SimulateIoT-FIWARE Sensors* publish their data twice. On the one hand, one of the publications is directly addressed to *Orion Context Broker* to enable it to record the information published by each *Sensor* separately, so that the data published by each *Sensor* can be consulted independently. On the other hand, the other publication is addressed to the *OrionTopicManager* component which allows *Orion Context Broker* to record the data published in a certain *Topic*. In this way,



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

all the data published in a specified *Topic* can be consulted independently from the *Sensor* that published it. In this way, for instance, the *CEP* component can apply rules by Topic and it does not need to collect and consult the data that each sensor has published in the *Topic* where it is going to apply the rules. As for the topology of *Topics* concerning the receipt of publications by each sensor independently, it is as follows: ''*/token/SensorName + SensorId/attrs*'' for instance, when *token* is *1234* and *SensorName* and *SensorId* are parameters predefined in a *Sensor* element within the *IoT Environment* model, we get ''1234/temperaturemeter5/attrs.''

- **NotificationMiddleware**. Since *CEP Perseo* only sends its notifications by the *HTTP* protocol, a middleware is necessary to act as a bridge between the *HTTP* protocol and the *MQTT* protocol as this is used by the *Actuator* elements to receive data or to receive notifications in this instance. *NotificationMiddleware* performs the above actions.
- **OrionTopicManager**. In *SimulateIoT*, the rules can be applied by *Topic*. However, due to the internal operation of *Orion Context Broker* and *CEP Perseo*, applying rules by *Topic* becomes more complex. To cope with this complexity, an additional module named *OrionTopicManager* has been developed, enabling *CEP Perseo* to apply its rules by *Topic* as in *SimulateIoT*. It allows reusing the *SimulateIoT* metamodel and related tools.

The *Sensors* logic, *NotificationMiddleware* component and *OrionTopicManager* component will be generated from the *M2T* transformation.

B. KNOWING THE INTERACTIONS OF THE INTERNAL COMPONENTS IN ORDER TO INTEGRATE FIWARE ARTEFACTS

In order to deploy the IoT environments of SimulateIoT on FIWARE, it is necessary to define the relationship and interaction between components. Thus, this section describes these relationships or interactions that allow the deployment of IoT environments on FIWARE.

- **IoTAgent-Json and *Mosquitto* Broker**. IoTAgent-Json needs to receive the messages published in the *Mosquitto*'s Topics from sensors to send them to *Orion Context Broker*.
- **Orion Context Broker and IoTAgent-Json**. *Orion Context Broker* can receive in NGSI protocol the messages published by the *Sensors* because *IoTAgent-Json* is able to act as a bridge between *Sensors* and the *Orion Context Broker*.
- **Orion Context Broker and MongoDB**. Firstly, MongoDB is a functional dependency of *Orion Context Broker*. Thus, *Orion Context Broker* needs to interact with MongoDB to manage all the context data and the devices. Figure 5 shows a sequence diagram illustrating the interactions described up to this point which includes *IoT-Agent, Sensors, Mosquito, Orion Context Broker and MongoDB*.
- **Orion Context Broker and CEP Perseo**. *CEP Perseo* needs to interact with *Orion Context Broker*. Specifically, *CEP Perseo* subscribes to *Orion Context Broker* data and it is able to apply event pattern analysis to them.
- **Perseo Core and Perseo-FrontEnd**. *CEP Perseo* is composed of two components which need to interact

TABLE 1. Relationships among the main metamodel elements with the main target components.

Metamodel Element	*SimulateIoT* Components	Description	FIWARE Components	Description
ProcessNode	Mosquitto	MQTT Broker	*Mosquitto*	MQTT Broker
			MQTT client	Publish/Subscribe on topics
	MQTT Client (internal component)	Publish/Subscribe on topics	*Orion* Context Broker	Device and Context data management
			IoTAgent-Json	Bridge between MQTT-Json and NGSI
Component Database	MongoDB client	MongoDB management	*Orion* Context Broker	*Orion* has a client to interact with MongoDB
	MongoDB	NoSql Database	MongoDB	NoSql Database. A dependency of *Orion* Context Broker. Due to the above fact, the *ComponentDatabase* is no longer an optional component
Component Process Engine	CEP Engine	Apply rules to topics	Perseo	Perseo-FrontEnd: Subscribe to *Orion* context data and Publish notifications (HTTP), Perseo-Core: Apply rules to *Orion* context data
	MQTT client	Subscribe on topics, Publish notifications on topics		

with each other. Basically, *Perseo Core* is the CEP engine which applies rules to the data and notifies *Perseo-FrontEnd*. *Perseo-FrontEnd* is the component that gets the data from *Orion Context Broker* and sends them to *Perseo Core*.

- **CEP Perseo and *MiddlewareNotificationComponent*.** *CEP Perseo* only sends its notifications through the HTTP protocol, however, in the SimulateIoT code generation, the *Actuators* can only receive it through the MQTT protocol.
  MiddlewareNotificationComponent is an additional component developed that listens to *CEP Perseo* notifications and redirects them to *Actuators* through MQTT. Figure 6 shows a sequence diagram that illustrates the interactions described up to this point. Note that in Figure 6 data start from *Orion* to *Perseo Front-end*. The elements involved in this sequence messages includes *PerseoFrontEnd, Orion, PerseoCore, MiddlewareNotificationComponent, Mosquito and Actuators* elements.
- **OrionTopicManager and Orion Context Broker.** In order to ensure the application of rules based on Topics carried out by *CEP Perseo*, *OrionTopicManager* resends all the messages of a concrete Topic to *Orion Context Broker*. Figure 7 shows a sequence diagram illustrating the interactions described up to this point. The elements involved in this sequence messages includes *Orion Topic Manager, Sensors, Mosquito and Orion elements.*

At this point, the steps needed to integrate *FIWARE* components and SimulateIoT artefacts in order to be successfully deployed has been explained. Next, the *M2T* transformation and the specific IoT environment deployment characteristics are described.

V. IoT ENVIRONMENT CODE GENERATION AND DEPLOYMENT

This section describes the main characteristics of the *M2T* transformation and deployment phase based on the analyses about how to integrate *FIWARE* components and artefacts defined on a SimulateIoT model (Section IV-B).

A. MODEL-TO-TEXT TRANSFORMATION

Once the models have been defined and validated, an *M2T* transformation is able to generate the *IoT* environments that have been modelled for a specific technology. Thus, the generated software includes *FIWARE* components such as

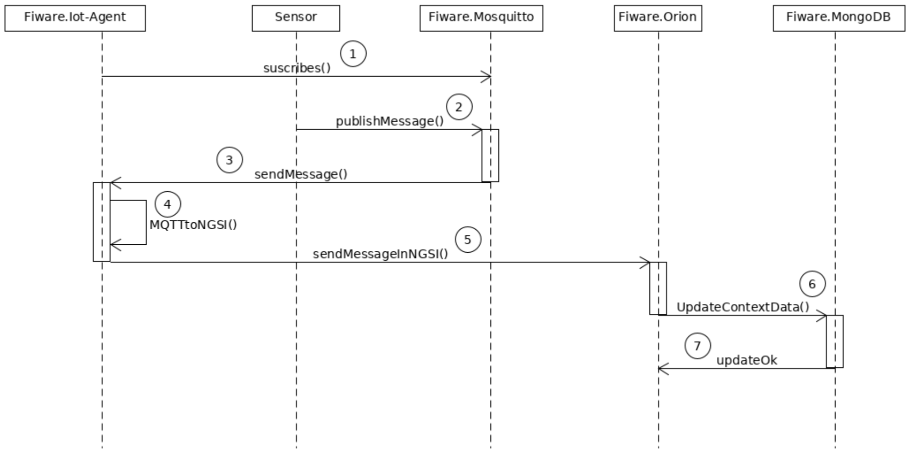


FIGURE 5. ***Orion Context Broker*** **updated from** ***Sensors*** **by means of the IoT-Agent.**

TABLE 2. Available code generation for each different kind of node defined from a SimulateIoT model.

Component	EdgeNode	FogNode	CloudNode
NoSQL DB	None	Required ∣ L	Required ∣ XL
DB client	None	Required ∣ XS	Required ∣ S
REST API	None	Optional ∣ M	Optional ∣ L
MQTT Broker	None	Required ∣ S	Required ∣ M
MQTT Client	Required ∣ XS	Required ∣ XS	Required ∣ S
Perseo	None	Optional ∣ M	Optional ∣ L
Orion	None	Required ∣ M	Required ∣ XL
IoTAgent	None	Required ∣ S	Required ∣ M
NotificationMiddleware	None	Optional ∣ S	Optional ∣ M
OrionTopicManager	None	Optional ∣ S	Optional ∣ M

Orion Context Broker, *CEP Perseo* or *IoTAgent*, and others components can also be generated like an *MQTT* messaging broker, device infrastructure, databases, a graphical analysis platform, docker container, a *REST* API etc. These *FIWARE* components can be deployed as a part of *Node* elements defined on a *SimulateIoT* model. In this regard, Table 2 summarises for each *Node* type the components that can be generated and deployed including *NoSQL* database, *DataBase* Client, *REST* API, *MQTT Broker*, *MQTT Client*, *Orion Context Broker*, *CEP Perseo*, *IoTAgent*, *NotificationMiddleware* or *OrionTopicManager*. Besides, hardware requirements are included with each component. These hardware requirements indicate the minimum hardware power needed to deploy each component of *Cloud*, *Fog* or *Edge* node. The hardware power is represented with the following labels: *XS*, *S*, *M*, *L*, *XL*, where *XS* represents the lowest hardware requirements, for instance, a *RaspBerry Pi*, and *XL* represents the highest hardware requirements, for instance, a cloud infrastructure.

In addition to these components, the *M2T* transformation also generates all the configuration files required to deploy all the artefacts successfully. These configuration files include:

- The registration in *Orion* for each device. An excerpt of the device registry file configuration in *Orion Context Broker* can be seen in Appendix B
- The specification of CEP Perseo's rules. A fragment of the file to configure *CEP Perseo* can be seen in Appendix C.
- The connection of each component with the others, which is managed with a docker-compose file. An example of the *docker-compose* file is illustrated in Appendix A.
- The deployment scripts needed to deploy the artefact generated.

B. IoT ENVIRONMENT DEPLOYMENT ON FIWARE INFRASTRUCTURE

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, *Orion*, *Perseo*, *IoTAgent*, etc. are configured and deployed.

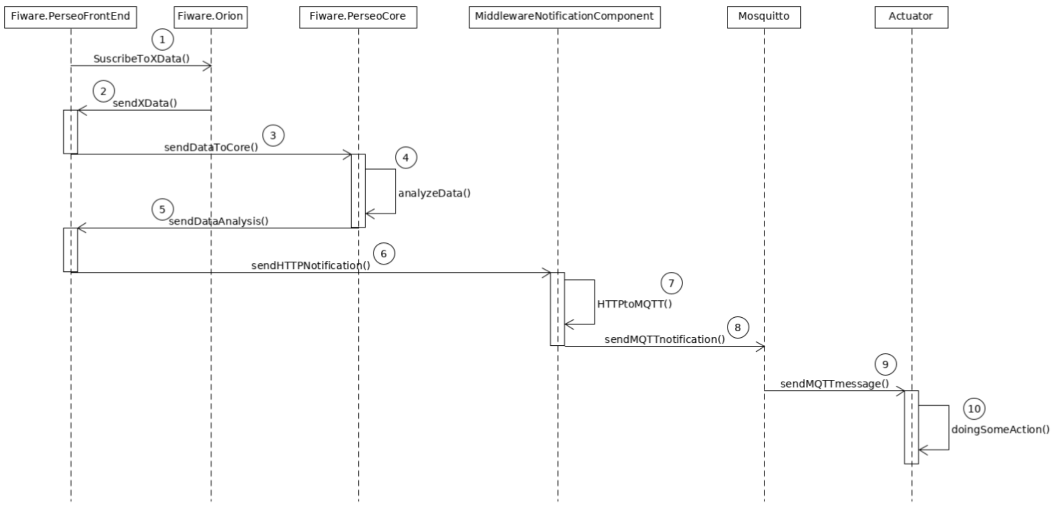


FIGURE 6. Managing *CEP Perseo* notifications to notify the event patterns detected to the *Actuators*.

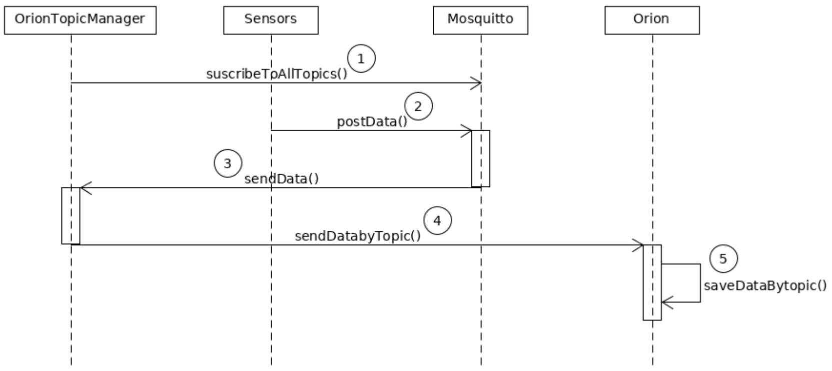


FIGURE 7. OrionTopicManager operation and interactions.

It should be noted that, although it is a simulation, the deployment of the Fog and Cloud layers of the environment is a real deployment, in other words, the architecture generated could be implemented in a real IoT environment. However, the devices of the Edge layer (sensors and actuators) are fully simulated, interacting with the rest of the layers publishing data, imposing the pace of the simulation (speed of data generation), connecting to and disconnecting from different nodes in the Fog layer (displacement), receiving notifications processed in the Fog or Cloud layer (actuators), etc.

Figure 2 shows the architecture of a Smart Building environment where it is possible to observe the different elements that can be deployed including a *CloudNode* or *FogNode*, *Sensors* and *Actuators*. Note that *CloudNode* and *FogNode* are composed of several elements, including *FIWARE* elements such as *Orion Context Broker*, *CEP Perseo* or *IoTAgent*.

Furthermore, each *CloudNode*/*FogNode* can define a Complex Event Processing Engine or, in other words, the inclusion of *CEP Perseo*. Besides, it includes *Orion Context Broker*, *IoTAgent-Json*, a Non-SQL database, as MongoDB is essential due to it being a dependency of *Orion Context Broker*, a *DataBase Client*, a REST API, an MQTT broker (e.g *Mosquitto*) and an MQTT Client. Likewise, as can be

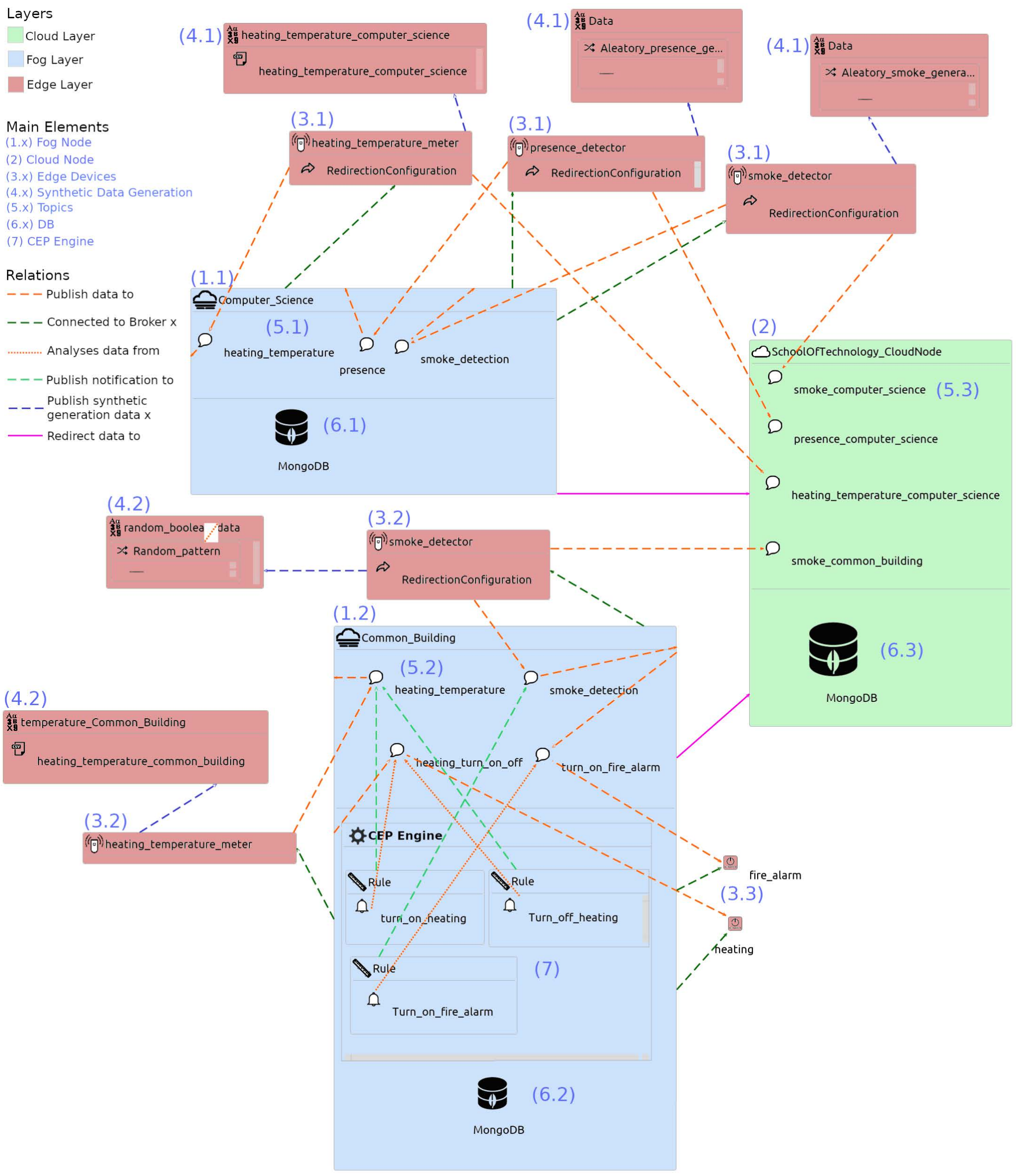


FIGURE 8. Case 01. The school of technology model conforms to the *SimulateIoT* metamodel.

observed in Figure 2, all of these elements are interconnected and are deployed on Docker containers. Specifically, all Docker containers are orchestrated using Docker Swarm.

Finally, along with the device code generated, a *deployment script* is included which contains the necessary instructions for deploying the *IoT* environments. Algorithm 1

Algorithm 1 Deploying an *IoT* Environment Simulation on FIWARE Architecture and Microservices

```
begin
    //Step 1) Compilation and docker wrapping of
        the artefacts.
    while there are components which will have to
        be compiled do
        Compile component
        if component should be in a  docker then
            Wrap component in a docker
        endif
    endwhile

    //Step 2) Pushing of docker images to the
        local registry.
    while there are docker images to push to local
        registry do
        Push image
    endwhile

    //Step 3) Creating the Swarm Cluster.
    create manager node 'Environment'
    foreach fogNode do
        Create worker node 'fogNode.name'
        Connect worker node to the manager node
    endforeach

    //Step 4) Configuration of each Swarm node and
        deploy of the FIWARE components on them.
    foreach worker node do
        Configure node
        Deploy FIWARE components from docker-
            compose-file
        configure FIWARE components from
            configuration files
    endforeach

    //Step 5) Pulling of docker images from the
        local registry in each Swarm node.
    foreach worker node do
        foreach docker image wrapped for this node
            in the local registry  do
            Pull image
        endforeach
    endforeach

    //Step 6) Deployment of the docker components
        as services from the manager node of the
        Swarm cluster.
    Connect to the swarm manager node
    foreach worker node do
        foreach pulled docker image in the worker
            node do
            Deploy docker as service
        endforeach
    endforeach
end
```

shows the deployment phase of an IoT environment on *FIWARE*.

VI. CASE STUDIES

Next, two case studies have been defined using *SimulateIoT*. The first one defines an IoT simulation for a a smart building while the second one defines an IoT simulation for an agricultural environment. Note that these two cases are the same as the ones modelled in [3]. Thus demonstrating that the proposal presented in this paper can deploy these same environments with SimulateIoT-FIWARE on the FIWARE platform, without the need to know any technical aspects about it.

A. CASE 01. SCHOOL OF TECHNOLOGY

The first case study presents the simulation of a smart building, more specifically, we have modelled our School of Technologies. It has six buildings (Computer Science, Civil Works, Architecture, Communications, 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 8 shows an excerpt from the School of Technology model. The IoT system modelled includes several *Node* elements shared throughout the different buildings. Each building takes over its own *ProcessNode* (Figure 8 references 1.1, 1.2 and 2) which gathers all the information produced by the *Sensors* (Figure 8 references 3.1 and 3.2). Thus, these data are suitably stored on specific databases (Figure 8 references 6.1 and 6.2), analysed and monitored by the *ProcessNode* element. In this case study, a *FogNode* element has been defined for each building. For instance, *Common_Building or Computer_Science* have defined *FogNode* elements (Figure 8 references 1.1 and 1.2).

Furthermore, a *CloudNode* named *SchoolTechnologyCloudNode* (Figure 8 reference 2) is defined to store information gathered from the *FogNode* elements. Both *FogNode* and *CloudNode* elements define several *Topic* elements (Figure 8 references 5.1, 5.2 and 5.3) such as *heating_temperature*, *presence* and *smoke_detection*. These *Topic* elements communicate data among the *Node* elements defined in the IoT system.

In order to model the School of Technology case study, several *Sensors* such as *heating_temperature_meter*, *presence_detector*, *smoke_detector* and so on have been defined in Figure 8. Each of them publishes its own data on a specific *Topic* element. As can be observed in Figure 8, the *Sensor* elements publish data to several *FogNode* through *Topic* elements.

Note that *Sensor* elements are *EdgeNode* elements that generate data, so the data pattern generators should be defined (Figure 8 references 4.1 and 4.2). For instance, in order to describe the synthetic data generated by a temperature sensor 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 from the external tools such as [1], [19].

As mentioned, in Figure 8 each *FogNode* has its own characteristics about how data should be managed including storing, analysing or addressing. For instance, the *ComputerScience FogNode* element addresses the information every *thirty seconds*, storing the data obtained in a specific NoSQL database. Then all data are flushed to the next node *FogNode or CloudNode* defined in the architecture and named in the example *SchoolTechnologyCloudNode*.

On the other hand, the *Common_Building FogNode* element 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 and several *Rule* elements (Figure 8 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 a specific *Actuator* named *heating* (Figure 8 reference 3.3. 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 *CEP Engine* by means of the *rule_heating* Rule. As a consequence, when the defined pattern is matched (for instance, *if (temperature < 20) then switch on heating*), the CEP engine generates an event to *turn_on_off_heating* Topic.

2) CASE 01. CODE GENERATION AND DEPLOYMENT

Once the model has been defined, the *M2T* transformation is applied with the following goals: i) to generate Java code that wraps each device behaviour; ii) to generate configuration code to deploy based on *FIWARE* components. These files include the code necessary to register all devices in *Orion Context Broker* and the code required to define all rules in *CEP Perseo*. iii) to generate configuration code to deploy the message brokers necessary (and connect them with FIWARE), including the *topic* configurations defined; iv) to generate for each *ProcessNode* and *EdgeNode* a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm* v) to generate the Swarm cluster to deploy the simulation in orchestrated mode.

Figure 9 shows a simplified excerpt from the School of Technology IoT model deployed (full version available in Figure 12) 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. Thus, each *EdgeNode* and *ProcessNode* element carries out its own functions such as sending messages, processing and storing messages, acting from messages, etc.

Additionally, 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 the cloud. Not only is the simulation code generated, but also the final IoT system code is partially generated.

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. The analysis that can be carried out is fundamentally based on the log behaviour of each node within the simulation. This log behaviour includes parameters such as: i) Each component performs its functions successfully, such as publishing, receiving, analysing, redirecting data, etc. ii) The resources used by each component, such as CPU or Memory usage iii) The general function of the IoT architecture modelled, in other words, if the IoT environment is satisfying the user needs or requirements iv) The evolution of the above-mentioned parameters over time.

In this sense, users using the simulation logs, could evaluate the behaviour of the environment by exposing it to different levels of stress by experimenting with different number of devices, size of published messages, publication periods, etc. and study parameters such as a) jitter between messages, checking in mongodb the timestamps of the messages of a sensor, b) response delay of a particular component, for instance, checking the CEP engine logs it can be seen when a rule is met and when the notification is sent to the actuator, c) packet loss rate, checking the difference of number messages between the messages published by a sensor (sensor logs) and the messages stored in MongoDB from that sensor, etc.

In short, users can carry out different experiments by creating different models and simulating them, thus determining which aspects can be improved until the version that meets his requirements is achieved.

B. CASE 02. AGRICULTURAL ENVIRONMENT

This case study focuses on designing an IoT system for managing irrigation and weather data 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 [49], [50]: i) Collection of weather, crop and soil information; ii) Monitoring of distributed land; iii) Multiple crops on a single piece of land; iv) Different fertiliser and water requirements for different pieces of uneven land; v) Diverse requirements of crops for different weather and soil conditions; vi) Proactive solutions rather than 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*

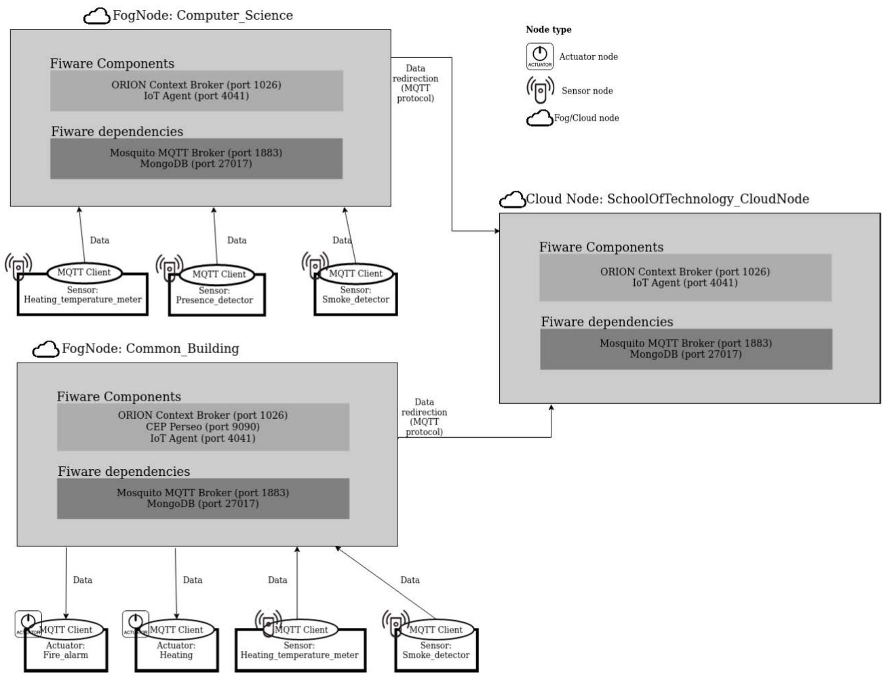


FIGURE 9. Case 01. Deployment of the school of technology IoT model (Simplified version, full version available in Figure 12).

element which is gathering data and re-sending them if 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 messages by *Topic* elements.

1) CASE 02. MODEL DEFINITION

In Figure 10 an excerpt from an IoT model conforming to the *SimulateIoT-FIWARE* metamodel is defined. It shows different *Sensor* elements such as (*ph_H1, temperature_H1, Humidity_H1, etc.*) which generate data for simulation (Figure 10 references 3.1 and 3.2). Moreover, several Fog computing nodes have been defined, although in Figure 10 (for the sake of simplicity) only two *FogNode* elements are shown (Figure 10 references 1.1 and 1.2). They define several *Topics* such as *Humidity, Temperature, pH, Water_pressure, etc* (Figure 10 references 5.1 and 5.2). In addition, each *FogNode* element defines a CEP engine by means of *Perseo* elements (Figure 10 references 7.1 and 7.2). 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 10 references 3.1) is activated when the *Notification* element named *Low_Humidity* is thrown.

2) CASE 02. CODE GENERATION AND DEPLOYMENT

Once the model has been completed and validated, an *M2T* transformation is carried out obtaining the simulation code, which can be deployed on a specific platform, specifically using FIWARE components.

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. It is worthy of mention that one component could have a complex architecture and be defined in several microservices. In consequence, these kinds of components will be wrapped in several *Docker* containers (each defined microservice in a container, as for example in the case of *FogNode* and *CloudNode* components). Figure 11 shows a simplified excerpt from the case study deployment architecture (full version available in Figure 13). In Figure 11 the main characteristics of each node can be observed. For instance, each *ProcessNode* defines an *Orion Context Broker* with its *MongoDB* database, an *IoTAgent*, a *Mosquitto MQTT* message broker and a *CEP*

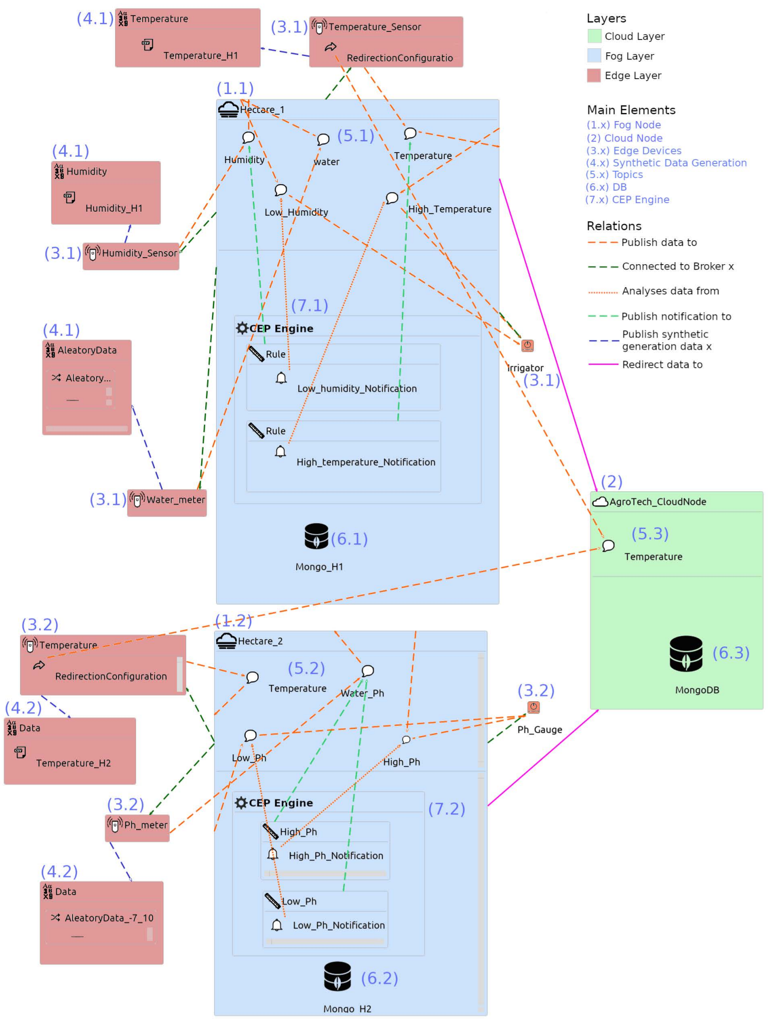


FIGURE 10. Case 02. AgroTech model conforming to the *SimulateIoT* metamodel.

Perseo engine. In addition, the *Rule* elements defined are processed through the *CEP Perseo* engine defined.

Each *ProcessNode* element deployed on a *Docker* container has its own characteristics:

- *CloudNode*, named *AgroTe_CloudNode*, is composed of an *Orion Context Broker* with its *MongoDB* [30], an *IoT Agent* and a message-driven broker like *Mosquitto* (that implements an *MQTT* communication protocol). Moreover, the *CloudNode* deploys a *Compass* instance [7] to monitor the data gathered.
- Each *FogNode* named *Hectare_1* and *Hectare_2* respectively, is composed of an *Orion Context Broker* with its *MongoDB* [30], an *IoT Agent*, a message-driven broker like *Mosquitto* (that implements an *MQTT* communication protocol) and a *Perseo* engine. *MongoDB* stores the temporal data gathered by the *FogNode* instance. Currently, the main difference between a *CloudNode* and *FogNode* is the processing capability. Using the *size* attribute at the *FogNode* element makes it possible to define the process capabilities. Consequently, both *CloudNode* elements and *FogNode* elements are deployed as Docker containers on hardware nodes such as PC, VM or Raspberry Pi.
- The *CEP* characteristic defined at *ProcessNode* deploys a complex event processor to process high amounts of messages in real-time. As can be observed in Figure 11 a *CEP Perseo* engine is deployed on each *FogNode*. Later on, each *CEP Perseo* engine analyses

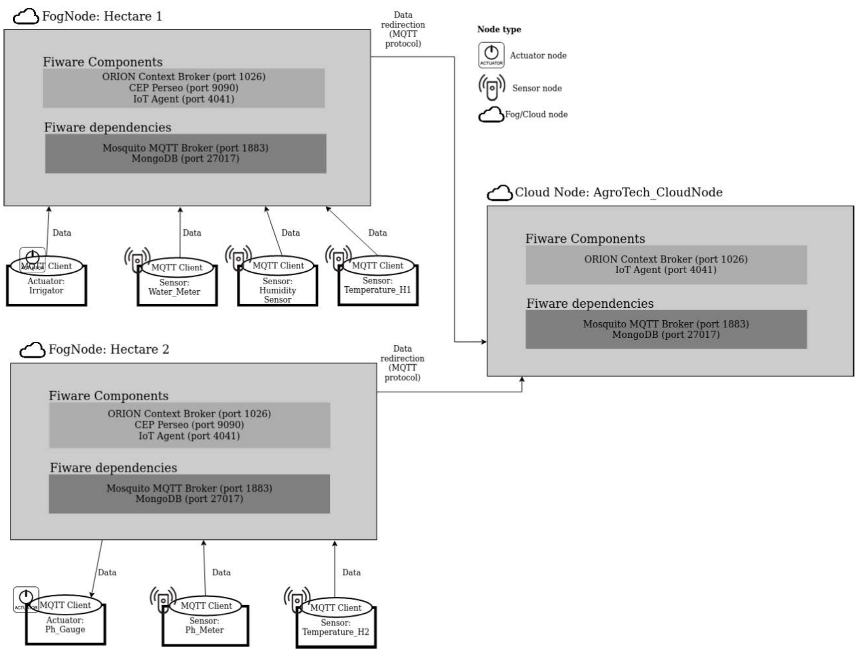


FIGURE 11. Case 02. Agrotech deployment architecture (Simplified version, full version available in Figure 13).

the incoming events by the *Rule* elements associated with it.
- The *EdgeNode* elements including *Sensors* and *Actuators* defined in the model are suitably deployed in Docker containers.

Later on, the execution information can be audited 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 that gathers the available endpoints of each element, for example, to query a MongoDB database or to show information about a *Mosquitto* broker. In the above-mentioned ways, it is possible to perform the analysis of the environment mentioned at the end of Section VI-A.

VII. DISCUSSION

Model-driven development can be used to model complex IoT environments using domain concepts. They need not be tied to a specific technology, but rather an *M2T* transformation makes it possible to generate the code needed to deploy and simulate the systems.

The technology used as a target, such as FIWARE, microservices (Thorntail), containers (Dockers), message-oriented middleware, MQTT (Mosquitto) or a container orchestrator (Docker Swarm) can be quickly replaced by other suitable technology if needed. Of course, to change the target technology, an *M2T* transformation should be implemented.

For the reasons mentioned above, it has been considered to give *FIWARE* an added value through integration with *SimulateIoT*. The result of this integration is *SimulateIoT-FIWARE*, which is able to model and generate an IoT environment with FIWARE artefacts. In short, *SimulateIoT-FIWARE*, the resulting tool from the integration of *FIWARE* and *SimulateIoT* based on Model-Driven development, define an abstraction layer that allows the use of *FIWARE* artefacts without the need to know how these components work internally, that is, how they interact with each other and with the other components of an environment, how their deployment is configured, how they are configured to work in the environment, etc. with the final purpose of generating and deploying an IoT environment powered by *FIWARE*.

Finally, the target users could be both: a) professional users and b) students. Professional users could use the methodology and tools presented in this work to define and analyse complex IoT environments where finally heterogeneous technology is used, even though the core comprises components provided by FIWARE. 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 study the code generated to learn the technology used to deploy the IoT system. Thus, they can understand IoT cutting-edge technology such as FIWARE, edge technology and integration patterns such as data patterns, IoT characteristics, publish-subscribe communication protocols, MQTT, containers, NoSQL databases, distributed systems and so on.

A. LIMITATIONS

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

- The Edge Nodes can be defined as mobile nodes by using several approaches (FogCloudRoute, LinearRoute and RandomRoute). However, IoT mobility is a wide and interesting research area where multiples protocols and mobility mechanisms could be additionally defined.
- For the sake of simplicity, the current version of our simulator IoT environment for *FIWARE* allows defining connected nodes by TCP/IP, and it is assumed 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 taking hardware restrictions into account.

VIII. RELATED WORK

At this point, several Model-Driven Development approaches have been defined to manage IoT complexity, however, there are no MDD approaches focused on generating code for well-know or global IoT platforms such as FIWARE. Next, additional MDD approaches related to IoT environment definition are analysed. Next, the main Model-Driven approaches to generate IoT systems are reviewed.

FRASAD [33] is a model-driven software development framework to manage the complexity of the Internet of Things (IoT) applications. *FRASAD* is based on node-centric software architecture and a rule-based programming model that allows designers to describe their applications (IoT environments).

An application within *FRASAD* could include several sensors with multiple characteristics. For instance, some sensors could publish temperature, others could receive it and, if rules are specified, the centric-node will apply the rules to this temperature data and from the result of the analysis, modify the behaviour of the sensors.It is worthy of mention that, regardless of the characteristics chosen, within *FRASAD* all devices are *sensors*, there is not a hierarchy of devices. In addition, each sensor could have multiple inputs and outputs. Although *FRASAD* provides multiple options to model sensors, users cannot choose the target technology, for example, to apply rules, to communicate each sensor (communication protocol), to store data, etc.

MDE4IoT [6] is a *Model Driven Engineering* [41] approach that allows the modelling of IoT components and supports intelligence as self-adaptation of Emerging Configurations in the *IoT*. Within *MDE4IoT* they call Emergent Configuration (EC) of connected systems a set of things/devices with their functionalities and services that connect and cooperate temporarily to achieve a goal. In short, *MDE4IoT* allows users to define an *IoT* environment that is able to adapt the behaviour of its devices at run-time. For instance, *MDE4IoT* could define and generate an *IoT* environment where several inter-connected *Smart-Lamps* adapt their behaviour (light colour, brightness, etc.) depending on the traffic flow or other environmental data such as car speed, the distance between cars, natural light, etc. *MDE4IoT* allows users to define hardware and software characteristics of a device, being able to define a sensor, an actuator or another kind of device, of course, each with its own characteristics. However, *MDE4IoT* does not allow users to choose the technology they want to use, for instance, the users cannot choose the database, the rule engine (to manage the EC), the communication protocol, etc. On the other hand, *MDE4IoT* generates the code to be implemented in the physical devices of the environment, not allowing a simulation of it. Additionally, a global target such as *FIWARE* is not available.

Another approach such as [38] proposes a model-driven software development framework that allows users to model *IoT* environments with several types of devices with many modelling features. It proposes that the stakeholders could add features to the framework. These stakeholders are: 1) The sensor Manufacturer/sensor Provider, who could add device features such as device drivers, data models or device interfaces, 2) The Algorithm expert/Algorithm developer who defines algorithm features as CPU/Memory requirements, performance or accuracy, 3) The Domain Expert who manages the model requirements or the mapping of the algorithms to the sensors and 4) The System Administrator who could add features such as CPU/Memory availability or the calculations of the network characteristics through devices and the cloud. In this way, these four stakeholders could develop a powerful framework to generate *IoT* environments, however, although the abstraction layer to develop *IoT* environments has been incremented with this framework, the user needs to know several concepts about the domain of these four stakeholders. For instance, this framework incorporates many algorithms that can be added to devices, in this way, the user does not need to know how to implement the algorithms, but they need to know how they work because several algorithms could do the same thing in different ways, and the user needs to know which one best fits their needs and requirements. The above-mentioned example can be extrapolated to the other features which could be modelled with this framework. In short, due to the low abstraction layer that this framework provides, the users need to be experts in the IoT and all the concepts around it, such as the hardware used, algorithms or the IoT domain. In [38] an initial prototype had been developed to cover some of the aforementioned aspects. In addition, the IoT applications defined using this framework are deployed using their own implementation. Consequently, they do not use a global IoT architecture such as *FIWARE* as a target.

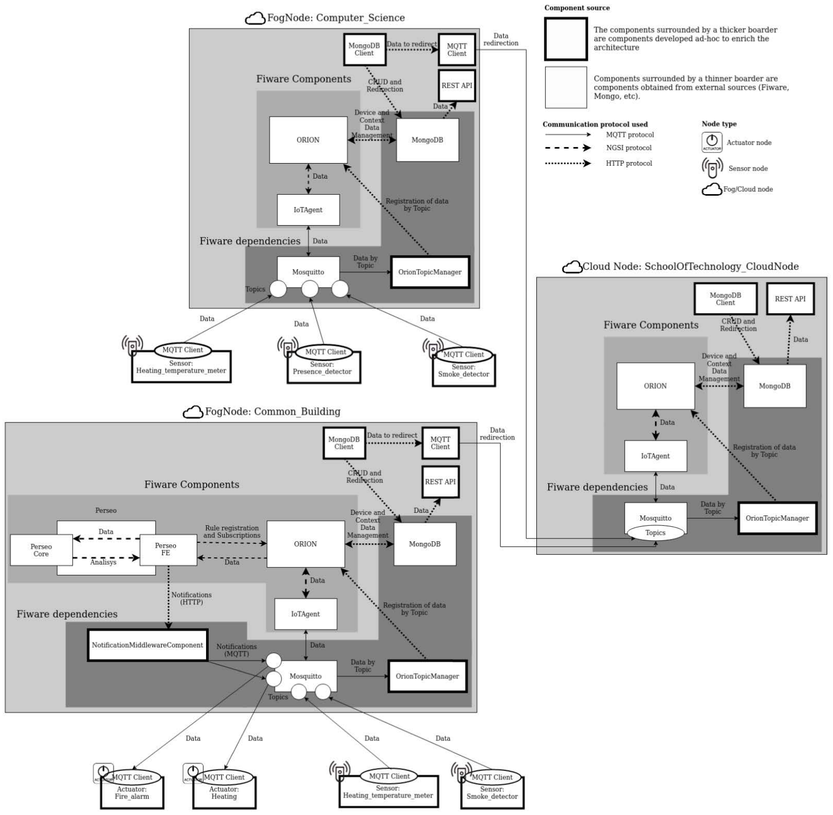


FIGURE 12. Case 01. Deployment of the school of technology IoT model (Full version, simplified version available in Figure 9).

Fog computing is proposed to solve the latency problems of the services offers by the Cloud. Nevertheless, to realise the full potential of Fog and IoT paradigms, it is necessary to design resource management techniques that determine which modules of analytics applications are pushed to each edge device to minimize the latency and maximse the throughput [18].

In [18] *iFogSim* an *IoT* simulator is proposed that enables the quantification of the performance of resource management policies on an IoT or Fog computing infrastructure in a repeatable manner. This simulator can measure performance in four different areas: latency, network congestion, energy consumption, and cost.

iFogSim allows users to model an *IoT* environment with several nodes such as *Sensors*, *Actuators*, *Fog* devices, etc. with different nodes or environmental properties such as 1) Hardware characteristics: accessible memory, processor, storage size, uplink, and downlink bandwidths, 2) Network characteristics: connectivity among devices, latency, network congestion, etc. 3) Data characteristics: Data flow, type of data, etc. among other devices or environment properties.

iFogSim is a simulator that, because of the great capacity of expression that it possesses, can simulate very similar environments to a real one. In consequence, the users that employ this tool need to skillfully manage a lot of concepts about *IoT*, *Networking*, *Fog and Cloud* paradigms, etc. Due to the above-mentioned aspect, this tool is recommended for expert users, and may not be the best option to some purposes or targets such as education, novel users in *IoT* or engineering, small *IoT* environments such as a domotic house, or *IoT* environments that do not need to use *Fog* computing.

MobIoTSim [39] is an *Android IoT* simulator which aims to help Cloud application developers to create *IoT* environments with several devices without buying real sensors. In this way, *MobIoTSim* allows the simulation of *IoT* environments where developers can learn, test and demonstrate *IoT* applications, which works with *IoT* Cloud providers such as *Bluemix* [21] or *Google IoT Platform*, in a fast and efficient way.

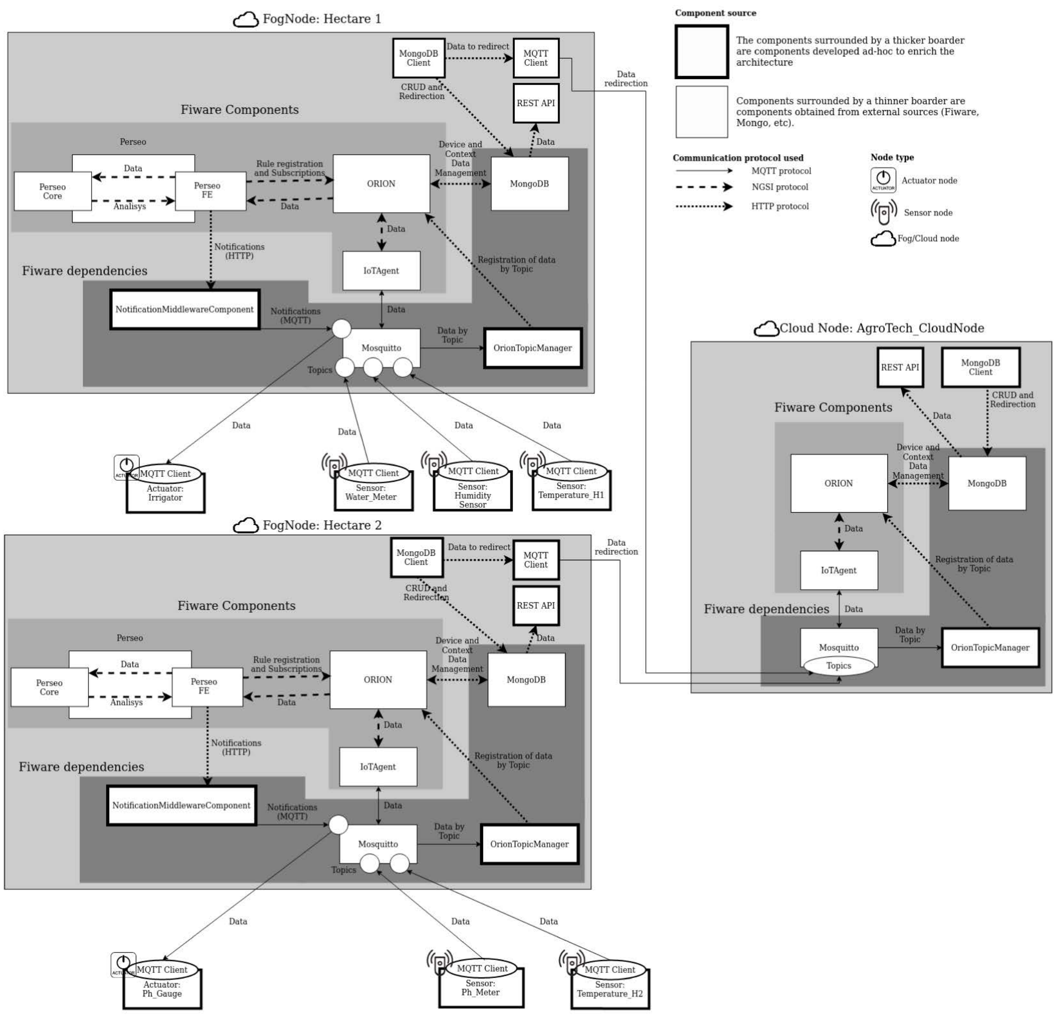


FIGURE 13. Case 02. Agrotech deployment architecture (Full version, simplified version available in Figure 11).

MobIoTSim allows, from an Android device, the configuration of several *IoT Cloud* gateways to allow connection with the devices. Add, edit or delete devices that sends (MQTT and JSON) random data in a range defined by the user with a frequency. In addition, devices can receive data, for instance, in [39] *Bluemix* is configured to send warning notifications to devices if it detects critical values. Besides, *MobIoTSim* provides the display of the data published by the simulated devices.

In [32], the authors make an in-depth analysis of the state of the art of deployment and orchestration in IoT environments. Additionally, the authors have developed a taxonomy of DEPO4IOT to classify, analyse, and compare the studies. This taxonomy takes into account factors such as the deployment and orchestration support, design support and other advanced supports. Our proposal takes into account the importance of deployment and orchestration of the IoT environment, including the possibility of deploying them in Docker containers and orchestrating them using Docker Swarm, also generating a deployment script from the models defined by the users, where all the necessary parameters are automatically configured to carry out a reliable deployment.

The expressiveness of *SimulateIoT* Domain Specific Language determines the expressive capacity of *SimulateIoT-Fiware*. Therefore, simulation of specific aspects of particular IoT environments require developing an ad-hoc extension.

So, *SimulateIoTFiware* requires additional changes to model IoT environments focus on particular aspects such as : a) *ThinORAM* [20], a lightweight client-side ORAM system that substantially improves response time and network usage respect the existing ORAM systems in literature; b) The approach conducted in [34], a descentralised Blockchain-based architecture to manage the roles and permissions of the IoT devices of an IoT environment; c) *ProfilIoT* [28], a machine learning approach that, from the traffic generated by a device on the network, is able to determine whether it is an IoT device (and what kind of IoT device it is) or not.

Although *SimulateIoTFiware* requires additional changes to model an IoT environment with *ThinORAM* [34] or *ProfilIoT* [28], it has the expressive capability to model the IoT architecture of FIWARE-based Edge, Fog and Cloud nodes.

To sum up, although there are several approaches focused on rising the abstraction level from the IoT applications that can be developed, there is a lack of approaches to carry out this process using a global IoT infrastructure as the target. The present proposal is tailored to *FIWARE* technology, allowing modelling large IoT projects which can be deployed later on using this IoT platform.

IX. CONCLUSION

Model-driven development techniques are a suitable way to tackle the complexity of domains integrating heterogeneous technologies. Initially, they focus on modelling the domain, then, by using *M2T* transformations, the code for specific technology could be generated. *SimulateIoT* takes advantage of this technology to allow the modelling and the generation of IoT environments.

Moreover, *FIWARE* has a large catalogue with several components oriented to the *IoT* which allow the development of complex *IoT* architectures. Besides, *FIWARE* is a popular open-source project,and as a consequence, *FIWARE* has great support and its components are highly tested.

Both technologies (SimulateIoT and FIWARE) have been integrated. In this way, the resulting tool allows users to define and validate models conforming to the *SimulateIoT* metamodel. Then, an *M2T* transformation makes it possible to generate the *FIWARE* components needed to deploy the *IoT Simulation* defined.

Future projects include new concepts taking into account the *FIWARE* catalogue. For instance, components such as Cosmos, which enables an easier BigData analysis, FogFlow, to support dynamic processing flows over cloud and edges, or Knowage, which brings a powerful Business Intelligence platform enabling users to perform business analytics over traditional sources and big data systems. Other interesting further work includes the improvement of the *SimulateIoT* components which cannot be replaced by *FIWARE*, for instance, the *Sensors*, which could be improved by defining and generating new kinds of data generation patterns. Finally, it is expected to explore the code generation to other IoT platforms such as Google Cloud's IoT Platform [17], Microsoft Azure IoT suite [23], ThingSpeak IoT Platform [47] or Thingworx 8 IoT Platform [48].

APPENDIX A

```
orion:
  image: FIWARE/orion:2.0.0
  hostname: orion
  container_name: FIWARE-orion
  depends_on:
    - mongo-db
  expose:
    - "1026"
  ports:
    - "8082:1026"
  ....
iot-agent:
  image: FIWARE/iotagent-json
  hostname: iot-agent
  container_name: FIWARE-iot-agent
  depends_on:
    - mongo-db
    - Mosquitto
  expose:
    - "4041"
  ports:
    - "4041:4041"
  environment:
    - IOTA_CB_HOST=orion
    - IOTA_CB_PORT=1026
    - IOTA_NORTH_PORT=4041
    - IOTA_REGISTRY_TYPE=mongodb
    - IOTA_MONGO_HOST=mongo-db
    - IOTA_MONGO_PORT=27017
    - IOTA_MONGO_DB=iotagent-json
    - IOTA_MQTT_HOST=mosquitto
    - IOTA_MQTT_PORT=1883
    - IOTA_PROVIDER_URL=
    http://iot-agent:4041
  ....
mongo-db:
  image: mongo:3.6
  hostname: mongo-db
  container_name: db-mongo
  expose:
    - "27017"
  ports:
    - "27017:27017"
  \ldots.
perseo-core:
  image: FIWARE/perseo-core
  environment:
    - PERSEO_FE_URL=http://perseo-fe:9090
    - MAX_AGE=6000
  depends_on:
    - mongo-db
  environment:
    - PERSEO_FE_URL=http://perseo-fe:9090
    - MAX_AGE=6000
 ....
perseo-fe:
  image: FIWARE/perseo
  ports:
    - 9090:9090
  depends_on:
    - perseo-core
  environment:
    - PERSEO_MONGO_ENDPOINT=mongo-db
    - PERSEO_CORE_URL=
    http://perseo-core:8080
    - PERSEO_LOG_LEVEL=debug
    - PERSEO_ORION_URL=http://orion:1026/
  ....
  Mosquitto:
    image: eclipse-mosquitto
    hostname: Mosquitto
    container_name: Mosquitto
    expose:
      - "1883"
      - "9001"
    ports:
      - "1883:1883"
      - "9001:9001"
    ....
```

APPENDIX B
CODE FRAGMENT TO CONFIGURE ORION

```
curl -iX POST \
  'http://localhost:4041/iot/devices' \
  -H 'Content-Type:␣application/json' \
  -H 'FIWARE-service:␣openiot' \
  -H 'FIWARE-servicepath:␣/' \
  -d '{
␣"devices":␣[
␣␣␣{
␣␣␣␣␣"device_id":␣␣␣"
    Sensor_Heating_Temperature_meter_5",
␣␣␣␣␣"entity_name":
␣␣␣␣␣"urn:ngsi-ld:Sensor_Heating_Temperature_meter
    :5",
␣␣␣␣␣"entity_type":␣"
    Sensor_Heating_Temperature_meter",
␣␣␣␣␣"protocol":␣␣␣␣"JSON",
␣␣␣␣␣"transport":␣␣␣"MQTT",
␣␣␣␣␣"timezone":␣␣␣␣"Europe/Berlin",
␣␣␣␣␣"attributes":␣[
␣␣␣␣␣␣␣{␣"object_id":␣"v",␣"name":␣"value",
␣␣␣␣␣␣␣"type":␣"Integer"␣}
␣␣␣␣␣],
␣␣␣␣␣"static_attributes":␣[
␣␣␣␣␣␣␣{␣"name":"name",␣"type":
␣␣␣␣␣␣␣"String",␣"value":␣"
    Sensor_Heating_Temperature_meter"}
␣␣␣␣␣]
␣␣␣}
␣]
}'
```

APPENDIX C
CODE FRAGMENT TO CONFIGURE PERSEO

```
curl -iX POST 'http://localhost:9090/rules'  -H '
    FIWARE-service:␣openiot'   -H 'FIWARE-
    servicepath:␣/'  -H 'Content-Type:␣application
    /json' -d '{
␣␣"name":␣"rule0_sensor_heating_temperature_meter
    ",
␣␣"text":"select␣*,\"
    rule0_sensor_heating_temperature_meter\"␣as
␣␣ruleName␣from␣pattern␣[everyev=iotEvent
␣␣(cast(cast(value?,String),float)>25␣and
␣␣id=\"urn:ngsi-ld:Topic_heatingtemperature:0\")
    ]",
␣␣"action":␣{
␣␣␣␣␣␣␣␣"type":␣"post",
␣␣␣␣␣␣␣␣"template":␣"{\"value\":\${value}}",
␣␣␣␣␣␣␣␣"parameters":␣{
␣␣␣␣␣␣␣␣␣␣␣␣"url":␣"
␣␣␣␣␣␣␣␣␣␣␣␣http://mncseccionte cnologia2
␣␣␣␣␣␣␣␣␣␣␣␣:5150/heating_0",
␣␣␣␣␣␣␣␣␣␣␣␣"headers":␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"Content-type":
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"application/json"
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣}
␣␣␣␣}
}'
```

APPENDIX D
COMPLETE USE CASE DEPLOYMENT ARCHITECTURE

See Figures 11 and 12.

REFERENCES

[1] J. W. Anderson, K. E. Kennedy, L. B. Ngo, A. Luckow, and A. W. Apon, ''Synthetic data generation for the Internet of Things,'' in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 171–176.

[2] C. Atkinson and T. Kühne, ''Model-driven development: A metamodeling foundation,'' *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, Sep. 2003.

[3] J. A. Barriga, P. J. Clemente, E. Sosa-Sanchez, and A. E. Prieto, ''SimulateIoT: Domain specific language to design, code generation and execute IoT simulation environments,'' *IEEE Access*, vol. 9, pp. 92531–92552, 2021.

[4] T. Bass, ''Mythbusters: Event stream processing versus complex event processing,'' in *Proc. Inaugural Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2007, p. 1.

[5] R. 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,'' *Softw., Pract. Exp.*, vol. 41, no. 1, pp. 23–50, 2011.

[6] F. Ciccozzi and R. Spalazzese, ''Mde4iot: Supporting the Internet of Things with model-driven engineering,'' in *Proc. Int. Symp. Intell. Distrib. Comput.* Cham, Switzerland: Springer, 2016, pp. 67–76.

[7] (2018). *MongoDB Compass*. [Online]. Available: https://www.mongodb.com/products/compass

[8] G. Cugola and A. Margara, ''Processing flows of information: From data stream to complex event processing,'' *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–62, 2012.

[9] EsperTech. (Nov. 2016). *Esper Cep*. [Online]. Available: https://www.espertech.com/esper/

[10] Fiware. (2019). *Orion Context Broker*. [Online]. Available: https://fiware-orion.readthedocs.io/en/master/#welcome-to-orion-context-broker

[11] Fiware. (2019). *Perseo Architecture*. [Online]. Available: https://perseo.readthedocs.io/en/latest/architecture/architecture/

[12] Fiware. (2019). *Perseo Context-Aware Cep*. [Online]. Available: https://perseo.readthedocs.io/en/latest/#perseo-context-aware-cep

[13] FIWARE. (2021). *Fiware*. [Online]. Available: https://www.fiware.org/about-us/

[14] Fiware. (2021). *Ngsi Protocol*. [Online]. Available: https://knowage.readthedocs.io/en/6.1.1/user/NGSI/README/index.html

[15] E. Fotopoulou, A. Zafeiropoulos, F. Terroso-Sáenz, U. Şimşek, A. González-Vidal, G. Tsiolis, P. Gouvas, P. Liapis, A. Fensel, and A. Skarmeta, ''Providing personalized energy management and awareness services for energy efficiency in smart buildings,'' *Sensors*, vol. 17, no. 9, p. 2054, Sep. 2017.

[16] M. García, ''New businesses around open data, smart cities and fiware,'' Eur. Public Sector Inf. Platform, Tech. Rep. 4, 2015. [Online]. Available: https://data.europa.eu/sites/default/files/report/2015_new_businesses_around_open_data_smart_cities_and_fiware.pdf

[17] Google. (2017). *Google Cloud IoT*. [Online]. Available: https://cloud.google.com/solutions/iot/

[18] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, ''IFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments,'' *Softw., Pract. Exper.*, vol. 47, no. 9, pp. 1275–1296, 2017.

[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.

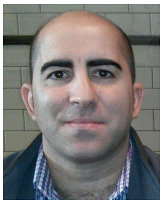
[20] Y. Huang, B. Li, Z. Liu, J. Li, S.-M. Yiu, T. Baker, and B. B. Gupta, ''ThinORAM: Towards practical oblivious data access in fog computing environment,'' *IEEE Trans. Services Comput.*, vol. 13, no. 4, pp. 602–612, Jul. 2020.

[21] IBM. (2014). *IBM Cloud*. [Online]. Available: https://www.ibm.com/cloud/bluemix/

[22] J. Han, E. Haihong, G. Le, and J. Du, ''Survey on NoSQL database,'' in *Proc. 6th Int. Conf. Pervasive Comput. Appl. (ICPCA)*, Oct. 2011, pp. 363–366.

[23] S. Klein, *IoT Solutions Microsoft's Azure IoT Suite*. New York, NY, USA: Springer, 2017.
[24] 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, pp. 1–27, Feb. 2015.
[25] J. A. López-Riquelme, N. Pavón-Pulido, H. Navarro-Hellín, F. Soto-Valles, and R. Torres-Sánchez, "A software architecture based on FIWARE cloud for precision agriculture," *Agricult. Water Manage.*, vol. 183, pp. 123–135, Mar. 2017.
[26] Arun 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] Y. Mehmood, F. Ahmad, I. Yaqoob, A. Adnane, M. Imran, and S. Guizani, "Internet-of-Things-based smart cities: Recent advances and challenges," *IEEE Commun. Mag.*, vol. 55, no. 9, pp. 16–24, Sep. 2017.
[28] Y. Meidan, M. Bohadana, A. Shabtai, J. D. Guarnizo, M. Ochoa, N. O. Tippenhauer, and Y. Elovici, "ProfilIoT: A machine learning approach for IoT device identification based on network traffic analysis," in *Proc. Symp. Appl. Comput.*, Apr. 2017, pp. 506–509.
[29] *Meta Object Facility (MOF) Core Specification Version 2.5.1*, Meta Object Facility, Milford, MA, USA, Nov. 2016.
[30] MongoDB. (2018). *Mongodb is a Document Database*. [Online]. Available: https://www.mongodb.com/
[31] Mosquitto. (2018). *Eclipse Mosquitto: An Open Source MQTT Broker*. [Online]. Available: https://mosquitto.org/
[32] P. Nguyen, N. Ferry, G. Erdogan, H. Song, S. Lavirotte, J.-Y. Tigli, and A. Solberg, "Advances in deployment and orchestration approaches for IoT—A systematic review," in *Proc. IEEE Int. Congr. Internet Things (ICIOT)*, Jul. 2019, pp. 53–60.
[33] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "FRASAD: A framework for model-driven IoT application development," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 387–392.
[34] O. Novo, "Blockchain meets IoT: An architecture for scalable access management in IoT," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 1184–1195, Apr. 2018.
[35] *OMG Object Constraint Language (OCL), Version 2.3.1*, OMG, Milford, MA, USA, Jan. 2012.
[36] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *Proc. Int. Conf. Inf. Soc. (i-Society)*, Nov. 2014, pp. 92–97.
[37] Oracle. (2019). *CEP EPL Language Reference*. [Online]. Available: https://docs.oracle.com/cd/E12839_01/apirefs.1111/e14304/toc.htm
[38] A. Pal, A. Mukherjee, and P. Balamuralidhar, "Model-driven development for Internet of Things: Towards easing the concerns of application developers," in *Internet Things. User-Centric IoT*, R. Giaffreda, R.-L. Vieriu, E. Pasher, G. Bendersky, A. J. Jara, J. J.P.C. Rodrigues, E. Dekel, B. Mandler, Eds. Cham, Switzerland: Springer, 2015, pp. 339–346.
[39] T. Pflanzner, A. Kertesz, B. Spinnewyn, and S. Latre, "MobIoTSim: Towards a mobile IoT device simulator," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Aug. 2016, pp. 21–27.
[40] Acceleo Project. (2016). *Acceleo Project*. [Online]. Available: https://www.acceleo.org
[41] D. C. Schmidt, "Model-driven engineering," *IEEE Comput. Soc.*, vol. 39, no. 2, p. 25, Feb. 2006.
[42] K. Schwaber and M. Beedle, *Agile Software Development With Scrum*, vol. 1. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.
[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] 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.
[46] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2009.
[47] ThingSpeak. (2010). *Thingspeak for IoT Projects*. [Online]. Available: https://thingspeak.com
[48] ThingWorxs. (2019). *Thingworxs IoT Platform*. [Online]. Available: https://www.ptc.com/en/products/iiot/thingworx-platform
[49] 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.
[50] 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.
[51] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: An extended systematic mapping study," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 67–94, Jan. 2020.

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 for two years in IoT and simulation IoT environments research areas.



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 Engineering of Computer and Telematics Systems Department, University of Extremadura. He has published numerous peer-reviewed papers in international journals, workshops, and conferences. His research interests include component-based software development, aspect orientation, service-oriented architectures, business process modeling, and model-driven development. He is involved in several research projects. He has participated in many workshops and conferences as speaker and member of the program committees. He has been the Head of the Engineering of Computer and Telematics Systems Department, University of Extremadura, since February 2018.

JUAN HERNÁNDEZ received the B.Sc. degree in mathematics from the University of Extremadura, Spain, and the Ph.D. degree in computer science from the Technical University of Madrid. He is currently a Full Professor in languages and systems and the Head of the Quercus Software Engineering Group, University of Extremadura. His research interests include service-oriented computing, cloud computing, and model driven development. He is involved in several research projects as responsible and senior researcher related to these subjects. He has published the results of his research in more than 150 papers in international journals, conference proceedings and book chapters. He has participated in many workshops and conferences as a speaker and a member of the program committee. He is currently the Vice President of SISTEDES, the Spanish Society of Software Engineering and Software Development Technology, and the Vice-Chancellor for Digital Transformation with the University of Extremadura.

MIGUEL A. PÉREZ-TOLEDANO received the M.Sc. degree in computer science from the Polytechnic University of Catalonia, in 1993, and the Ph.D. degree in computer science from the University of Extremadura, in 2008. He is currently an Associate Professor with the Engineering of Computer and Telematics Systems Department, University of Extremadura. He belongs to the Quercus Software Engineering Group. His research interests include software architecture, component-based software development, software coordination and adaptation, and aspect oriented software development. He has participated as an organizer of different editions of the workshop and conferences. He was the Head of the Engineering of Computer and Telematics Systems Department, University of Extremadura, from September 2009 to February 2018.

Chapter 6

Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile

"A dream can make a man feel alive or it can kill him instead. But to be without a dream is to be dead."

Berserk (2003)
Miura, Kentaro

Authors: José A. Barriga, Pedro J. Clemente, Miguel A. Pérez-Toledano, Elena Jurado-Málaga, Juan Hernández
Title: Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile

Year: 2023
Journal: Pervasive and Mobile Computing (PMC)
Quality (JCR): Q2
DOI: https://doi.org/10.1016/j.pmcj.2023.101751

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile[☆]

José A. Barriga [*], Pedro J. Clemente, Miguel A. Pérez-Toledano, Elena Jurado-Málaga, Juan Hernández

INTIA Research Institute, Quercus Software Engineering Group, Spain [1]
Department of Computer and Telematic Systems Engineering, Universidad de Extremadura, Av. Universidad s/n, 10003, Cáceres, Spain

A R T I C L E   I N F O

Article history:
Received 15 January 2022
Received in revised form 30 December 2022
Accepted 11 January 2023
Available online 20 January 2023

Keywords:
IIoT systems
IoT simulation
IoT mobility
Model-driven development
Model to text transformation

A B S T R A C T

Systems based on the Internet of Things (IoT) are continuously growing in many areas such as smart cities, home environments, buildings, agriculture, industry, etc. Device mobility is one of the key aspects of these IoT systems, but managing it could be a challenge. Mobility exposes the IoT environment or Industrial IoT (IIoT) to situations such as packet loss, increased delay or jitter, dynamism in the network topology, new security threats, etc. In addition, there is no standard for mobility management for the most commonly used IoT protocols, such as MQTT or CoAP. Consequently, managing IoT mobility is a hard, error-prone and tedious task. However, increasing the abstraction level from which the IoT systems are designed helps to tackle the underlying technology complexity. In this regard, Model-driven development approaches can help to both reduce the IoT application time to market and tackle the technological complexity to develop IoT applications. In this paper, a Domain-Specific Language based on SimulateIoT is proposed for the design, code generation and simulation of IoT systems with mobility management for the MQTT protocol. The IoT systems generated integrate the sensors, actuators, fog nodes, cloud nodes and the architecture that supports mobility, which are deployed as microservices on Docker containers and composed suitability. Finally, two case studies focused on animal tracking and a Personal mobility device (PMD) based on bicycles IoT systems are presented to show the IoT solutions deployed.

1. Introduction

The Internet of Things (IoT) and Industrial Internet of Things (IIoT) are being exploited in several areas such as smart-cities, home environments, agriculture, industry, intelligent buildings, etc. [1]. As can be seen, IoT applications can be

☆ This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

* Corresponding author at: Department of Computer and Telematic Systems Engineering, Universidad de Extremadura, Av. Universidad s/n, 10003, Cáceres, Spain.

E-mail addresses: jose@unex.es (J.A. Barriga), pjclemente@unex.es (P.J. Clemente), toledano@unex.es (M.A. Pérez-Toledano), elenajur@unex.es (E. Jurado-Málaga), juanher@unex.es (J. Hernández).

1 http://quercusseg.unex.es.

https://doi.org/10.1016/j.pmcj.2023.101751

very different from each other and therefore have different requirements and needs. Thus, one of the more interesting requirements of an IoT environment is the mobility of its devices [2,3]. This is because in certain environments some devices need to be mobile to perform their tasks, e.g. personal mobility devices such as bicycles, scooters, etc. that can be rented in any city, or GPS sensors that may be placed on animals in extensive farms [4,5]. In the same way, manufacture and industrial processes could demand the deployment IoT mobile devices throughout on industrial factory [6].

However, managing the mobility of a device through an IoT environment can be challenging. In this sense, according to the design of the environment, mobility can lead to periods of disconnection, resulting in packet loss [7]. Besides, mobility can also lead to increased delay or, in distributed systems, increased jitter [8], e.g. when a device is far away from its gateway or migrates to another gateway. Mobility also means, at the network level, dealing with network dynamism [9]. As for the most commonly used communication protocols in IoT, such as MQTT [10] or CoaP [11], they do not offer mechanisms for mobility management. In addition to these issues, there are also some concerns such as security [12], efficient battery management of the devices [13], etc.

Approaches to managing all these problems need to be measured and tested in order to handle mobility in an efficient way. For example, in order to avoid packet loss during a disconnection period, the Intermediate Buffering technique [14] can be applied, however several tests are necessary to choose the optimal buffer size for each device. Another example is the need to measure jitter [8], as for some critical devices this parameter should not exceed certain limits. On the other hand, it is necessary to measure and efficiently use the energy of each device to guarantee the correct functioning of the device until the next load. It may also be interesting to test the behaviour of the environment if one of the mechanisms supporting mobility goes down (e.g. the neighbour discovery service to deal with the dynamism of the network topology).

Taken into account the aforementioned problems, several research questions could be defined:

RQ1. How could mobility be managed in IoT systems where the MQTT protocol is used?
RQ2. How might model-driven techniques be applied to model IoT systems with mobile nodes?
RQ3. To what extent is it possible to generate the code needed to simulate an IoT system with mobile nodes from a model of the system?
RQ4. To what extent could simulations of mobile IoT systems be useful for optimising the real system?

Taken into account the aforementioned problems and limitations, in this paper, we propose the use of a Model-Driven Development (MDD) [15,16] approach to design, simulate and generate the IoT mobility systems. In this context, MDD helps domain experts to model the system using high level tools based on models which can be modelled, validated and used to generate the IoT code. Using MDD for developing IoT environment with mobility support helps users reason about the IoT system focusing on the specific domain more than the specific code or framework to use.

SimulateIoT [17] is an MDD approach that makes it possible to design and simulate IoT systems. The IoT systems designed with SimulateIoT can include different IoT nodes such as Cloud, Fog, or Edge nodes and multiple computing services such as Complex Event Processing service, Publish/Subscribe service or Storage service.

However, it cannot model mobile devices or nodes. Mobility could be an interesting extension in order to facilitate the description and simulation of complex IoT environments where IoT mobility represents a key factor. In this way, solutions to device mobility can be measured and tested by means of simulations, thus helping IoT developers to handle mobility efficiently within an IoT system.

In this work, SimulateIoT-Mobile, an extension of SimulateIoT that includes support for simulating IoT systems with mobile nodes, is presented. In this regard, note that the content described in this communication only focuses on describing new contributions or features added as part of the SimulateIoT-Mobile extension.

Thus, the main work contributions are the following:

- This work shows that the use of Model-Driven Development techniques is suitable for developing tools and languages to tackle successfully the complexity of IoT environments where devices mobility is a key factor.
- This work includes a metamodel to model IoT environments with mobile nodes. It includes model restrictions and a Graphical Concrete Syntax.
- A Model-to-text transformation to code generate for specific IoT platform.
- Two case studies have been designed and evaluated in order to validate the proposal.

The rest of the paper is structured as follows. In Section 2, we give an overview of existing IoT simulation approaches centred on both low-level and high-level IoT simulation environments. Next, Section 3 introduces SimulateIoT-Mobile. In Section 4 the MQTT Mobility Management Model is defined. Next, Section 5 presents the *SimulateIoT-Mobile* taken into account design and implementation phases including the *SimulateIoT-Mobile* metamodel and the graphical editor. In Section 6 the model-to-text transformation from SimulateIoT-Mobile models to code is explained. Section 7 shows the IoT environment simulation outputs and how they could be analysed. In Section 8 two case studies are presented. Finally, Section 9 elaborates on the discussion of the presented approach before Section 10 concludes the paper.

2. Related works

Mobility devices in IoT environment has been addressed for multiple points of view: (1) including extending communication protocols, (2) including communication protocols with additional devices characteristics such as the battery, QoS, latency, etc. or (3) using high-level proposals for managing IoT mobility devices.

On the one hand, several protocols allow mobility in IoT environments as reflected in [18]. This article discusses and compares various communication protocols for Wireless Sensor Networks based on 6LoWPAN technology. Some of these protocols are MIPv6, HMIPv6, ZoroMSN or LoWMob among others. All of them try to achieve optimal performance in terms of QoS, Resource Management, Security, Topology control and Routing protocol. However, the work [18] concludes that there is no efficient solution to meet all requirements and constraints of WSN with 6LoWPAN technology.

Similar studies such as [19] communicates that due to resource constraints in IoT or WSNs, the design of new communication protocols is required. Furthermore, studies such as [20,21] come to the same conclusions and present their own proposals to solve this issue. However, IoT is a heterogeneous area, where systems have different requirements and where technologies such as 6LoWPAN are still being studied and applied in IoT systems nowadays [22–24].

An example of such studies is [25], that conducts a comparative study between classical and bio-inspired mobility. This study addresses different schemes of mobility within a WSN, however, a greater effort is made to optimise the so-called sink node. The sink nodes are nodes that move through the WSN collecting data sensed by different devices on the network. In this way, the use of energy of the other devices is harvested by avoiding the use of multi-hop communication. Besides, mobile sink nodes offer other features such as load balancing of the network, in the sense that they can transfer the data collected anywhere in the WSN. Therefore, by optimising the sink nodes, the entire WSN is optimised. To this end, the authors compare the different classical mobility protocols with the bio-inspired ones, concluding that the latter surpasses the classics in several aspects such as network congestion, computational complexity or latency among others. However, although these types of protocols are promising, major research efforts are still needed to effectively implement them in a real WSN.

With the aim of addressing the resource constraints of some IoT systems or WSNs, protocols such as MQTT, Coap or DDS have been developed, becoming in the most widely used in the IoT due to their high performance [26]. However, this protocols lack mechanisms for the use of mobile nodes. In this regard, several works [8,27,28] focus on addressing the challenges of mobility management of these protocols.

In [27] the authors propose a solution to avoid the loss of information when mobile devices are not connected to any MQTT Broker, such as when devices are migrating from one Broker to another. This proposal is based on a technique called intermediate buffering. This technique decouples the production of messages from their publication, establishing between these two phases an intermediate buffer where the messages are stored in an ordered manner with the aim of publishing them in the first instance when the mobile device in question recovers the connection. This technique avoids the loss of information, however, it has not been tested in large scale IoT environments, and it only partially solves one of the problems associated with mobility in IoT when MQTT protocol is being used.

In [28] a protocol is proposed to allow mobility nodes in CoAP IoT environments. For this, an architecture based on three elements is used, these elements are: (1) CoAP Node (Server), (2) CoAP Node (Client), (3) Mobility Management Table (MMT). Thus, the CoAP Node Client can request data from some CoAP Node Server through the Mobility Management Table. The Mobility Management Table stores relevant information about CoAP nodes such as their IP address, temporal IP address, state of the nodes (handover data or not), etc. making possible the communication between nodes whether they are moving or not. Besides, mechanisms to avoid packet loss have been included in this protocol, this mechanism put in "hold" mode the CoAP nodes to avoid wrong publications, for instance, when a node is changing its IP. However, is not taken into consideration the loss of connection with the Mobility Management Table, the connection to another Mobility Management Table, the new data that a node could retrieve during the "hold" mode and its storage (intermediate buffer), etc. which can result in packet loss and also reduced scalability in the sense that devices can only use one Mobility Management Table.

The authors of [8] define a proposal for mobility handling in IoT applications using the MQTT protocol. Since MQTT is not a protocol adapted to handle mobility issues, the authors rely on Intermediate Buffering to guarantee that there is no packet loss in hand-off periods due to mobility nodes. Thus, several experiments were carried out to study the behaviour of intermediate buffers. These experiments include parameters such as access point migration, no available access point periods, the size of the messages published, the number of publisher devices, the inter-message delay, etc. The results of these experiments indicate the optimal buffer size to avoid packet loss depending on the situation to which the mobile environment is subjected. However, Although the experiments carried out deal with a wide range of situations, no mechanism is provided for the user to determine the size of the intermediate buffers in their specific situation.

SimulateIoT-Mobile, the proposal that will be described in this communication, has the aim of helping to develop this kind of IoT systems, i.e. the development of IoT systems with mobile nodes that use publish/subscribe protocols. In this sense, a literature review has been carried out to identify key concepts for managing mobility in IoT systems. Specifically, for those that use the MQTT protocol. So, SimulateIoT-Mobile includes in its implementation an MQTT mobility management model (Section 4). Thus, being able of simulate this kind of IoT systems. Developers can therefore use SimulateIoT-Mobile to model, validate, generate and simulate their IoT systems with mobility characteristics, and use the simulation results to optimise them, identifying weaknesses or errors in their designs (Sections 5–7).

3. Introduction to SimulateIoT-Mobile

SimulateIoT-Mobile is an extension of SimulateIoT [17]. For the sake of clarity, the aim of this Section is to outline the new features added as part of this extension. Thus, differentiating between what was the previous work (SimulateIoT) and what is new (SimulateIoT-Mobile).

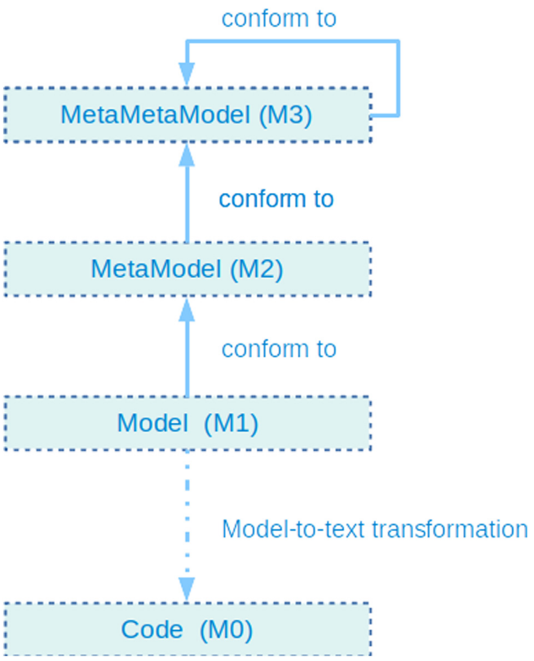


Fig. 1. The four layers of metamodeling. In SimulateIoT [17]: (a) M3 is Ecore, (b) M2 is SimulateIoT Metamodel (c) M1 is a model conform to SimulateIoT Metamodel and (d) Code is generated using the model-to-text transformations defined in SimulateIoT approach.

In this regard, SimulateIoT and therefore SimulateIoT-Mobile are based on the MDD, which 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 Fig. 1). Thus, a Model (M1) is conform to a MetaModel (M2). Moreover, a Metamodel conforms to a MetaMetaModel (M3) which is reflexive [29]. So, 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 [30]. 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.

Therefore, in order to extend SimulateIoT towards SimulateIoT-Mobile, it is required to work in these metamodelling layers. Specifically, it is required to extend: (1) The Metamodel or Abstract Syntax (M2), (2) The Graphical Concrete Syntax or the element that allows to graphically design models (M1) from the Metamodel (M2) and (3) Model-to-Text Transformations (M2T), the element that carry out the code generation (M0) from models (M1).

In this regard, as indicated in Section 1, SimulateIoT does not support mobile devices. Therefore, all the new features added to SimulateIoT (above mentioned metamodelling layers) are focused on allowing it to integrate mobile devices in its simulations (Sections 5 and 6). In addition, SimulateIoT uses the MQTT protocol and this protocol does not natively support device mobility. So, firstly, it is necessary to define, develop and integrate a MQTT mobility model (Section 4) to SimulateIoT to allow it to support mobile devices.

This mobility model aims to manage mobile devices, i.e. this model assumes that mobile devices exist. However, as aforementioned, SimulateIoT, the previous version of this work, is not able to generate or simulate IoT environments with mobile devices. Therefore, additional mobility-related concepts such as the mobile devices itself, their movement logic, the route that each mobile device will follow, or the battery consumption of these devices also have to be part of the SimulateIoT-Mobile metamodel.

Fig. 2 shows, from a high level of abstraction, the deployment of a generic simulation generated by using model-to-text transformation from a SimulateIoT-Mobile model. In Fig. 2 is possible to differentiate the main components included as extensions in this communication of the components belonging to the previous version of the simulator.

In this regard, the Edge/Mist, Fog and Cloud layers have been extended. On the one hand, the Mist/Edge layer has been extended with mobile devices. These devices can follow a (user-defined) route, connect to different fog nodes (to their brokers) during their route, so publishing their data to different brokers, receive coverage signals from different fog

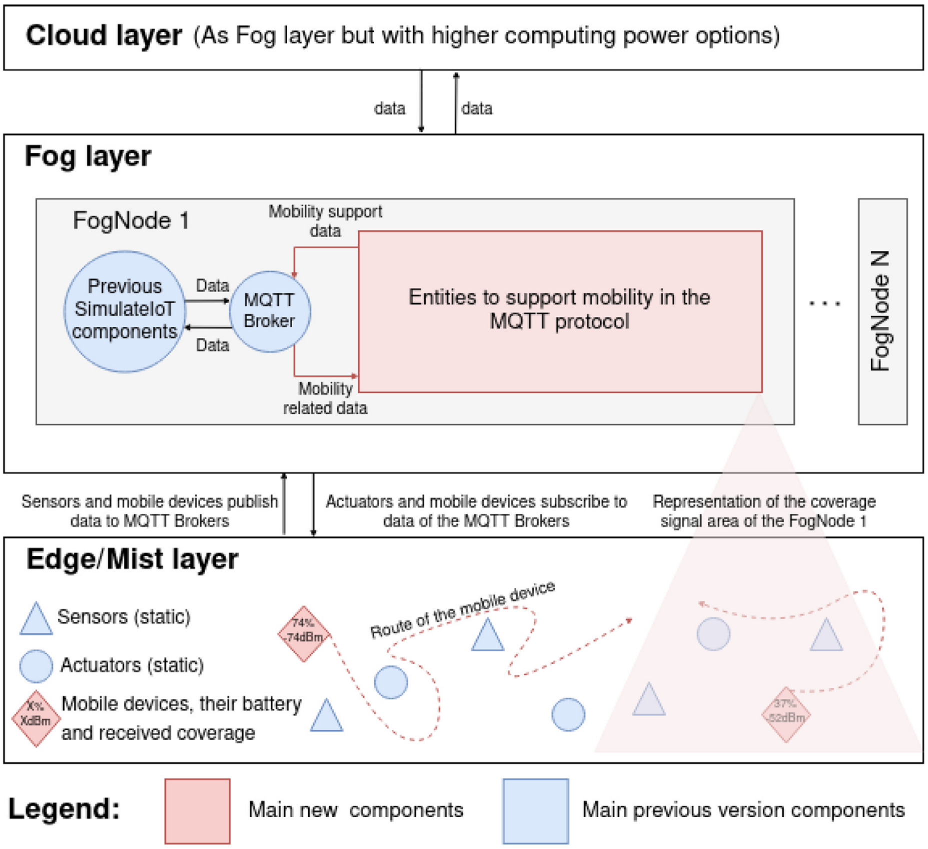

Cloud layer (As Fog layer but with higher computing power options)
data
data
Fog layer
FogNode 1
Mobility support data
Previous SimulateIoT components
Data
MQTT Broker
Data
Entities to support mobility in the MQTT protocol
Mobility related data
FogNode N
Sensors and mobile devices publish data to MQTT Brokers
Actuators and mobile devices subscribe to data of the MQTT Brokers
Representation of the coverage signal area of the FogNode 1
Edge/Mist layer
Sensors (static)
Actuators (static)
Mobile devices, their battery and received coverage
X% XdBm
74% -74dBm
Route of the mobile device
37% -52dBm
Legend:
Main new components
Main previous version components

Fig. 2. A generic model of an IoT system conforms SimulateIoT-Mobile metamodel.

nodes, carry out the simulation of battery consumption due to these new mobility-related concepts, etc. On the other hand, the Fog/Cloud layer has been extended by a set of components that aim to support mobile devices when using the MQTT protocol. These components represent the integration the mobility model (Section 4) in the IoT simulator.

All these extensions are described in detail as follows. First, the MQTT mobility model is presented in Section 4. The extension made to the Metamodel and to the Concrete Syntax is presented in Section 5. The extension made to the M2T is presented in Section 6. The new knowledge that can be obtained to optimise the real system from this extension, is described in Section 7. Besides, Section 8 presents two case of study focused on show the simulations that can be carried out with the extension presented. Finally, note that everything described in these sections is focused on showing the new contributions carried out on SimulateIoT-Mobile and it was not part of SimulateIoT.

4. MQTT mobility management model

In this section, the envisioned model to support mobility in protocols that follow the publish/subscribe paradigm is described. Specifically, the model has been designed for the MQTT protocol, one of the most widely used publish/subscribe protocols in the IoT [26].

First, in Section 4.1 a set of preliminary considerations are made based on the analysis of the protocols that traditionally support mobility and publish/subscribe protocols. Second, in Section 4.2, the necessary entities proposed to provide mobile support for the MQTT protocol are identified and described. In Section 4.3, a solution to mitigate packet loss in the mobility model is proposed. Section 4.4 describes a basic security mechanism to address the vulnerabilities of the model. Next, in Section 4.5 the deployment of the main elements required and their interactions are described. Finally, Section 4.6 presents a review of envisioned scenarios where the MQTT mobility model could be applied.

Note that this model is not intended to be a standard for mobility in publish/subscribe protocols. The MQTT mobility management model proposed is claimed to simulate IoT environments using the MQTT protocol and mobile nodes.

4.1. Preliminary considerations

When traditional protocols address mobility in the literature, they generally address issues such as the entities that carry out the support of mobility, e.g., the Home Agent in the Mobile IP protocol [31], or the Mobility Anchor Point in the HMIP protocol [32]. Also, they address the required data that needs to be shared by the nodes to support mobility, such as the Identifier (Home Address) or the Locator (Care of Address) of mobile nodes in the Mobile IP protocol [31].

On the other hand, interactions between entities are also addressed, such as the interactions needed to start direct communication with a mobile node in the Host Identity Protocol. The Correspondent Node (CN) needs to send the Host Identity Tag to the DNS to get the IP address of the Rendezvous Server. Later, after sending the first packet, the CN and the mobile node can start communication on the direct path [33].

In short, it addresses those issues that allow managing the IPs of the mobile nodes in an efficient and effective way so that changes in the IP by the mobile nodes do not affect the communication between the nodes of the network, and that this management affects as less as possible the parameters involved in the QoS (delay, etc.).

However, the main protocols used in IoT includes publish/subscribe protocols such as MQTT, AMQP or JMS [26]. In the publish/subscribe paradigm, there is no CN to start communication and no mobile node to start communication with. In publish/subscribe protocols there are Brokers that provides Topics, where devices can publish/subscribe to data. For example, in a hypothetical scenario of "intelligent temperature management" of a room, the devices that measure the temperature of the room would publish their measurements in a Topic "Temperature", to which the devices that control the temperature would subscribe.

In our proposal the Brokers are kept static (located on Fog nodes and Cloud nodes), being the devices publishing or subscribing to Topics the mobile nodes. In this regard, the mobile nodes will always be able to communicate with the Broker (static IP) and the Broker will be responsible for redirecting the data. Therefore, mobility management in publish/subscribe protocols (where the Broker remains static) differs from traditional mobility management proposals (no identifiers, locators, mappings, etc.).

4.2. Entities to support mobility in the MQTT protocol: The Broker Discovery Service and the Topic Discovery Service

Thus, focusing on a publish/subscribe protocol such as MQTT for IoT environments there are several key elements: publishers, subscribers, topics and brokers. Data are organised on Topics that are deployed by a Broker and where several elements (IoT nodes) are subscribed and where other elements (IoT nodes) have the role of data publishers. In this context, if a device needs to move through an IoT environment to carry out its own behaviour then it will be needed to connect to different Brokers in order to publish and receive data. Thus, specific entities or services to manage this issue are required. For this regard, a *Broker Discovery Service (BDS)* will be necessary.

On the other hand, publications and subscriptions are addressed to Topics. Since each Topic can be used very differently, a service is needed to analyse the Topics offered by a Broker. Thus, a mobile node can determine whether or not it is feasible to publish/subscribe to a particular Topic. Consequently, an additional service named *Topic Discovery Service (TDS)* is also needed.

Using the Broker and Topic discovery services, a device will be able to: (a) Connect to different Brokers in case it needs it; and (b) Publish/subscribe to compatible Topics that allows it to continue performing their tasks.

In this sense, the BDS allows devices to know the Brokers with which they can establish a connection (Ip). For this, each BDS is deployed together with the rest of the services on a Fog node (one BDS node per Fog/Cloud node). So, the BDS subscribes to the Topic "BDS" where it will receive requests from the mobile nodes. Later, it will publish the responses to each request on a specific Topic for each mobile node ("BDS+DeviceId"). Note that the data shared by the BDS include valuable information such as the distance measure between the device and each Broker.

However, in order to establish a suitable connection with the *Fog* node, not only should be reachable a Fog node (data obtained from BDS), but also the Topics of the Broker's Fog node should be compatible with the Topics required to publish on. To fulfil this requirement, the TDS is used.

Like the BDS, the TDS is a *Fog/Cloud* node service (one TDS node per Fog/Cloud node). This service allows IoT devices to know which Topics of a Broker are compatible with the requesting device. To carry out the above, the TDS connects to the Fog node's Broker, subscribing to a Topic ("TDS") reserved for listening to requests from the devices and publishing the responses in Topics generated dynamically for this purpose ("TDS+DeviceId"). To determine which Topics of a Broker are compatible with a device, the TDS analyses the information provided by the IoT device in its request (the Topics that the device uses and their characteristics) and compares this information with the information it has about each Broker's Topic.

4.3. Disconnection periods and packet loss

Due to the movement of devices and the topology of the designed IoT environment, periods of disconnection may occur, leading to the loss of packets. In order to handle this issue, the Intermediate Buffering technique is applied. The Intermediate Buffering consists of adding a buffer in each mobile device capable of storing the packets that should have been delivered during the disconnection periods. In this way, once the connection is reached, all buffered packets are delivered.

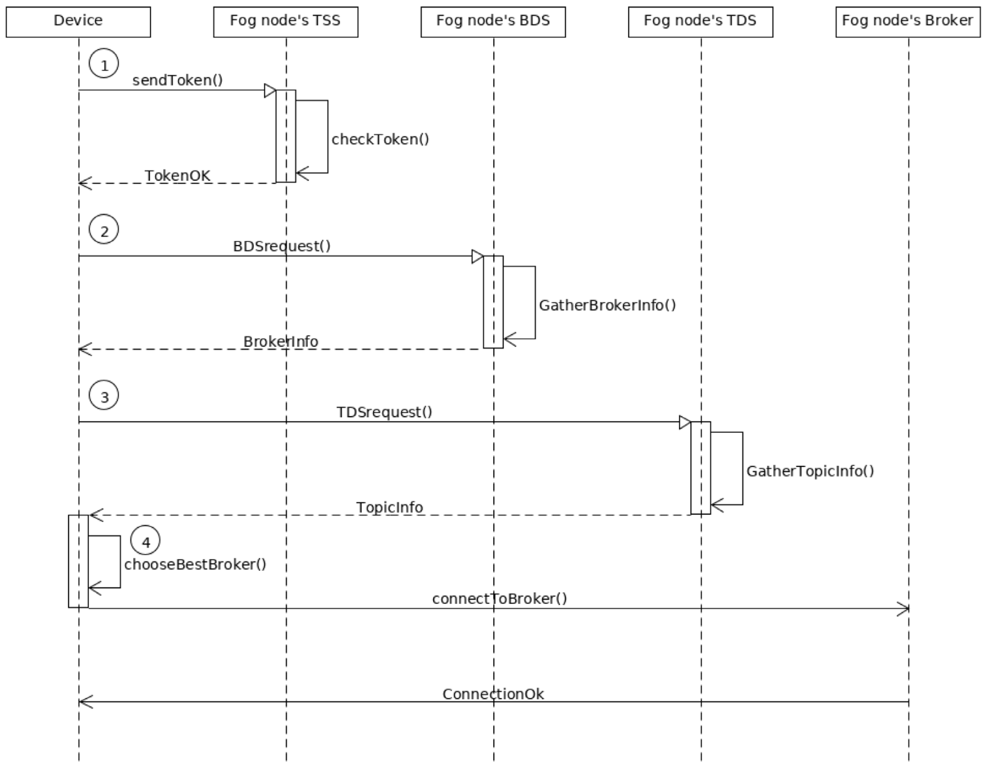


Fig. 3. Topic negotiation protocol to re-connect mobile devices with Fog nodes.

4.4. Security issues

Security is a critical issue in an IoT environment and the mobility of devices leads to the need for additional security services [34]. In this sense, token-based security is usually mandatory, as mobile devices move through the environment making different connections to different nodes and consuming different services in the environment along their way [34]. Thus, a token-based security environment, with the same philosophy as Fiware's token-based security environment [35], has been included in the proposal. Thus, this token-based security environment limits and controls the device connection and access to nodes and services in the IoT environment.

4.5. Model deployment and interactions between entities

A sequence diagram that illustrates the necessary interaction of a device with the mobility architecture (TSS, BDS and TDS) is shown in Fig. 3. It shows four key interactions: Fig. 3-(1) The Device interacts with the Fog node's Token Security System (TSS). In this first interaction, the Device sends its token to the TSS, then the TSS verifies that the token is valid and gives the device the approval to continue with its tasks; Fig. 3-(2). Once the TSS approval is received, the Device requests the BDS to obtain Broker information. The BDS then gathers Broker information and sends it to the Device; Fig. 3-(3). The third interaction is with the TDS, as the device now needs information about the Topics offered in each Broker. Thus, the Device request the TDS, the TDS then gathers information about Topics and sends it to the Device.; Fig. 3-(4). Finally, with all the Brokers and Topics information, the Device can choose the best Broker to connect and establish a connection with it.

4.6. Envisioned scenarios

This section describes several envisioned scenarios or IoT systems for which this MQTT mobility management model has been designed. Thus, being able to be simulated with SimulateIoT-Mobile, taking into account the limitations of the mobility model and the modelling expressiveness of SimulateIoT-Mobile (Section 5). Multiple scenarios could be modelled by using SimulateIoT-Mobile such as Data muling, Animal movement or Smart Cities.

Data muling and ferrying approaches have been vastly investigated. Such proposals focus on managing IoT systems where mobile devices collect data in a specific area, transporting it to a specific nodes of the system [36–39].

Animal movement can be the answer to many biological phenomena, whose understanding could be critical to successfully address challenges such as climate change, species conservation, health and food [40]. Therefore, many IoT-related studies focus their efforts on animal tracking [41–44]. In this regard, Section 8.1 describes a use case where SimulateIoT-Mobile is applied such a system.

Smart cities are IoT systems that can also include mobile nodes. In this sense, mobile nodes can be used for different purposes. The authors of [45] describe an IoT system that tracks vehicles in order to facilitate vehicle parking. The authors of [46] present results from an study where 80 riders of e-bikes discuss their experience with smart mobility. Other studies such as [47] makes a proposal to enable green mobility in cities. This work presents a device that can be integrated into citizens' personal mobility devices, such as segways or electric scooters. This device gathers environmental information to provide personal mobility devices with eco-efficiency services, integrating them in the smart city environment. A use case based on the latter study is carried out in Section 8.2.

In short, SimulateIoT-Mobile is designed to simulate several kinds of IoT systems with mobile nodes, taking into account the limitations outlined in the introduction of this subsection.

To sum up, several of the main characteristics taken into account in IoT environments with mobile devices have been identified. They, together the envisioned scenarios described in Section 4.6, facilitate describing suitably the context where IoT devices should be defined and deployed. In this sense, the next section presents the SimulateIoT-Mobile domain-specific language in order to define IoT environments with mobile devices.

5. Extensions of metamodel and concrete syntax

SimulateIoT-Mobile, as a MDD approach, is composed of three main elements: (1) Metamodel or Abstract Syntax, (2) Graphical Concrete Syntax and (3) Model-to-Text Transformations (M2T). This Section describes the SimulateIoT-Mobile Metamodel and Concrete Syntax.

5.1. Metamodel extensions

A Metamodel captures the concepts and relationships in a specific domain in order to model partially reality [15]. Then, it is possible to design models from this Metamodel. These models can 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.

SimulateIoT metamodel [17] defines in deep the core concepts and relationships related to the IoT domain, including elements such as sensors, actuators, edge node, fog node, cloud node, database, complex-event processing services, data definition, topics, message broker, etc. However, it has not enough expressiveness to simulate IoT systems with mobile nodes. Therefore, SimulateIoT-Mobile metamodel, an extension of SimulateIoT metamodel with enough expressiveness to define IoT systems with mobile nodes, has been developed.

For a sake of clarity, Fig. 4 shows an excerpt of the SimulateIoT-Mobile metamodel, concretely the elements required for modelling IoT mobile devices (elements which are numbered and highlighted in Fig. 4 on blue colour). Note that Fig. 14 (Appendix A) shows the complete metamodel, with the elements relating to the extension carried out highlighted in blue.

This extension includes the classes and relationships needed to model the mobility entities and services described in Section 4. Besides, some concepts necessary to capture the specific mobility domain, such as the routes that mobile nodes will follow, are also included. Finally, some concepts useful for the end-user in terms of simulation analysis (Jitter, Battery etc.) are also included.

Thus, in order to describe the SimulateIoT-Mobile metamodel, this section is divided into the domain-specific IoT mobility concepts identified: Device movement, Disconnection periods and packet loss, Jitter, Battery management, the Broker Discovery Service and the Topic Discovery Service and security issues. In this way, each of these subsections include the contributions that make it possible to model the aspects of these mobility concepts (classes and relations shown in Fig. 4).

5.1.1. Device movement

In order to model device mobility for simulation purposes, the *Route* concept is introduced. A *Route* is a set of coordinates that specifies the movement of one or more mobile devices. Thus, each mobile device is linked to a *Route* that specifies its movement through the IoT environment.

In this way, SimulateIoT-Mobile metamodel proposes defining several kinds of synthetic routes: *FogCloudRoute*, *LinearRoute*, *RandomRoute*, and *CSV_Route*. These *Routes* have been included as classes in the metamodel and are identified in Fig. 4 with numbers two, five, three and four respectively. Note that the class *Route* identified with the number one is an abstract class and superclass of the Route hierarchy.

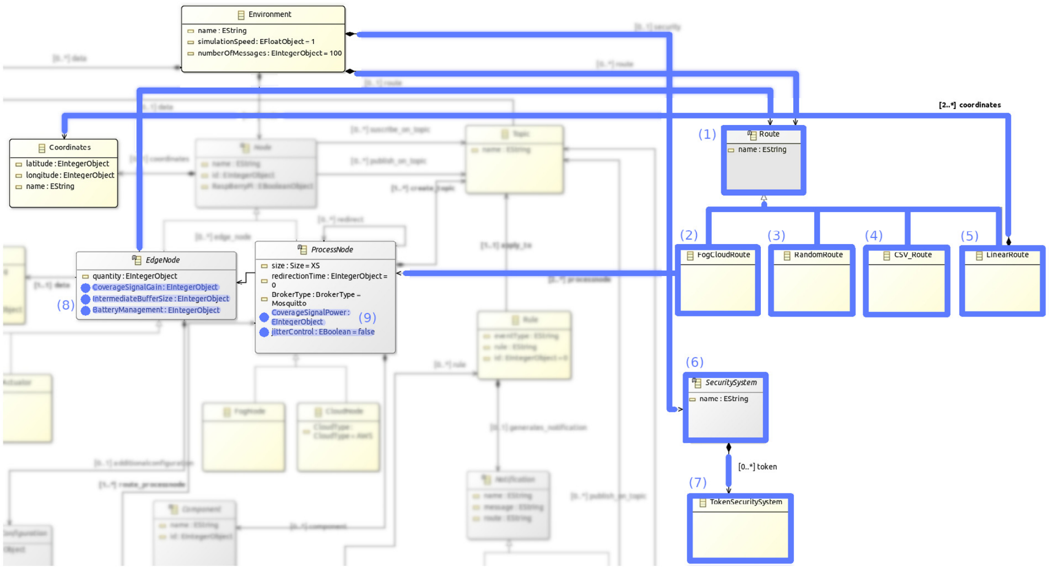

Environment
name : EString
simulationSpeed : EFloatObject = 1
numberOfMessages : EIntegerObject = 100
Coordinates
latitude : EIntegerObject
longitude : EIntegerObject
name : EString
(1)
Route
name : EString
[2..*] coordinates
(2)
FogCloudRoute
(3)
RandomRoute
(4)
CSV_Route
(5)
LinearRoute
EdgeNode
quantity : EIntegerObject
CoverageSignalGain : EIntegerObject
IntermediateBufferSize : EIntegerObject
BatteryManagement : EIntegerObject
(8)
ProcessNode
size : Size = XS
redirectionTime : EIntegerObject = 0
BrokerType : BrokerType = Mosquitto
CoverageSignalPower : EIntegerObject
jitterControl : EBoolean = false
(9)
(6)
SecuritySystem
name : EString
[0..*] token
(7)
TokenSecuritySystem

Fig. 4. Except of SimulateIoT-Mobile metamodel focusing on the mobile concepts. The complete SimulateIoT-Mobile could be found in Appendix A at Fig. 14.

- *FogCloudRoute* class allows users to define a sequence of Cloud and Fog nodes (related to *ProccessNode* class). These nodes are fixed nodes in the IoT environment that has a coordinate (position in the environment). Thus, the sequence of node coordinates defines the *Route* that will be followed by the device linked to it during the whole simulation. This *FogCloudRoute* will be followed by the device linked to it during the whole simulation, from beginning to end and backward.
- *LinearRoute* class allows the user to define routes as a sequence of x/y coordinates (related to *Coordinates* class). So, the mobile device will move throughout this sequence of coordinates indefinitely. Note that once the end of the route is reached, it follows the route in reverse.
- *Random_Route* class makes it possible to generate random routes. These routes start in a specific coordinate and are ad-hoc generated up to the end of the simulation. Note that, from each coordinate is generated the next step direction, avoiding jumps in the route.
- *CSV_Route* class allows the user to load routes defined in a CSV file. The CSV files must include an x/y coordinate in each row. In this way, the device interprets the route and follows it throughout the simulation. As with all other routes, once the end is reached, it retraces the route in reverse.

5.1.2. Disconnection periods and packet loss

As described in Section 4, in order to avoid packet loss, the possibility of using the Intermediate Buffering technique is introduced. Thus, each *EdgeNode* has been extended with the *IntermediateBuffersize* attribute (Fig. 4-(8)). In this way, the end-user is able to specify the amount of memory in terms of Kb that the Intermediate Buffer of a mobile device will have.

5.1.3. Jitter

Currently, a critical aspect in IoT is the delay between the communication of two or more components. In this sense, there are many studies that address this issue which often use simulators to corroborate their hypotheses [48–50]. Therefore, it has been considered appropriate to provide SimulateIoT-Mobile with mechanisms able to model and to measure the delay between components. In particular, when it comes to the delay caused by the mobility of devices (e.g. in a handover period).

Jitter is the variation in the delay of two messages received consecutively by a subscriber from a publisher. This way of measuring delay has been chosen because of the asynchrony of the internal clocks of the devices in a distributed system. When measuring jitter, only the subscriber clock is used. Thus, a possible asynchrony among the internal clocks of the publisher, broker or the subscriber does not affect the measurement [8].

In order to add the concept of jitter in the metamodel, a Boolean attribute called *jitter Controller* (Fig. 4-(9)) has been added to the *ProcessNode* element (nodes where the control services will be deployed). Whether it is specified as *True*, all the services required to measure jitter will be generated in the model-to-text transformations; if it is set to *False*, these services will not be generated.

5.1.4. Battery management

As well as jitter, the battery management is also currently a critical aspect in IoT. An efficient battery consumption in IoT environments is one of the challenges that researchers are currently facing [51,52]. Concerns about energy consumption are even more pronounced in mobile environments, where devices must also expend energy on the movements. Therefore, it has been considered to add the possibility to model the battery of each device so that users can analyse the behaviour of the battery after the simulation.

In this sense, note that there are a large number of IoT devices with different features, so there could be a big difference in consumption from one device to another. Therefore, in order to simulate energy consumption, a count of the tasks that consume energy is carried out. These tasks include: (a) data publishing; (b) data receiving; (c) movement; (d) data processing and storage; and (d) other interactions (e.g. neighbour discovery or security). Thus, the aforementioned parameters are used at simulation run-time to simulate battery consumption.

To model the concept of the battery usage of devices to the metamodel. To do so, an Integer attribute named *batteryManagement* (Fig. 4-(8)) has been added to the *EdgeNode* element (nodes that will be able to simulate their energy consumption). Thus, the user is able to specify the battery milliampere capacity in each device. If it is specified with a value >0, all the services required to simulate the battery consumption will be generated in the model-to-text transformations; if it is set to 0, these services will not be generated.

5.1.5. The Broker Discovery Service and the Topic Discovery Service

The BDS and TDS are two entities introduced in Section 4, designed to manage mobility in the MQTT protocol. These entities are static and their properties are not needed to be modelled by the user. Therefore, the domain-specific features of these entities have not been added to the metamodel. However, there are some concepts related to the execution of these two entities that the user should be able to model, such as the coverage of the access points to these entities and the gain of the devices to sense this coverage.

In order to extend the metamodel in this way, an attribute named *coveragesignalPower* (Fig. 4-(9)) has been added to *FogNode* and *CloudNode* elements. Thus, the end-user is able to define the signal strength of the gateways (included on the *FogNode* and *CloudNode* elements).

During a simulation, this signal strength limits the perimeter within which a mobile node may or may not connect to a gateway. It therefore plays a key role in the design of the architecture of the IoT simulation environment. Thus, users can model the IoT simulation environment and identify areas where there will be no connection, and whether in these areas there are communication problems taken into account properties such as packet loss, size of intermediate buffers, signal strength, etc.

On the other hand, to allow users to model the communications capabilities of a mobile device to detect the gateway coverage signal, an attribute named *coverageSignalGain* has been added to the *EdgeNode* element (Fig. 4-(8)). In this way, the aforementioned coverage perimeters will be variable for each mobile device, thus having different needs (e.g. the size of the intermediate buffer), bringing the simulation closer to reality.

5.1.6. Security issues

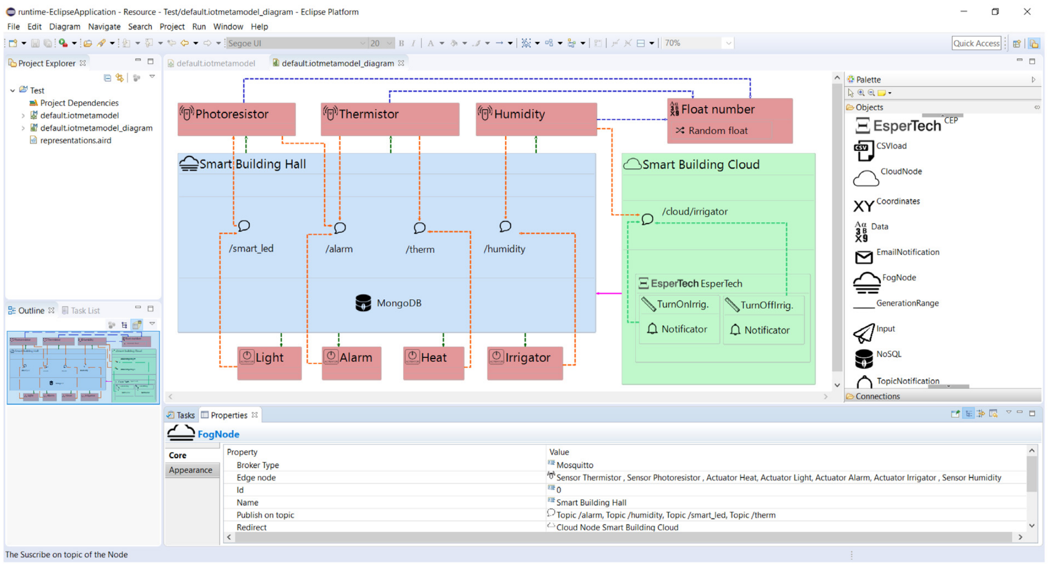
Section 4 describes a token-based security system to address vulnerabilities arising from the proposed mobility management model.

In this regard, the metamodel is extended to provide the user the possibility to choose whether or not to add this security system to the IoT environment. For this purpose, a hierarchy of elements has been added to the metamodel. The superclass of this hierarchy is named *SecuritySystem*(Fig. 4-6 SimulateIoT-Mobile metamodel). This class can contain a security service called *TokenSecuritySystem*(Fig. 4-7 SimulateIoT-Mobile metamodel). If, when modelling an IoT environment, an instance of the *TokenSecuritySystem* class is created, the model-to-text transformations will generate the security architecture necessary to implement the token-based security services discussed in Section 6.6. If it is not instantiated, these services shall not be generated.

5.2. Graphical concrete syntax and validator extensions

*M*odel-Driven Development allows creating models conforming to a metamodel. So, in order to do this, the Eugenia tool [53] makes it possible to generate a Graphical Concrete Syntax (Graphical editor). The Graphical Concrete Syntax generated for SimulateIoT-Mobile metamodel is an extension of the Graphical Concrete Syntax defined in SimulateIoT, which is based on Eclipse GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Tools). Consequently, models (EMF and OCL (Object Constraint Language) [54] based) can be validated against the defined metamodel (EMF and OCL based). Note that OCL is a standard to define model constraints. Fig. 5 shows an excerpt from this graphical editor. It helps users to improve their productivity allowing not only defining models conforming to the *SimulateIoT-Mobile metamodel* but also their validation using this metamodel and OCL constraints [54].

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 SimulateIoTModel metamodel (Fig. 4). Later on, the graphical concrete syntax will be used to model and validate several case studies.



Property	Value
Broker Type	Mosquitto
Edge node	Sensor Thermistor , Sensor Photoresistor , Actuator Heat, Actuator Light, Actuator Alarm, Actuator Irrigator , Sensor Humidity
Id	0
Name	Smart Building Hall
Publish on topic	Topic /alarm, Topic /humidity, Topic /smart_led, Topic /therm
Redirect	Cloud Node Smart Building Cloud

Fig. 5. Graphical editor based on the Eclipse to model conforming to the SimulateIoT-Mobile metamodel.

6. Extensions of model-to-text transformations

As aforementioned, SimulateIoT-Mobile, as a MDD approach, is composed of three main elements: (1) Metamodel or Abstract Syntax, (2) Graphical Concrete Syntax and (3) Model-to-Text Transformations (M2T). In Section 5, the extensions carried out in (1) Metamodel or Abstract Syntax (Section 5.1) and (2) Graphical Concrete Syntax (Section 5.2) were described. Thus, in this section, the extensions for SimulateIoT-Mobile carried out in (3) Model-to-Text Transformations (M2T) are described.

Once the models have been defined and validated conforming to the SimulateIoT-Mobile metamodel (examples of models in the Figs. 8 and 11), a model-to-text transformation defined using Acceleo [55] can generate the IoT environment modelled.

Thus, this section describes the main features of the Model-To-Text transformation carried out in order to generate the IoT environment, focusing in the transformations which allow mobile support (the target of this work). For the sake of clarity, this section is divided into the domain-specific IoT mobility concepts identified (as in Section 5). In this way, each subsection contains the contributions that make it possible to generate the code of each component (M2T transformations) related to these mobility concepts. Finally, a section describing the overall generation and integration of the artefacts is included.

6.1. Device movement

Section 5.1.1 describes the extensions carried out to make it possible to model the movement of mobile devices. In this sense, *Route* is an abstract class that can be specified by different elements: (a) CSV file (*CSV_Route* class), (b) Fog/Cloud nodes (*FogCloudRoute* class), (c) Predefined Coordinates (*LinearRoute* class), and (d) Random Coordinates (*Random_Route* class). In order to manage the *Route* elements and their specifications, the following services are required:

- Mapping services, to map the routes defined in a CSV file, list of Fog/Cloud nodes or Coordinates to a suitable format for the devices.
- A coordinate generation service, to generate realistic random routes in real-time (this service takes care that the direction of the route is consistent, that there are no incoherent movements from one position to another, etc.).
- A route management service, in order to make the mobile devices capable of interpret the routes and move along them during the simulation.

Therefore, Simulate-IoT model-to-text transformations have been extended to generate and integrate these three services on every mobile device in the environment (i.e. on every mobile device modelled by the user in a model).

6.2. Disconnection periods and packet loss

Section 5.1.2 describes the extensions carried out to make it possible to model the application of the *Intermediate Buffering technique* for mobile devices. Thus, in order to implement and apply the Intermediate Buffering technique, it is necessary to include two new services to the mobile devices, a buffer storage service and a buffer publish service.

- Buffer storage service: This service have the capacity (Kb) modelled through the EdgeNode element *Intermediate-Buffersize* attribute (Fig. 4-8). Thus, this service controls the size of the messages that are stored in the buffer and that the memory does not overflow. Whether the buffer is full and the device is still offline, this service acts as a queue, eliminating the oldest messages (packet loss) so that the new ones can be stored, always taking into account the size of each message.
- Buffer publish service: Once the device decides to connect to a gateway, the buffer publish service (integrated with the device's publishing logic) reads and empties the buffer, subsequently publishing all this data.

Therefore, Simulate-IoT M2T transformations have been extended to generate and integrate these two services on every mobile device in the environment (i.e. on every mobile device modelled by the user in a hypothetical model).

6.3. Jitter

Section 5.1.3 describes the extensions needed to make it possible to model whether to deploy the *Jitter* analysis service or not. Thus, in order to generate and deploy the Jitter analysis service, it is necessary to include this service in the Cloud and Fog nodes.

At simulation start, the jitter analysis service is deployed to monitor jitter next to each Broker (deployed at each Fog/Cloud node). Thus, the jitter analysis service subscribes to all Topics, receiving all the messages published in them and registering the reception timestamp of each message. At the end of the simulation, this service calculates the jitter of the messages received by the devices. It should be noted that the data published by each device is structured in JSON format and contains a field reserved for identifying the publisher and the timestamp of each published message [17].

At simulation ends, the jitter control service generates an output with the jitter experienced during the whole simulation so, the average jitter, the maximum jitter and the minimum jitter.

Note that the following expression is used to determine the jitter:

$$Jitter = m'_n - m'_{n-1} - T$$

This expression considers the reception of two messages. The arrival time for message n is defined as m'_n. Note that T is a fixed parameter representing the publishing period of the publisher.

As an instance of the above, consider a situation where a hypothetical sensor has a period T equal to 500 ms, assuming that a message $m'_n - 1$ from the sensor is received by an actuator at instant 0 and the next message m'_n from this sensor is received 621 ms later, the Jitter between these two messages is: $621 - 0 - 500 = 121$ ms.

Thus, Simulate-IoT M2T transformations have been extended in this sense to generate and integrate this service on every Fog or Cloud node modelled in the environment (i.e. on every Fog or Cloud node modelled by the user in a model, whose attribute *jitter_Controller* is setted as True).

6.4. Battery management

Section 5.1.4 describes the extensions carried out to make it possible to model whether to include the *Battery consumption* simulation or not. Thus, in order to generate and deploy the *Battery consumption* simulation, it is necessary to include this simulation module in the Cloud and Fog nodes.

Therefore, the battery simulation is based on the integration of several counters throughout the devices code generated, thus counting each of the tasks carried out by a device. These tasks include: (a) Data publishing, (b) Data receiving, (c) Movement, (d) Data processing and storage, (d) Other interactions (e.g. neighbour discovery or security interactions).

All these parameters are used at simulation run-time to simulate the battery consumption of each device. In addition, once the simulation is finished, the battery simulation service of each device outputs a log with the results of these counters. Thus, the user can then use these parameters to predict more accurately the battery consumption of a specific real device.

Therefore, Simulate-IoT M2T transformations have been extended in this sense to generate and integrate all the aforementioned counters in the device code, thus simulating the battery consumption of each modelled device (i.e. on every device modelled by the user in a hypothetical model).

6.5. Broker Discovery Service and Topic Discovery Service

The BDS and TDS are nodes deployed on each Fog/Cloud node of the system. These nodes are designed to support mobility in IoT environments where the MQTT protocol is used. The behaviour of the BDS nodes and TDS nodes is also described in Section 4.2. However, this section aims to identify and describe individually each of the services generated from the model-to-text transformation for the BDS nodes and the TDS nodes.

The BDS nodes are entities that communicate to mobile devices useful information about the Brokers deployed in the system. In this way, mobile devices can use this information to make an appropriate selection of which Broker to publish to or subscribe to.

In this sense, the device communicates to the BDS (to those within their reach) information about its geographical location. Using this data, the BDS nodes reply to the device with a list of Brokers and details about each one of them, such as their geographical location, IP address or the distance to them in a straight line. Therefore, three services are identified:

- MQTT client: The first service identified is the MQTT client that uses the BDS and the underlying logic to communicate with the target device.
- DataBase client: Secondly, it is identified the client of the database where the BDS queries all the data related to the Brokers.
- To Measure of distance between device and Brokers: Thirdly, it is identified the component that applies the logic necessary to interpret the coordinates of the mobile devices and calculate the distance between it and the Brokers deployed in the system.

On the other hand, the TDS nodes are entities that communicates to mobile devices useful information about the Topics deployed in the system's Brokers. In this regard, the device requests from the TDS nodes data about the Topics deployed in one or several Brokers. Using this list of Brokers, the TDS nodes reply to the device with a list of Topics for each Broker, including information about each of the Topics such as a set of Tags (describing the Topic), its name, etc. For this, two services are identified:

- MQTT client: The first service identified is the MQTT client that uses the TDS and the underlying logic to communicate with the device in question.
- DataBase client: Secondly, the client of the database where the TDS consults all the data related to the Topics of each Broker is identified.

In addition to BDS, TDS code generated, the compilation of its code, the wrapping of it in a Docker, and its deployment and integration with the rest of the system, must also be generated. In this sense, SimulateIoT-Mobile takes these issues into account in the deployment script of the system.

To summarise, Simulate IoT model-to-text transformations have been extended to generate and integrate the BDS and the TDS and each of their services in each Fog or Cloud node of the environment (i.e. on every Fog or Cloud node modelled by the user in a model).

6.6. Security issues

Section 5.1.6 describes the extensions carried out to make possible the modelling of the *Security* services. Thus, in order to secure mobile IoT environments and simulate the impact on the overall performance of the environment, a token-based security system is included in SimulateIoT-Mobile, the *TokenSecuritySystem* (TSS). When the simulation starts, all devices generated from the metamodel share a security token. This token is used by devices when they publish or subscribe to a Topic, so if the Topic is named *temperature*, the device publishes or subscribes to */{token}/temperature*. In this way, if an external device tries to connect to the IoT environment, as it is not in possession of the security token, it will not be able to obtain the data published in any Topic, and will not be able to publish false information in any Topic.

As for the TSS, it is a Fog node's service and it is responsible for managing the tokens. In this way it gives them a random lifespan, generates new tokens when they expire, communicates the new token to the devices, etc.

Therefore, SimulateIoT model-to-text transformations have been extended in this sense to generate and integrate this TSS in all Fog or Cloud nodes of the environment (i.e. on every Fog or Cloud nodes modelled by the user in a model).

6.7. IoT environment generated from M2T transformations

For a better understanding of the extensions carried out in this work and their relationships or interactions, this section describes the overall architecture generated from the M2T transformations from SimulateIoT-Mobile models. To explain the generated architecture, it is divided into the three layers that can constitute an IoT environment defined with SimulateIoT-Mobile: (A) Edge Layer; (B) Fog Layer; (C) Cloud Layer.

(A) Edge Layer

The Edge layer is composed of the set of sensors and actuators of the IoT environment. The architecture of an *Edge* node is illustrated in Fig. 6. In terms of the main elements of the architecture (numbered with the numbers used in Fig. 6):

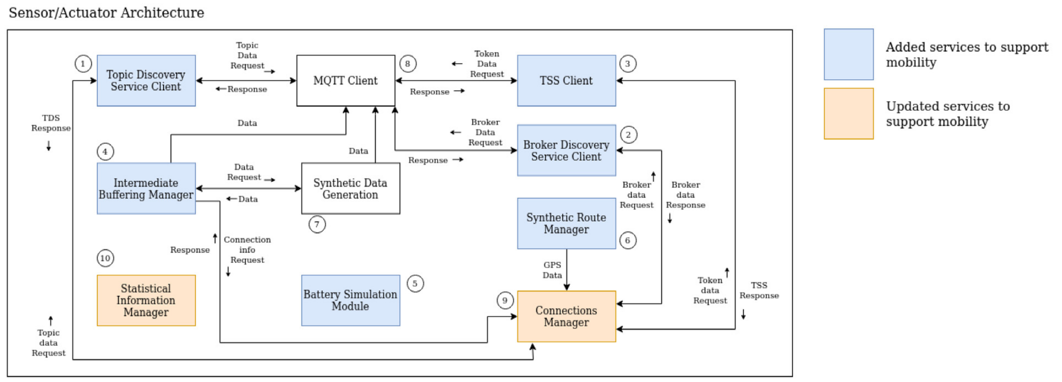


Fig. 6. Software architecture of a Edge node generated.

1. *Topic Discovery Service Client*: Embedded client in the Edge nodes that allows Edge nodes to interact with the Topic Discovery Service offered by the Cloud or Fog nodes. The communication is done through the MQTT protocol (MQTT client relationship) and, once the response is received from the Fog/Cloud node, it is transferred to the Connection Manager component.
2. *Broker Discovery Service Client*: Embedded client in the Edge nodes to interact with the Broker Discovery Service offered by the Cloud or Fog nodes. The communication is done through the MQTT protocol (MQTT client relationship) and, once the response is received from the Fog/Cloud node, it is transferred to the Connection Manager component.
3. *Token Security System Client*: Embedded client in the Edge nodes that allows Edge nodes to interact with the Token Security System offered by the Cloud or Fog nodes. The communication is done through the MQTT protocol (MQTT client relationship) and, once the response is received from the Fog/Cloud node, it is transferred to the Connection Manager component.
4. *Intermediate Buffering Manager*: Intermediate Buffer included in *Edge* nodes to avoid packet loss. This element is related to: (a) The Connections Manager which informs when the connection is *on* or *off*, in order to start or stop storing data. (b) The Synthetic Data Generation element, in order to know which data to store; (c) The MQTT client, to publish the stored data when the connection is *on*.
5. *Battery Simulation Module*: Battery simulation module embedded in the *Edge* nodes to simulate the energy consumption.
6. *Synthetic Route Manager*: It manages, generates or loads routes that Edge devices should follow. This component is linked to the Connections Manager module by sending it the device location. Thus, the Connection Manager module can use the device location to optimise the establishment of new connections.
7. *Synthetic Data Generation*: It manages, generates or uploads the publication of data made by an *Edge* device. It is linked to the MQTT client, thus being able to publish the generated data. It is also related to the Intermediate Buffer so that, in case of disconnection, it stores the generated data.
8. *MQTT Client*: It allows an Edge device to publish or subscribe to Topics on an MQTT. As can be observed in Fig. 6, several components on the Edge node require to publish or subscribe to Topics by using the MQTT Client.
9. *Connections Manager*: It manages the connections among an *Edge* node with the rest of the nodes in the IoT environment. It is related to the Topic Discovery Service, Broker Discovery Service, Token Security System and the Synthetic Route Generation element with the aim of coordinating them when making requests, thus being able to use the responses obtained from each of them to establish optimal connections.
10. *Statistical Information Manager*: It collects data from the device during the simulation in order to integrate them and to produce statistics to be analysed at the end of the simulation for the analysis of the simulation.

(B) Fog Layer

The Fog layer is composed of the set of *Fog* nodes of the IoT environment. The architecture of a *Fog* Node is illustrated in Fig. 7. In terms of the main elements of the architecture (numbered with the numbers used in Fig. 7):

1. *Topic Discovery Service*: Component that implements the Topic Discovery Service explained in Section 6.2. This service is linked to the MQTT client in order to receive requests. In addition, it relates to the MongoDB Client to obtain information about the *Fog* node Topics (response to requests from devices).
2. *Broker Discovery Service*: Component that implements the Broker Discovery Service explained in Section 6.2. This service is linked to the MQTT client in order to receive requests from the Edge nodes.

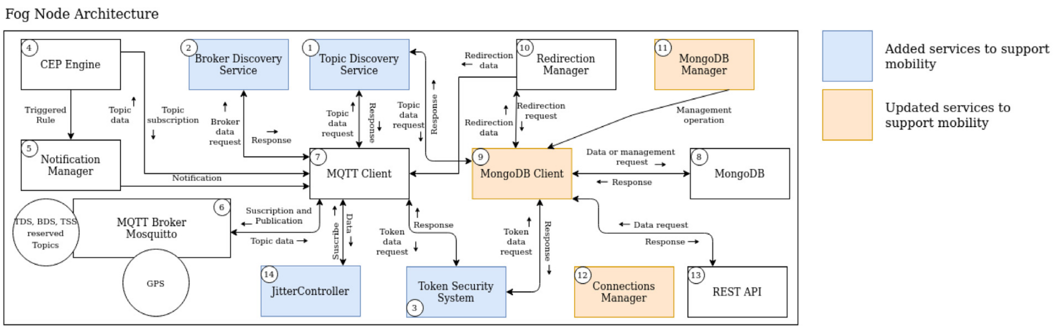


Fig. 7. Software architecture of a Fog node generated.

3. *Token Security System*: Component that implements the Topic Discovery Service explained in Section 6.2. This service is linked to the MQTT client in order to receive requests from the Edge nodes. In addition, it relates to the MongoDB Client to manage Tokens.
4. *Complex Event Processing Engine*: CEP engine that analyses and applies user-defined rules (modelled previously) to data published in the Topics (it is related to MQTT Client). Besides is linked to the Notification Manager element to which it sends its output.
5. *Notification Manager*: Component that collects the analyses carried out by the CEP engine (related to CEP Engine) and publishes them in the Topics that the user has defined for this purpose during modelling phase (relation with MQTT client).
6. *MQTT Broker Mosquitto*: MQTT Broker that supports communication by the MQTT protocol. It is related to the MQTT client of the Fog node to allow it to communicate by using the MQTT protocol.
7. *MQTT Client*: It allows the Fog node to connect to its MQTT Broker and publish or subscribe to its Topics.
8. *MongoDB*: No-Sql database used for data storage on a Fog node. It is related to the MongoDB client as it is the client that performs the queries.
9. *MongoDB Client*: MongoDB client that allows the Fog node to interact with the MongoDB database (related to MongoDB).
10. *Redirection Manager*: Component that allows redirecting data (related to MongoDB Client) among Fog nodes and Cloud nodes.
11. *MongoDB Manager*: Component that includes the necessary interactions with MongoDB (relation with MongoDB Client) in order to ensure the correct performance of the Fog node.
12. *Connections Manager*: Module that manages the connections of a *Fog* node with the rest of the nodes in the IoT environment.
13. *REST API*: REST API that provides information about the *Fog* node to external components. So, internal aspects of the *Fog* node could be requested, for instance, data stored on MongoDB.
14. *JitterController*: Component that measures the jitter produced in the exchange of messages between the different devices in the IoT environment. It has a relationship with the MQTTClient as it needs to subscribe to all Topics in the environment in order to receive the messages published and thus measure the jitter of them.

To sum up, each *Fog* node exposes several interfaces based on different protocols: (a) REST API publish a REST API on port 4000 based on request–response communication schema; and (b) MQTT Broker Mosquito exposes port 18XX that could be used to interchange messages using MQTT protocol based on the well-know publish–subscribe communication schema; (c) The MongoDB database which listens for requests on port 27017 from which queries can be made.

(C) Cloud Layer

The Cloud layer is composed of the set of Cloud nodes of the IoT environment. The architecture of a *Cloud* node is the same as that of a *Fog* node, differing from it only in computational performance, where *Cloud* performance and store capabilities are greater than Fog capabilities. This architecture is illustrated in Fig. 7.

7. Simulation outputs and analysis that can be obtained from the extensions

The main motivation for simulating an IoT system is to gain knowledge to optimise it. Therefore, the benefit that can be derived from an IoT simulator is determined by its outputs. In this regard, SimulateIoT-Mobile provides several outputs that allow to perform several analyses from which to gain knowledge. The main analyses that can be carried out with SimulateIoT-Mobile are the following:

- Whether all messages have been successfully sent from sensors to gateways (*ProcessNode* elements). Data obtained comparing sensors logs and MongoDB storage.
- How many mobile devices have reached the maximum local storage (Intermediate Buffer) due to they do not found a gateway to send data during their routes. Data obtained from each IoT mobile device log.
- Check packet loss rate. Data obtained from each IoT mobile device log.
- Check the jitter produced in the environment during the exchange of messages. Data obtained from the jitter controller component.
- To check the state of the battery of the IoT mobile devices simulated. Data obtained from each IoT mobile device log.
- To check if the gateways deployed (*FogNode* elements) have been enough to attend the IoT mobile devices. Data obtained from each IoT mobile device log and the *FogNode* elements database and logs.
- To check if the complex event processing rules defined have been executed suitable and the *Actuator* elements have executed their actions. Data obtained from each complex event processing event engine log and the message sent to *Actuator* elements.
- Visualise the data interchanged among the IoT mobile devices and the *FogNode* or *CloudNode* elements. Data can be visualised using the view tool *Compass* associated with each *FogNode* or *CloudNode* element.
- To check if there are message bottlenecks on specific *ProcessNode*. It implies that a specific IoT node is a sink of messages which is a potential system risk and a situation that should be avoided. This situation requires to analyse what has been the percentage of messages that cross each *ProcessNode* identifying those which they have a high message rate. Data obtained from different sources, such as the jitter produced at certain times, packet loss rate, node downtime, etc.
- To check if the resources available on the *EdgeNode, FogNode or CloudNode* are enough to deploy suitable the IoT system modelled. Data obtained from different sources, such as the jitter produced at certain times, packet loss rate, node downtime, etc.
- To obtain several statistics related with the number of connections carried out by IoT mobile devices with the gateways deployed (*FogNode* elements). Data obtained from each IoT mobile device log.

8. Case studies

Next, two case studies have been developed using the SimulateIoT-Mobile metamodel and M2T transformations previously presented. The first one defines an IoT simulation of Animals tracking while the second one defines an IoT simulation of Personal mobility devices (PMD) based on public bicycles .

Below is defined a synthesis of the methodology required to use *SimulateIoT-Mobile* 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 and simulate. This model corresponds to the *DSL* and therefore can contain all the elements defined in it.
2. *M2T transformations 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.

8.1. Case 1: Animal tracking

Animal movement can be the answer to many biological phenomena, whose understanding could be critical to successfully address challenges such as climate change, species conservation, health and food [40]. Therefore, many IoT-related studies focus their efforts on optimising the application of these systems in such environments. Moreover, many of these studies corroborate and justify their results through the use of simulations [41–44].

For all these reasons, this first use case is focused on the simulation of an IoT system based on animal movement tracking. So, modelling the behaviour of a system based on GPS devices on animals with MQTT communications *ProcessNode* elements facilitates the animal tracking making it possible to analyse data.

In order to model this IoT system the following aspects are taken into account:

- Each animal has its own GPS devices which communicate with the gateways deployed on the area. So, several *Sensor* elements with mobility capabilities should be included in the model.
- Each *Sensor* element has defined the route that they should follow, this route is a *FogCloudRoute* (Section 5.1.1) that is shared, simulating a flock.
- There are defined several *ProcessNode*, specifically three *FogNode* elements which could be deployed on strategic locations on the area, such as the lagoons where periodically animals should access to drink water.
- Each *Sensor* element (GPS devices) send storage data to the gateway represented by the *FogNode* elements deployed.
- Each *FogNode* element defined, that is the gateways deployed, notifies to a central *CloudNode*.

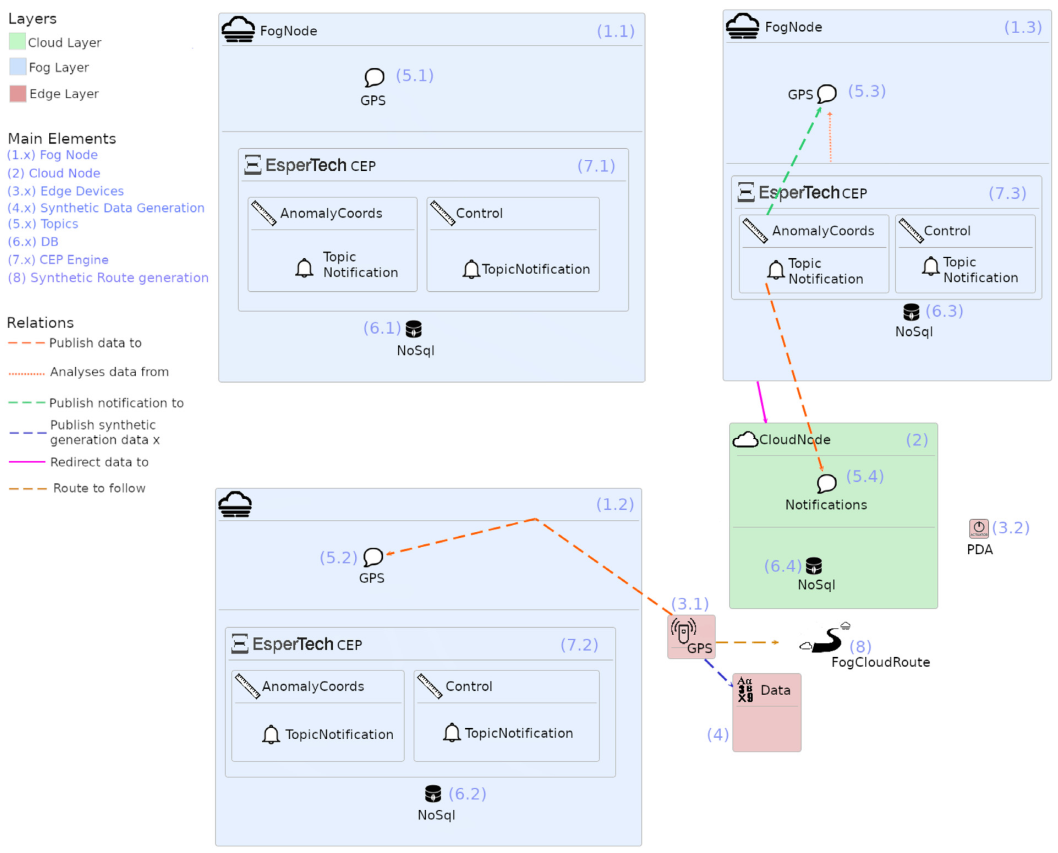


Fig. 8. Case 1. Model simplified conforms to SimulateIoT-Mobile metamodel for animals tracking. Complete version in Fig. 15.

8.1.1. *Model definition*

Fig. 8 shows an excerpt from the animals tracking model. It also includes numerical references for each node which are then used to describe the use case. Note that, for the sake of clarity this extract is simplified, including only one instance of each possible type of relationship between components. The complete version of this model is shown in Fig. 15.

For the purpose of explaining the model, it is divided into three parts: (1) Edge Layer (Red nodes), (2) Fog Layer (Blue nodes), (3) Cloud Layer (Green nodes).

(1) Edge Layer

The Edge layer contains the definition of the sensors (Fig. 8 label 3.1) and actuators (Fig. 8 label 3.2) of the simulation. This sensor represents the GPS that has been incorporated into each animal. This GPS sensor monitors the different locations of a animal throughout the day. On the other hand, the *PDA* actuator (Personal Digital Assistant) has been modelled bearing in mind that there may be use cases where workers are in charge of keeping the integrity of the animals safe, being the PDA the device where they receive notifications of danger. For instance, receiving notifications when an animal is not in the area where it should be, such as outside of a hypothetical protected area where it might be at risk. In addition to this PDA, notification could be also defined to send a message to user applications such as email.

GPS data is assigned by a synthetic data generation (Fig. 8 label 4) and a *Route* (Fig. 8 label 8). Regarding the publication of the data, GPS could publish data in the Topic called *GPS* (Fig. 8 labels 5.1, 5.2, 5.3) located in the Fog nodes. On the other hand, the PDA actuator (Fig. 8 label 3.2) subscribes to the Topic *Notifications* (Fig. 8 label 5.4) located in the Cloud node (Fig. 8 label 2).

Note that for simulation purposes it is not necessary to re-model all elements of the above for each animal. Since each GPS has a *quantity* attribute to specify how many times it should be generated in the M2T transformation phase. Nevertheless, *Route* and the synthetic generation of data should be defined for each animal, otherwise, it will be the same for each of them.

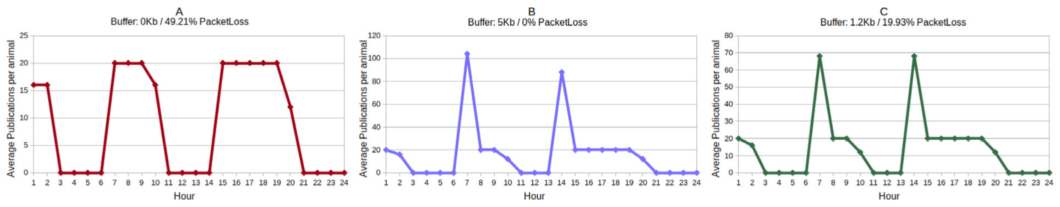


Fig. 9. Case01. Simulation analysis: Intermediate buffer size and Packet loss rate.

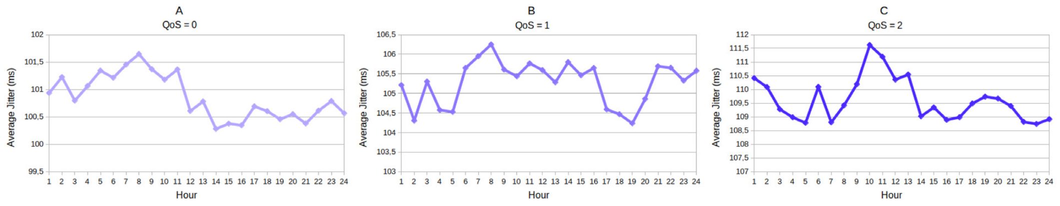


Fig. 10. Case 1. Simulation analysis: Jitter variation during simulation according to the selected QoS (MQTT protocol).

(2) Fog Layer

Fog nodes ((Fig. 8 labels 1.1, 1.2, 1.3) are those that integrate the necessary services for the Edge nodes to carry out their duties. Taking into account the example modelled, each Fog node could be located near watering places where the animals live. For this case study, three Fog nodes have been defined.

The modelling of the Fog nodes is divided into Topics (Fig. 8 labels 5.1, 5.2, 5.3) and the CEP engine (Fig. 8 labels 7.1, 7.2, 7.3). In this use case study, each Fog node offers one Topic, *GPS*, where the GPS incorporated in each animal publishes its location during the day. On the other hand, the CEP engine analyses the data published in these Topics and applies a set of rules to detect anomalies. Specifically, the CEP engine defines two rules: a) *AnomalyCoords* rule, which analyses the data published in the Topic *GPS* and identifies if the location of an animal is inappropriate; (b) Control rule, which analyses the data published in the Topic *GPS* and identifies if an animal does not publish its location for too long a period of time. If one of these rules is met, the CEP engine publishes a notification in the Topic *Notifications* (Fig. 8 label 5.4), located in the Cloud node (Fig. 8 label 2), where the PDA actuator (Fig. 8 label 3.2) is subscribed.

Finally, the Fog nodes are related to the Cloud node. This relationship allows Fog nodes to forward all the data received by their Topics to the Cloud node for storage and future analysis. Note that Fog nodes can also store data if they include a database (Fig. 8 labels 6.1, 6.2, 6.3)

(3) Cloud Layer

As for the Cloud node (Fig. 8 label 2), in this use case it is necessary to model the relationship with the Fog nodes (Fig. 8 labels 1.1, 1.2, 1.3). Thus, it is specified that the Cloud node will receive all the data published in their Topics.

On the other hand, the notifications of the CEP engines incorporated in each Fog node (Fig. 8 labels 7.1, 7.2, 7.3) are sent directly to the Cloud node via MQTT, therefore, it is necessary to define a Topic in the Cloud node. This Topic is *Notifications* (Fig. 8 label 5.4) and is where the PDA actuator is subscribed (Fig. 8 label 3.2), thus receiving any anomaly regarding the animals.

Finally, it is also necessary to model a database to store the received data (Fig. 8 label 6.4).

8.1.2. Model-to-text transformation and deployment

Once the model has been defined, the model-to-text transformation is applied with the following goals: (i) to generate Java, Python, NodeJs, etc. code that wraps each device behaviour; (ii) to generate configuration code to deploy all the generated services, such as the message brokers necessary, including the *topic* configurations defined, the gateway configurations, etc. (iii) to generate the code and deployment configuration files of the architecture that supports mobility (Broker Discovery Service, Topic Discovery Service, Token Security System, etc.). (iv) 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; (v) to generate for each *ProcessNode* and *EdgeNode* a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm*.

Consequently, each Edge node, Fog node and Cloud node is generated following the software architecture defined in Section 6 where model-to-text transformation has been defined.

8.1.3. Simulation analysis

SimulateIoT-Mobile allows users to iteratively model, simulate (execute) and analyse their environment as many times as necessary until the final version is achieved. So, having executed a simulation the users can analyse several data (Section 7).

To exemplify the above mentioned, some experiments and analysis have been applied below on Case 01, Animals Tracking. In this sense, the optimal size of the Intermediate Buffer in different situations, the battery behaviour of the devices and the jitter produced in the message exchange are studied.

First, the packet loss rate is analysed. To carry out this analysis, in a first approximation of the model, it is specified that the GPS incorporated in each animal publishes its data every three minutes. In addition, no Intermediate Buffer has been included. The results after one day simulation (2 min real time - simulation accelerated) are 45.82% packet loss on average per animal (Fig. 9-A). In order to reduce this packet loss rate, an Intermediate Buffer of 5 Kb (250 publications) is added to the GPS of each animal. The results of this second approach are 0% packet loss rate (Fig. 9-B). Finally, a series of tests are carried out to optimise the buffer size and keep the packet loss rate below 20% (hypothetical acceptable threshold). The test results indicate that a buffer size of 1.2 Kb would be necessary to keep the packet loss rate below 20% (Fig. 9-C).

As for the battery, different valuable data can be extracted about its consumption. For example, in this use case when the buffer size is set to 1.2 Kb, (around 20% packet loss) during one day each GPS was connected to the internet for an average of 10.88 h, published a total of 340 messages on average, made 3 connections and disconnections of gateways, etc.

On the other hand, it is also possible to analyse the jitter that occurs during the exchange of messages between devices. Jitter can be measured from different perspectives, in this case, the jitter is measured during a normal exchange, ignoring the increase produced by a handover period (gateway switch) or a disconnection period. In this sense, the results obtained are an average jitter of 100.859 ms, a maximum of 102.831 ms and a minimum of 100.116 ms. Fig. 10-A shows the average jitter of each simulated hour when QoS is set to 0 (this case).

One of the factors involved in the Jitter results is the quality of service offered. In this sense, MQTT has three QoS levels. The above tests have been carried out with a QoS of 0 (minimum QoS allowed by MQTT). When using a QoS of 1 (intermediate QoS level in MQTT) the results are an average jitter of 105.280 ms, a maximum of 109.611 ms and a minimum of 104.259 ms. Fig. 10-B shows the average jitter of each simulated hour when QoS is set to 1. Finally, if the QoS is raised to its maximum level (QoS = 2), the results obtained are an average jitter of 109.614 ms, a maximum of 113.459 ms and a minimum of 105.981 ms. Fig. 10-C shows the average jitter of each simulated hour when QoS is set to 2.

In short, with SimulateIoT-Mobile the users can analyse different aspects of the IoT environment in order to optimise or adapt it to their requirements.

8.2. Case 2: Personal mobility device (PMD) based on public bicycles

In recent years, the presence of PMD's such as bicycles or electric scooters has grown significantly in cities. In order to manage these PMD's and ensure the safety of their users, they can be equipped with several sensors that monitor the status of the PMD in real-time [46,47]. Thus, our second case study presents the simulation of a city with a smart PMD system.

In order to model this case study several assumptions should be taken into account:

- Each PMD includes the following sensors: (a) A GPS that publishes data related to its geolocation; (b) A Wheels pressure sensor, that monitors wheels pressure; (c) A Timer, that monitors the time the PMD is used by a user. On the other hand, the PMD incorporates an Actuator that notifies the user of anomalies, e.g. inadequate wheel pressure.
- The PMD route could be defined as *CSV_Route* based on specific routes defined on the map where the PMD and gateways are deployed.
- Each gateway can be defined as a *FogNode* element that is able to manage the data available on each PMD that reaches a gateway. Note that, from our point of view a *FogNode* element can act as gateway gathering data from sensors or sending data to other *FogNode*, *CloudNode* or *Actuator* elements.
- Each *FogNode* element re-send data to a *CloudNode* element which is able to store and analyse all the data available.
- Each *FogNode* element deployed is able to analyse the data send from the PMD in order to automatically notify the device if it has reached the lease term, the battery is low or the pressure of the wheel is not appropriate. Consequently, PMD incorporates an *Actuator* element that is able to notify the user.

8.2.1. Model definition

Fig. 11 shows an excerpt from the PMD based on the public bicycles model. It also includes numerical references for each node which are then used to describe the use case. Note that, for the sake of clarity, this model is simplified, including only one instance of each possible type of relationship between components. The complete version of this model is shown in Fig. 16 (Appendix B).

For the purpose of explaining the model, it is divided into three parts: (1) Edge Layer (Red nodes), (2) Fog Layer (Blue nodes), (3) Cloud Layer (Green nodes).

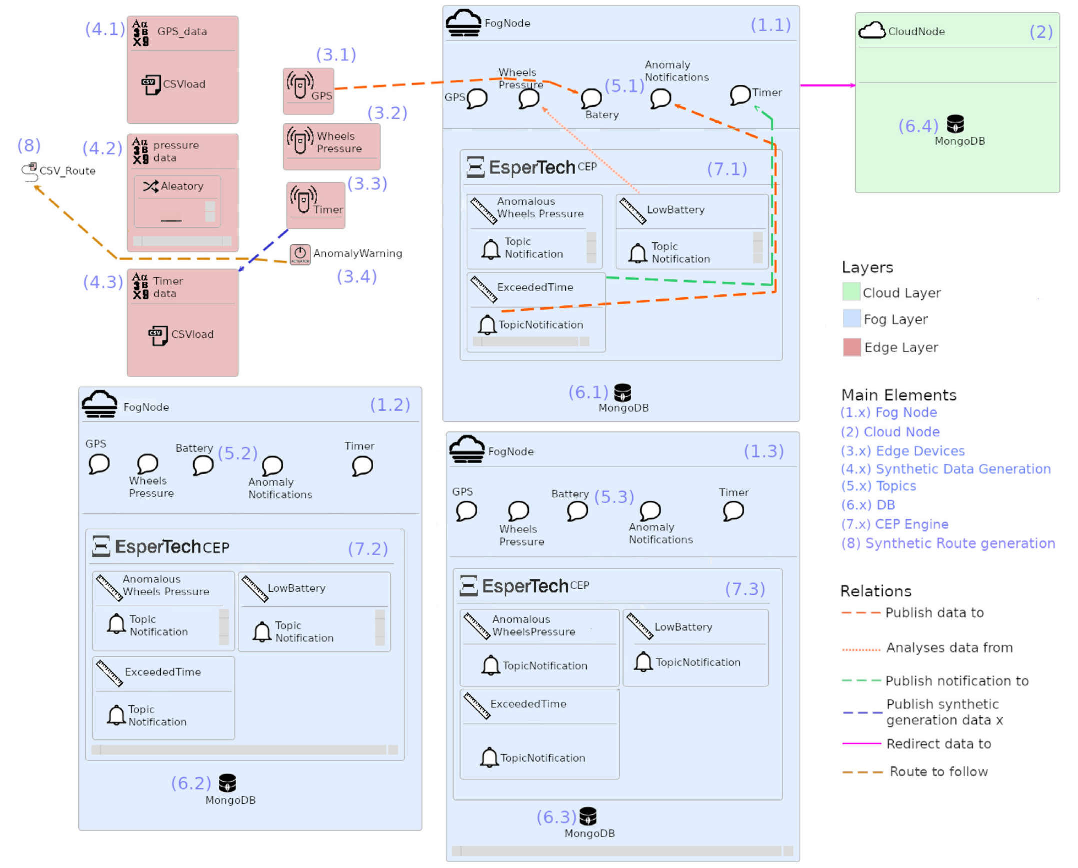


Fig. 11. Case 2. Model conforms to SimulateIoT-Mobile metamodel for Personal mobility device (PMD) based on public bicycles (Simplified version). Complete version in Fig. 16 (Appendix B).

(1) Edge Layer

The Edge layer contains the sensors (Fig. 11 labels 3.1, 3.2, 3.3) and actuators (Fig. 11 label 3.4) of the simulation. This set of devices is the one that has been incorporated into each PMD, thus representing a PMD.

These devices are three sensors and one actuator for each PDM: (a) A GPS (Fig. 11 label 3.1), which monitors the position of the PMD; (b) A pressure sensor (Fig. 11 label 3.2), which monitors the pressure of the wheels; (c) A timer (Fig. 11 label 3.3), which monitors the time the user uses a PMD; (d) An anomaly notifier (Fig. 11 label 3.4), which notifies the user when an anomaly occurs.

For simulation purposes each sensor has assigned a synthetic data generation (Fig. 11 labels 4.1, 4.2, 4.3) and a *Route* (Fig. 11 label 8). Note that, all devices have assigned the same *Route* (Fig. 11 label 8), consequently, this is the PMD *Route*.

Finally, the sensors and the actuator are linked to several Topics (Fig. 11 label 5.1), where they will publish their data or from where they will receive them respectively.

(2) Fog Layer

Fog nodes (Fig. 11 labels 1.1, 1.2, 1.3) are those that integrate the services necessary for the Edge nodes to carry out their functions. For this case study, three Fog nodes have been defined although other numbers of Fog nodes could be defined if needed.

Modelling of the Fog nodes is divided into Topics (Fig. 11 labels 5.1, 5.2, 5.3) and the CEP engine (Fig. 11 labels 7.1, 7.2, 7.3). In this use case, each Fog node offers five Topics: (a) *GPS*, where the *GPS* publishes the location of the *PMD*; (b) *WheelsPressure*, where the *Pressure* sensor publishes the pressure of the wheels; (c) *Battery*, where the different Edge nodes publish their remaining battery life; (d) *AnomalyNotifications*, where the actuator is subscribed for anomalies and the CEP engine publishes the anomalies identified; (e) *Timer*, where the Timer publishes the remaining leasing time.

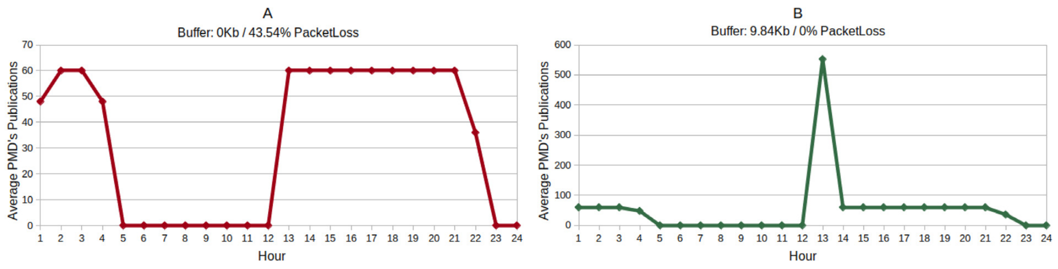


Fig. 12. Case 2. Simulation analysis: Intermediate buffer size required to avoid Packet loss.

On the other hand, the CEP engine analyses the data published in the Topics and applies a set of rules to detect anomalies. Specifically, the CEP engine defines three rules: (a) *AnomalousWheelPressure*, which analyses the data published in the Topic *WheelsPressure* and identifies if the wheel pressure is not adequate; (b) *LowBattery*, which analyses the data published in the Topic Battery and identifies if any device has a low battery; (c) *ExceedTime*, which analyses the data published in the Topic Timer and identifies if the elapsed lease time has expired. If one of the rules is met, the CEP engine publishes a notification in the Topic *AnomalyNotifications*.

Finally, the Fog nodes are related to the Cloud node. This relationship allows Fog nodes to forward all the data received by their Topics to the Cloud node for storage and future analysis. Note that Fog nodes can also store data if they include a database (Fig. 11 labels 6.1, 6.2, 6.3).

(3) Cloud Layer

Cloud node (Fig. 11 label 2) makes it possible to model a node with high capabilities to store and process data. In this case study, the Cloud node (Figure) 11 label 2 stores all data produced in the IoT environment during the simulation process. So, it is needed to model a database to store the received data (Fig. 11 label 6.4). Additionally, it is related to the Fog nodes defined on the model which redirect their data to the cloud node. The Cloud node has defined a Topic named *Notification* which receives all messages thrown several CEP rules defined at the Fog layer.

8.2.2. Model-to-text transformation and deployment

Once the model has been defined, the model-to-text transformation is applied with the same goals as in Case 01 (Section 8.1.2).

Consequently, each Edge node, Fog node and Cloud node is generated following the software architecture defined in Section 6 where model-to-text transformation has been defined.

8.2.3. Simulation analysis

Section 8.1.3 describes and exemplifies some of the experiments and analyses that can be carried out with SimulateIoT-Mobile. This subsection illustrates some additional experiments and analyses that the user could carry out in Case02, a Personal mobility device (PMD). In particular, the impact of a Fog node downtime in terms of packet loss is studied. Besides, the impact of switching brokers on jitter is analysed.

In this use case, the gateways are strategically distributed so that the devices in the environment do not suffer periods of disconnection. Therefore, the use of the Intermediate Buffer is not necessary. However, it is interesting to study the case where one of the Fog nodes goes down (including its gateway) and analyse the number of packets that could be lost in this case.

For this experiment, a device that follows a route that frequents the area with no coverage due to the Fog node downtime has been selected. This device publishes one publication per minute. The output logs of this device show a result of 43.54% of packets lost (Fig. 12-A).

In a hypothetical IoT environment where this Fog node could be down on a regular basis, the user could choose to add an Intermediate Buffer to the devices to avoid packet loss. In this use case, after several tests with SimulteIoTMobile, it is concluded that a 9.84 Kb (492 publications) buffer is needed to avoid packet loss (Fig. 12-B).

On the other hand, this use case studies the impact of switching brokers on jitter. Thus, the jitter of the messages published by a random device has been analysed during simulation execution. This device has switched Brokers approximately 100 times. Each switch involves interacting with the TSS, TDS and BDS, as well as coordinating the requests and responses of these components. The results of this study are an average jitter of 115.668 ms, a maximum of 824.735 ms and a minimum of 100.014 ms. Looking at the maximum jitter it is possible to state that during a Broker switch there is an additional jitter of about 724 ms (worst case). These results may indicate to the user the need to re-model their environment with a view to reducing the impact of jitter in their environment, e.g. critical section that requires a jitter of fewer than 820 ms. Fig. 13 shows an extract of 140 delay measurements where three periods of handover or gateway switching occur.

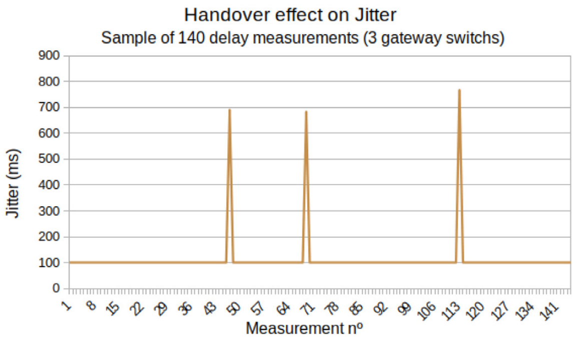


Fig. 13. Case 2. Simulation analysis: Extract of 140 delay measurements where three periods of handover occur.

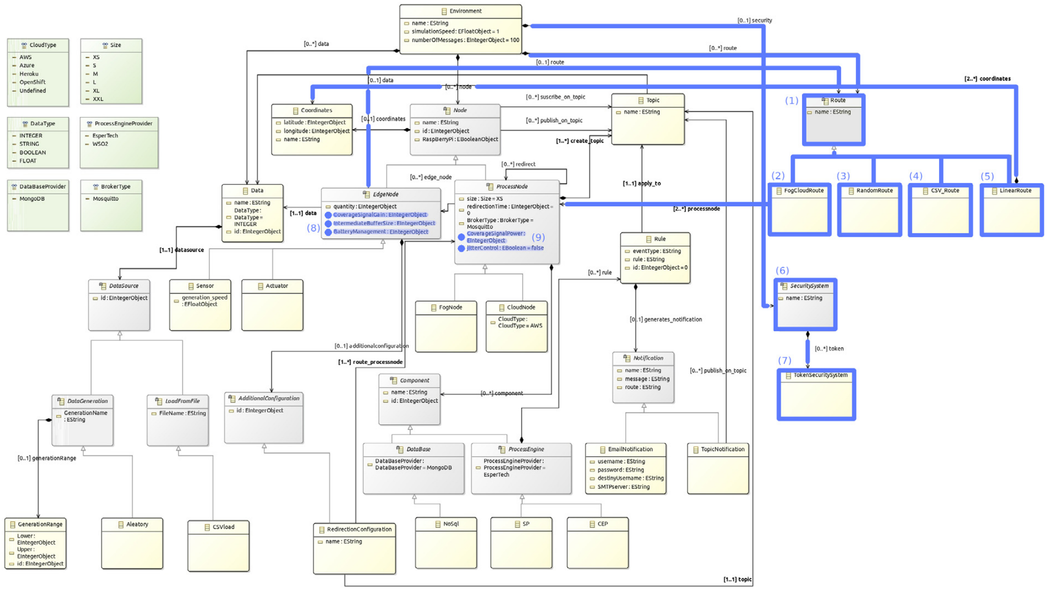


Fig. 14. Complete SimulateIoT-Mobile metamodel.

9. Discussion

In order to discuss the main facts reached by the proposal, the research questions previously defined will be answered.

In relation to RQ1, *"How could mobility be managed in IoT systems where the MQTT protocol is used?"*, in order to manage mobility in IoT systems based on MQTT protocol, several artefacts should be suitably generated (TSS, BDS, TDS) to manage the data among IoT devices and the additional application layer interactions needed to manage IoT mobility should be implemented. Consequently, as has been shown previously it is possible to manage mobility in IoT systems by using MQTT protocol.

Regarding RQ2, *"How might model-driven techniques be applied to model IoT systems with mobile nodes?"*, using model-driven development helps to manage the complexity of heterogeneous technology as a success during an IoT environment development. In this work the IoT systems with mobile nodes are modelled at high abstraction level by using metamodeling techniques. In addition, models obtained could be validated by using OCL (Objects Constrain Language) which guarantees that models are conformed to the metamodel proposed. The metamodel proposed makes it possible to model the target IoT systems using common domain elements.

Concerning RQ3, *"To what extent is it possible to generate the code needed to simulate an IoT system with mobile nodes from a model of the system?"*, modelling IoT environments is a key activity for any IoT project making it possible to focus on the

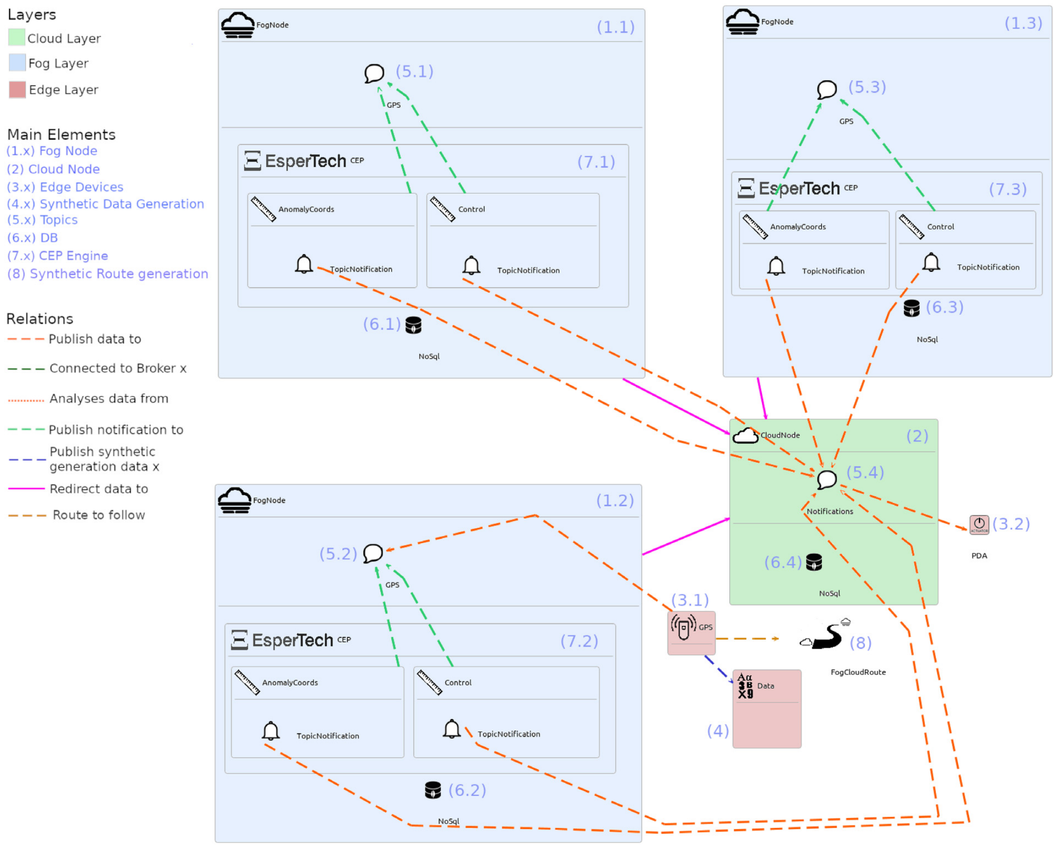


Fig. 15. Case 01. Model conforms to SimulateIoT-Mobile metamodel for animals tracking (complete version).

IoT domain in order to later on generate final code from the models defined. Additionally, modelling and simulating the behaviour of the IoT environments including mobile devices facilitates analysing of several system complex aspects such as battery behaviour, jitter, Intermediate Buffer, storage data, mobile communication protocols and so on. Code generate from the models defined includes multiple artefacts (described in Section 6) which are suitably orchestrated to simulate the IoT environment defined.

Finally, in relation to RQ4, *"To what extent could simulations of mobile IoT systems be useful for optimising the real system?"*, users are able to evaluate the system modifying their characteristics in order to find the better trade-off among the devices and nodes deployed. Specifically, users can use DSL tools such as the Graphical Editor to model the system and the model-to-text transformation to generate the code for deploying the simulation and checking the statistics generated during the simulation.

On the other hand, although there are interesting advantages to using SimulateIoT-Mobile DSL, there are some issues related to the mobility proposal presented.

Firstly, the publish/subscribe communication protocol used is based on MQTT protocol [10], although other publish/subscribe protocols can be used adding it to the model-to-text transformation. Secondly, model-to-text transformation, it has been defined for a concrete target based on microservices deployed on Docker containers which represent the concrete IoT nodes defined on the model. Other technological targets could be defined which implies re-code the model-to-text transformation. Thirdly, the routes of the IoT devices have been defined using common IoT mobility patterns, but additional IoT mobility patterns could be defined. Consequently, it would imply including additional modelling elements and including the new behaviours on the model-to-text transformation. Finally, current version of SimulateIoT-Mobile, for the sake of simplicity, allows defining connected nodes by TCP/IP, and we assume that connectivity is guaranteed.

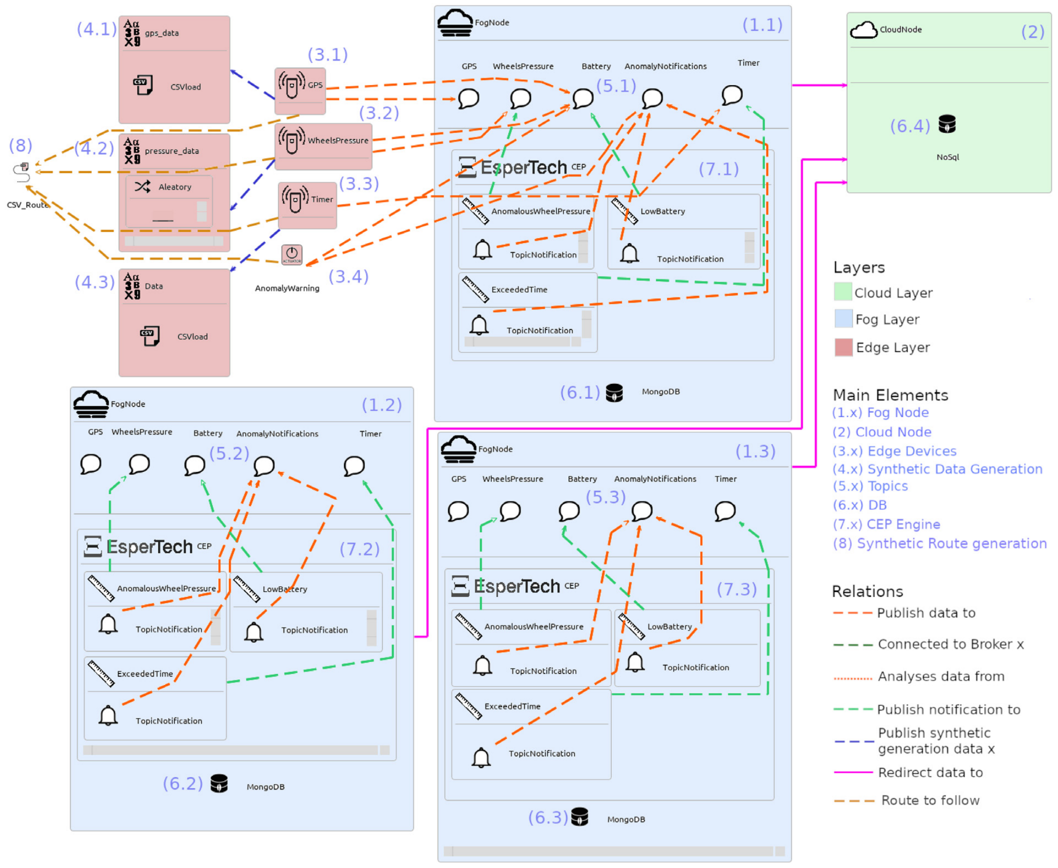


Fig. 16. Case02. Model conforms to SimulateIoT-Mobile metamodel for Personal mobility device (PMD) based on public bicycles (complete version).

10. Conclusions

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.

The IoT simulation methodology and tools proposed in this work help users to think about the IoT system in general and IIoT in particular, to propose several IoT alternatives and policies in order to achieve a suitable IoT architecture, including modelling IoT mobile nodes. In this sense, several kinds of mobile devices and routes can be defined, allowing defining realistic IoT environments. Finally, the IoT environments modelled can be deployed, simulated and analysed.

Future works include extending the metamodel and model-to-text transformation to model additional publish–subscribe communication protocols such as JMS or AMQP; or request–response protocols such as REST. Both extensions facilitate modelling IoT environments taking into account additional heterogeneity technology. Additionally, additional IoT mobile behaviours and routes could be identified and modelled. Finally, the model-to-text transformation could make it possible to generate Cloud support based on well-known Cloud providers such as AWS or Azure. It could open interesting research areas for IoT simulation purposes.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgement

Cofinanciado por la Unión Europea
JUNTA DE EXTREMADURA
Consejería de Economía, Ciencia y Agenda Digital

Appendix A

This Section shows in Fig. 14 the complete metamodel of SimulateIoT-Mobile. This metamodel is composed of the SimulateIoT metamodel and the extension carried out (highlighted in blue). The description of the classes and relationships that are not part of the extension (and that have not been addressed in this article), can be found in the article [17] Section IV, subsection A.

Appendix B

This Section shows the complete version of the models shown in Figs. 8 (Case 01. Animal tracking) and 11 (Case02. Personal mobility device (PMD) based on public bicycles) respectively in Figs. 15 and 16.

References

[1] E. Siow, T. Tiropanis, W. Hall, Analytics for the internet of things: A survey, ACM Comput. Surv. 51 (4) (2018) 74.
[2] S.M. Ghaleb, S. Subramaniam, Z.A. Zukarnain, A. Muhammed, Mobility management for IoT: a survey, EURASIP J. Wireless Commun. Networking 2016 (1) (2016) 1–25.
[3] K. Nahrstedt, H. Li, P. Nguyen, S. Chang, L. Vu, Internet of mobile things: Mobility-driven challenges, designs and implementations, in: 2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI), IEEE, 2016, pp. 25–36.
[4] H. Teng, Y. Liu, A. Liu, N.N. Xiong, Z. Cai, T. Wang, X. Liu, A novel code data dissemination scheme for internet of things through mobile vehicle of smart cities, Future Gener. Comput. Syst. 94 (2019) 351–367.
[5] L. Nóbrega, A. Tavares, A. Cardoso, P. Gonçalves, Animal monitoring based on IoT technologies, in: 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany), 2018, pp. 1–5.
[6] F. Almada-Lobo, The industry 4.0 revolution and the future of manufacturing execution systems (MES), J. Prod. Innov. Manage. 3 (4) (2015) 16–21.
[7] M.B. Yassein, S. Aljawarneh, W. Al-Sarayrah, Mobility management of internet of things: Protocols, challenges and open issues, in: 2017 International Conference on Engineering & MIS, ICEMIS, IEEE, 2017, pp. 1–8.
[8] J.E. Luzuriaga, J.C. Cano, C. Calafate, P. Manzoni, M. Perez, P. Boronat, Handling mobility in IoT applications using the MQTT protocol, in: 2015 Internet Technologies and Applications, ITA, IEEE, 2015, pp. 245–250.
[9] L. Farhan, S.T. Shukur, A.E. Alissa, M. Alrweg, U. Raza, R. Kharel, A survey on the challenges and opportunities of the internet of things (IoT), in: 2017 Eleventh International Conference on Sensing Technology, ICST, IEEE, 2017, pp. 1–5.
[10] Oasis, Message queuing telemetry transport (MQTT) v5.0 oasis standard, 2019, URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.
[11] CoAP, The constrained application protocol (CoAP) - RFC 7252, 2014, https://datatracker.ietf.org/doc/html/rfc7252.
[12] S.-M. Cheng, P.-Y. Chen, C.-C. Lin, H.-C. Hsiao, Traffic-aware patching for cyber security in mobile IoT, IEEE Commun. Mag. 55 (7) (2017) 29–35.
[13] X. Liu, N. Ansari, Toward green IoT: Energy solutions and key challenges, IEEE Commun. Mag. 57 (3) (2019) 104–110.
[14] J.E. Luzuriaga, M. Perez, P. Boronat, J.C. Cano, C. Calafate, P. Manzoni, Improving mqtt data delivery in mobile scenarios: Results from a realistic testbed, Mob. Inf. Syst. 2016 (2016).
[15] B. Selic, The pragmatics of model-driven development, IEEE Softw. 20 (5) (2003) 19–25.
[16] A. Wortmann, O. Barais, B. Combemale, M. Wimmer, Modeling languages in industry 4.0: An extended systematic mapping study, Softw. Syst. Model. 19 (1) (2020) 67–94.
[17] J.A. Barriga, P.J. Clemente, E. Sosa-Sánchez, A.E. Prieto, SimulateIoT: Domain specific language to design, code generation and execute IoT simulation environments, IEEE Access 9 (2021) 92531–92552.
[18] M. Bouaziz, A. Rachedi, A survey on mobility management protocols in wireless sensor networks based on 6LoWPAN technology, Comput. Commun. 74 (2016) 3–15.
[19] C.C. Sobin, A survey on architecture, protocols and challenges in IoT, Wirel. Pers. Commun. (ISSN: 1572-834X) 112 (3) (2020) 1383–1429, URL https://doi.org/10.1007/s11277-020-07108-5.
[20] R. Silva, J.S. Silva, F. Boavida, A proposal for proxy-based mobility in WSNs, Comput. Commun. 35 (10) (2012) 1200–1216.
[21] R. Silva, J. Sa Silva, F. Boavida, Mobility in wireless sensor networks – Survey and proposal, Comput. Commun. (ISSN: 0140-3664) 52 (2014) 1–20, URL https://www.sciencedirect.com/science/article/pii/S0140366414001911.
[22] B. Bettoumi, R. Bouallegue, LC-DEX: Lightweight and efficient compressed authentication based elliptic curve cryptography in multi-hop 6LoWPAN wireless sensor networks in HIP-based internet of things, Sensors (ISSN: 1424-8220) 21 (21) (2021) URL https://www.mdpi.com/1424-8220/21/21/7348.
[23] H.A. Al-Kashoash, H. Kharrufa, Y. Al-Nidawi, A.H. Kemp, Congestion control in wireless sensor and 6LoWPAN networks: toward the internet of things, Wirel. Netw. 25 (8) (2019) 4493–4522.
[24] M.L. Miguel, E. Jamhour, M.E. Pellenz, M.C. Penna, SDN architecture for 6LoWPAN wireless sensor networks, Sensors 18 (11) (2018) 3738.
[25] R. Hamidouche, Z. Aliouat, A.M. Gueroui, A.A.A. Ari, L. Louail, Classical and bio-inspired mobility in sensor networks for IoT applications, J. Netw. Comput. Appl. 121 (2018) 70–88.

[26] Y. Chen, T. Kunz, Performance evaluation of IoT protocols under a constrained wireless access network, in: 2016 International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT), 2016, pp. 1–7.

[27] J.E. Luzuriaga, J.C. Cano, C. Calafate, P. Manzoni, M. Perez, P. Boronat, Handling mobility in IoT applications using the MQTT protocol, in: 2015 Internet Technologies and Applications, ITA, 2015, pp. 245–250.

[28] S. Chun, J. Park, Mobile CoAP for IoT mobility management, in: 2015 12th Annual IEEE Consumer Communications and Networking Conference, CCNC, 2015, pp. 283–289.

[29] C. Atkinson, T. Kuhne, Model-driven development: a metamodeling foundation, IEEE Softw. 20 (5) (2003) 36–41.

[30] S. Sendall, W. Kozaczynski, Model transformation: The heart and soul of model-driven software development, IEEE Softw. 20 (5) (2003) 42–45.

[31] C. Perkins, Mobile IP, IEEE Commun. Mag. 35 (5) (1997) 84–99.

[32] R. Wakikawa, Z. Zhu, L. Zhang, A survey of mobility support in the internet. RFC 6301, 2011, URL https://www.rfc-editor.org/info/rfc6301.

[33] R. Moskowitz, P. Nikander, P. Jokela, T. Henderson, Host Identity Protocol, Tech. rep., 2008.

[34] A.R. Sfar, E. Natalizio, Y. Challal, Z. Chtourou, A roadmap for security challenges in the internet of things, Digit. Commun. Netw. 4 (2) (2018) 118–137.

[35] C. Thomás Oliveira, R. Moreira, F. de Oliveira Silva, R. Sanches Miani, P. Frosi Rosa, Improving security on IoT applications based on the FIWARE platform, in: 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications, AINA, 2018, pp. 686–693.

[36] E. Tuyishimire, A. Bagula, A. Ismail, Clustered data muling in the internet of things in motion, Sensors (ISSN: 1424-8220) 19 (3) (2019) URL https://www.mdpi.com/1424-8220/19/3/484.

[37] A. Bagula, E. Tuyishimire, J. Wadepoel, N. Boudriga, S. Rekhis, Internet-of-things in motion: A cooperative data muling model for public safety, in: 2016 Intl IEEE Conferences on Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016, pp. 17–24.

[38] O. Tsilomitrou, N. Evangeliou, A. Tzes, Mobile robot tour scheduling acting as data mule in a wireless sensor network, in: 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), 2018, pp. 327–332.

[39] A. Ismail, E. Tuyishimire, A. Bagula, Generating dubins path for fixed wing uavs in search missions, in: International Symposium on Ubiquitous Networking, Springer, 2018, pp. 347–358.

[40] R. Kays, M.C. Crofoot, W. Jetz, M. Wikelski, Terrestrial animal tracking as an eye on life and planet, Science 348 (6240) (2015) aaa2478, URL https://www.science.org/doi/abs/10.1126/science.aaa2478.

[41] T.M. Behera, S.K. Mohapatra, U.C. Samal, M.S. Khan, Hybrid heterogeneous routing scheme for improved network performance in WSNs for animal tracking, Internet Things (ISSN: 2542-6605) 6 (2019) 100047, URL https://www.sciencedirect.com/science/article/pii/S2542660518301914.

[42] F. Maroto-Molina, J. Navarro-García, K. Prí ncipe Aguirre, I. Gómez-Maqueda, J.E. Guerrero-Ginel, A. Garrido-Varo, D.C. Pérez-Marín, A low-cost IoT-based system to monitor the location of a whole herd, Sensors (ISSN: 1424-8220) 19 (10) (2019) URL https://www.mdpi.com/1424-8220/19/10/2298.

[43] Q.M. Ilyas, M. Ahmad, Smart farming: An enhanced pursuit of sustainable remote livestock tracking and geofencing using IoT and GPRS, Wirel. Commun. Mob. Comput. (ISSN: 1530-8669) 2020 (2020) 6660733, URL https://doi.org/10.1155/2020/6660733.

[44] J.G. Panicker, M. Azman, R. Kashyap, A LoRa wireless mesh network for wide-area animal tracking, in: 2019 IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT, 2019, pp. 1–5.

[45] P. Sadhukhan, An IoT-based E-parking system for smart cities, in: 2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI, 2017, pp. 1062–1066.

[46] F. Behrendt, Why cycling matters for smart cities. Internet of bicycles for intelligent transport, J. Transp. Geogr. (ISSN: 0966-6923) 56 (2016) 157–164, URL https://www.sciencedirect.com/science/article/pii/S0966692316300746.

[47] R. Sanchez-Iborra, L. Bernal-Escobedo, J. Santa, Eco-efficient mobility in smart city scenarios, Sustainability (ISSN: 2071-1050) 12 (20) (2020) URL https://www.mdpi.com/2071-1050/12/20/8443.

[48] A. Dorri, S.S. Kanhere, R. Jurdak, P. Gauravaram, Blockchain for IoT security and privacy: The case study of a smart home, in: 2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), 2017, pp. 618–623.

[49] Z. Ning, P. Dong, X. Kong, F. Xia, A cooperative partial computation offloading scheme for mobile edge computing enabled internet of things, IEEE Internet Things J. 6 (3) (2019) 4804–4814.

[50] Q. Fan, N. Ansari, Application aware workload allocation for edge computing-based IoT, IEEE Internet Things J. 5 (3) (2018) 2146–2153.

[51] H. Jayakumar, A. Raha, Y. Kim, S. Sutar, W.S. Lee, V. Raghunathan, Energy-efficient system design for IoT devices, in: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, 2016, pp. 298–301.

[52] N. Kaur, S.K. Sood, An energy-efficient architecture for the internet of things (IoT), IEEE Syst. J. 11 (2) (2015) 796–805.

[53] D.S. Kolovos, A. García-Domínguez, L.M. Rose, R.F. Paige, Eugenia: towards disciplined and automated development of GMF-based graphical model editors, Softw. Syst. Model. (2015) 1–27.

[54] OMG, OMG Object Constraint Language (OCL), Version 2.3.1, 2012, URL http://www.omg.org/spec/OCL/2.3.1/.

[55] Obeo, Acceleo project , 2012,.

Chapter 7

SimulateIoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments

“Never forget what you are, for surely the world will not. Make it your strength. Then it can never be your weakness. Armour yourself in it, and it will never be used to hurt you.”

A Game of Thrones (1996)
Martin, George R. R.

Authors: José A. Barriga, José M. Chaves-González, Arturo Barriga, Pablo Alonso, and Pedro J. Clemente
Title: Simulate IoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments
Year: 2023
Journal: Journal of Object Technology

Quality (JCR): Q4
DOI: 10.5381/jot.2023.22.1.a6.

Simulate IoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments

José A. Barriga, José M. Chaves-González, Arturo Barriga, Pablo Alonso, and Pedro J. Clemente
University of Extremadura, Quercus Software Engineering Group (http://quercusseg.unex.es), Spain

ABSTRACT Aiming to optimise the performance of the computing layers of Internet of Things (IoT) systems, one of the most widespread techniques is the well-known task scheduling. Not only to develop but also to put task scheduling techniques in production, they have to be tested. Nevertheless, IoT systems are complex scenarios with high technological heterogeneity. Thus, testing task scheduling methods in the IoT context involves an investment of money, time and effort in acquiring devices, their configuration, deployment, etc. To avoid this, the system can be simulated and tests can be conducted through these simulations. Moreover, the underlying technical complexity of IoT systems can be reduced by increasing the abstraction level from which these systems are designed. Simulators based on model-driven development can help both to test and tackle the technological complexity of IoT systems. In this paper, a Domain-Specific Language based on SimulateIoT is proposed for the design, code generation and simulation of IoT systems for the assessment of task scheduling methods. Simulations include the generation and offloading of workflow-based tasks, the components required to handle these tasks, as well as the required resources to integrate the users' task scheduling methods in the simulations. All this, while providing an infrastructure based on the cloud-to-thing continuum paradigm on which to deploy and test these task scheduling environments, i.e., simulations can include the mist, edge, fog and cloud layers and the federation between them. In addition, a case study focused on an Industrial IoT (IIoT) system is illustrated to show the applicability of the proposed simulator.

KEYWORDS IoT, Model-driven development, Simulation, Task scheduling, Cloud-to-thing continuum.

1. Introduction

The Internet of Things (IoT) is being exploited in several areas such as smart-cities, home environments, agriculture, industry, intelligent buildings, etc.(Siow et al. 2018). In this regard, IoT applications can be very different from each other and therefore have different requirements and needs such as specific Quality of Service (QoS) (Samann et al. 2021) or Service-Level-Agreement (SLA) (Girs et al. 2020).

In order to satisfy these requirements, cloud computing emerged. Thus, supporting the rapid growth of users and applications, and providing them with elastic services such as Infrastructure-as-a-Service (IaaS), Platforms-as-a-Service (PaaS), and Software-as-a-Service (SaaS) with minimum resource consumption (Qian et al. 2009; Rashid & Chaturvedi 2019). However, due to the rapid growth of the IoT and the increasing demand for a better QoS, fog computing emerged (H. & V. 2021). Nearer of the edge/mist computing, although with fewer computing resources than the cloud (H. & V. 2021), this computing layer is able to provide better QoS to specific IoT applications and users such as IoV (Internet of Vehicles) (Yu et al. 2018) or IIoT (Industrial Internet of Things) delay-sensitive applications (Aazam et al. 2018). Thus, different computing layers (cloud, fog, edge and mist) coexist in the IoT providing different services to the system, from the cloud layer to the end devices (mist/IoT layer). Furthermore, the nodes that form each of these layers can be federated, acting as a single entity instead of isolated nodes, including nodes that belong to different computing layers conforming cloud-fog-edge

JOT reference format:
José A. Barriga, José M. Chaves-González, Arturo Barriga, Pablo Alonso, and Pedro J. Clemente. *SimulateIoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments*. Journal of Object Technology. Vol. 22, No. 1, 2023. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2023.22.1.a6

heterogeneous federations (Bittencourt et al. 2018; Kar et al. 2022; Mijuskovic et al. 2021). Consequently, the cloud-to-thing continuum paradigm emerges, which could be defined as the coordination of services and resources between the different computing layers mentioned above. Allowing data flow across cloud data centers, intermediary nodes like edge or fog nodes, and end-user devices, facilitating more efficient, responsive, and resilient computing solutions (Bittencourt et al. 2018).

While the IoT infrastructure was being developed and enhanced, different techniques for optimally managing their resources were also developed and proposed. One of the most widespread techniques is the well-known task scheduling (Arunarani et al. 2019a; Alizadeh et al. 2020). Task scheduling is often applied to distributed computing environments, such as IoT systems that rely on an architecture based on the above described cloud-to-thing continuum, where services are decomposed into a set of tasks which have to be processed by the computing nodes of this federation (Singh et al. 2017; Hosseinioun et al. 2022). Note that in this communication, task refers to an individual unit of work or a specific job that needs to be performed. This can include data collection, data processing, control commands, or other computational processes. In this context, task scheduling proposals aim to schedule the processing of these tasks, thus optimising the system and the use of system resources from different perspectives, e.g., there are proposals that aim at reducing the makespan (time required to process a task) (S. Gupta et al. 2022; Al-Maytami et al. 2019), optimising the system energy consumption (Ding et al. 2020; Sandhu et al. 2021), the cost of task processing (Shu et al. 2021; Gazori et al. 2020), etc.

However, testing is required during the development stage of these proposals, besides these tests have to be isolated as otherwise, they could affect the system in production. Furthermore, novel task scheduling proposals are often compared with existing ones in order to better determine the strengths and limitations of these proposals. Consequently, this implies an investment of money, time and effort in the acquisition of devices, their configuration, deployment, etc. However, these IoT systems can be simulated, and the task scheduling proposals deployed, tested and analysed in these simulated systems, thus avoiding the aforementioned costs in device acquisition, configuration, etc. For instance, the study (Hosseinioun et al. 2022) reports that 90% of the task scheduling proposals applied to fog computing environments use simulators for the aforementioned purposes.

Note that in the context of this communication, simulation refers to the procedure of creating and deploying a digital replica of a real-world system, without having to materialise it physically. On the other hand, testing denotes the application of specific scenarios or conditions to a software component or a system. The purpose of these tests is to reproduce possible situations the system might encounter and then observe and analyse the responses of the system under these conditions. Therefore, in this communication, the intention is to facilitate the performing of these tests by means of simulations.

On the other hand, as described above, IoT systems present a high technological heterogeneity and a complex infrastructure. However, increasing the abstraction level from which the IoT systems are designed helps to tackle the underlying technological complexity. In this regard, model-driven development (MDD) can help to both reduce the IoT application time to market and tackle the technological complexity to develop IoT applications (Barriga et al. 2023).

In this regard, SimulateIoT (Barriga et al. 2021) is a simulator based on model-driven development that makes it possible to design and simulate IoT systems. The IoT systems designed with SimulateIoT can include different IoT nodes such as cloud, fog, or edge nodes and multiple computing services such as Complex Event Processing (CEP) services, publish/subscribe services or storage services. However, SimulateIoT is not able to simulate a suitable IoT infrastructure to test task scheduling proposals.

In this communication, SimulateIoT (Barriga et al. 2021) is extended towards the cloud-to-thing continuum paradigm for task scheduling assessments. Thus, the simulator proposed includes the main concepts of task scheduling (federations, tasks generation and processing, etc.) to model, generate and simulate IoT systems with the required infrastructure to support task scheduling, allowing users to deploy, test, compare and analyse their task scheduling proposals.

Note that the content described in this communication only focuses on describing new contributions or features added as part of the extension. Therefore, all the content in this communication is novel, although some references to SimulateIoT are included where necessary to describe some aspects of the new contributions.

The main work contributions are the following:

- The extension of the metamodel of SimulateIoT towards task scheduling and the cloud-to-thing continuum. This extension provides users with a metamodel that enables the design of models based on IoT systems with task scheduling capabilities. In addition, this extended metamodel enables users to model cloud-to-thing continuum infrastructures on which to deploy and test these task scheduling features.
- The extension of the model-to-text (M2T) transformations of SimulateIoT towards the task scheduling and the cloud-to-thing continuum. This extension ensures that the M2T transformations required to generate and simulate the systems modelled conform to the extended metamodel.
- The extension of the concrete syntax of SimulateIoT towards task scheduling and the cloud-to-thing continuum. This extension enables users to design, in a graphical manner, the IoT system models conform to the extended metamodel.
- A case study to validate and show the applicability of the proposal.

The rest of the paper is structured as follows. In Section 2, we give an overview of existing IoT simulation approaches centred on both low-level and high-level IoT simulation environments. Next, Section 3 gives a holistic view of the task scheduling and cloud-to-thing continuum model envisaged and the extended simulator. Next, Section 4 presents the extended simulator taking into account the design and implementation stages including the new metamodel and the graphical editor. In Section 5, the M2T transformations from the extended simulator models to code are addressed. In Section 6 the simulation outputs, possible tests and assessments are illustrated. In Section 7, a case study to show the applicability of the extended simulator is presented. Finally, Section 8 concludes the paper.

2. Related Works

A large amount of IoT simulators are available in the literature. However, only a few of them allow the simulation of IoT systems with task scheduling features. Below, those most relevant to the proposal carried out in this communication are addressed. Note that the review of these first related works is mainly focused on the task scheduling features that the simulators are able to simulate.

iFogSim (H. Gupta et al. 2017) is one of the most popular IoT simulators in literature. It is an extension of CloudSim (Calheiros et al. 2011), although it is focused on the simulation of the fog layer of the system. It is able to simulate the cloud, fog and edge layer of an IoT system, simulating hardware features, such as the CPU or memory of each device, network features such as the delay and bandwidth between devices, federation between the fog and the cloud nodes, etc. As for task scheduling, it facilitates the design of workflows using DAGs (Directed Acyclic Graphs), which are mathematical structures suitable for representing them. Moreover, iFogSim allows the simulation of the processing of tasks (those related to the above-mentioned workflows). In this way, users can design applications and specify the tasks they offload during the simulation. However, this simulator does not provide knowledge about the availability (status) of nodes, links between nodes or the tasks' waiting time, i.e. the time that a task needs to wait until its processing.

WorkflowSim (Chen & Deelman 2012) is another simulator based on CloudSim, although it is focused on simulating the workflow scheduling. In this way, this tool allows users to simulate the processing of these workflows, including task processing fails, to test several algorithms and policies (although it does not include resources focused on allowing the integration of users' proposals), and all the elements included in CloudSim. Although this tool is interesting because is mainly focused on task scheduling purposes, it was published in 2012 and is no longer maintained, so nowadays it is deprecated. Additionally, it does not support current elements related to IoT systems such as edge nodes, fog nodes, federations between nodes, etc.

YAFS (Lera et al. 2019) is a simulator whose main purpose is to simulate the deployment and execution of applications in a Cloud-fog IoT environment. In this way, users can analyse which is the best allocation of applications and resources strategies, the best network routing strategies for the offloaded data of the deployed applications and also the best scheduling strategy. In order to allow users to model the tasks that constitute an application, DDFs (Distributed Data Flows) are used, which are similar to workflows and DAGs. However, this simulator does not include some relevant data about task processing such as the time required to process a specific task or the status of the links that inter-connect each node of the simulation.

ScSF (Rodrigo et al. 2018) is a simulation tool that focuses only on task scheduling purposes. So, a positive aspect of this tool is that it is not only focused on IoT systems but focuses on any system with task scheduling needs, offering its utilities to a broad spectrum of users. But on the contrary, without offering specific concepts that could be found in an IoT system. In this way, ScSF includes a set of modules that takes as input a system model (processors) and the workflows to schedule. Then, it reports as output the scheduling of each workflow in the processor system taken as input.

These IoT simulators allow the simulation of IoT systems as well as the performance analysis of task scheduling algorithms running on top of these simulations. The most similar related work to the proposal presented in this paper is WorkflowSim. However, each simulator has its own advantages and disadvantages for specific use cases. Consequently, a thorough analysis tailored to the user's specific needs is necessary to select the most suitable simulator for their proposal.

So, in order to compare simulators several quality indicators could be taken into account such as the range of features or the types of environments it can simulate, its reliability, speed, flexibility or its learning curve. Following, several distinguishing features that set apart the proposal presented in this paper from WorkflowSim and the other simulators discussed in this section are highlighted.

1. The proposed simulator is a hybrid simulator/emulator, i.e. it generates and deploys the real architecture of the modelled IoT system (emulation), and simulates some processes related to task scheduling such as the generation of tasks, their offloading to the system or their processing. The rest of the simulators described above base their results on mathematical models. Note that, with the term mathematical model we refer to a set of algorithms that simulate the behaviour of a concrete physical device or system. However, they do not deploy real component architectures on which to simulate and test task scheduling processes. Although mathematical models have been widely and successfully used over time, relying only on mathematical models could affect the trustworthiness of the simulation and testing results due to the inherent limitations of these models in accurately reflecting reality, especially when dealing with complex systems (Sterman 2002; Saltelli & Funtowicz 2014; Fisher et al. 2019). For this reason, trustworthiness is one of the open challenges in the field of IoT simulation (Mishra et al. 2012; Gluhak et al. 2011; Chernyshev et al. 2018). However, by emulating part of the IoT system this gap can be mitigated (McGregor 2002; Erazo & Liu 2013).

2. The proposed simulator is based on the MDD, addressing the design of the simulations from a high level of abstraction. It focuses on the high-level concepts of the IoT and task scheduling systems domain and their relationships, rather than on low-level details.

3. The proposed simulator is updated to current simulation

and testing needs, e.g., it allows the federation of nodes regardless of the computing layer they belong to. Thus, allowing cloud-fog-edge federations (cloud-to-thing continuum infrastructure).

Finally, Table 1 shows a summary of the main features of each related work included in this section together with the proposed SimulateIoT extension. The meaning of each column in Table 1 is described below.

- Simulator: Name of the simulation tool or framework.
- Task Modelling: Indicates the type of task modelling supported by the simulator (e.g., DAG-based workflows).
- Failure Modelling: Specifies if failure modelling is supported or not.
- Task Optimisation: Indicates if task optimisation techniques are supported.
- Task Scheduling: Specifies if task scheduling capabilities are available.
- Task Queueing: Indicates if task queueing is supported.
- Network Delay: Describes the model or type of network delay that the simulator supports.
- Network Bandwidth Model: Specifies the model or approach used to simulate network bandwidth.
- Node Federation: Indicates if the simulator supports federations of edge, fog and/or cloud nodes.
- Underlying Simulation Model: Describes the underlying simulation model or approach used.
- Provides Integration API: Specifies if the simulator provides an integration API for external systems such as user proposals (e.g. task scheduling proposal).
- Focus: Describes the main focus or application domain of the simulator.
- Case Use Definition: Specifies the format or method for defining use cases in the simulator.

As for the notation used in Table 1, *Limited* means that the simulator provides some options and does not allow the user to integrate their own proposals. On the other hand, *User-proposal* means that the simulator provides the flexibility to incorporate user-suggested solutions or strategies. Note that these concepts, i.e., *Limited* and *User-proposal*, are used in several columns, where they mean the same but in the context of the respective column.

Finally, some findings related to Table 1 are highlighted below. Of the four simulators included, only YAFS, ScSF, and SimulateIoT allow users to integrate their custom solutions in terms of task scheduling or optimisation algorithms. The lack of this feature significantly compromises the practical utility of a simulator. Both SimulateIoT and iFogSim support the modelling and simulation across the four computing layers: mist, edge, fog, and cloud. Yet, SimulateIoT stands out as the only one that enables federation among these layers, making it the sole simulator capable of simulating the nuances of current cloud-to-thing IoT system infrastructures. In terms of network modelling, only iFogSim and SimulateIoT offer modelling of delay, bandwidth, and unidirectional links. Note that such features are key in any simulator oriented to the simulation of IoT systems with task scheduling capabilities. In this context, it should be noted that iFogSim is not specifically designed for testing task scheduling systems (thus lacking several task scheduling simulation capabilities). So, only SimulateIoT is specifically oriented towards task scheduling testing and provides the capabilities to model these networking features.

3. The Cloud-to-Thing Continuum and Task Scheduling Model, and the Resulting Simulator

This section aims to introduce the proposed simulator as well as the systems that it can simulate. Furthermore, as this work is an extension of SimulateIoT (Barriga et al. 2021), the aim of this section is also to outline the new features added as part of this extension. Thus, differentiating between what was previous work (SimulateIoT) and what is new. Later, with the aim of conveying a “big picture” of the work carried out in this communication, the extended SimulateIoT capabilities based on this model are shown by means of a generic simulation overview.

SimulateIoT and therefore the proposed simulator, are based on the MDD, which is an emerging software engineering research area that aims to develop software guided by models based on the metamodeling technique. Metamodeling is defined by four model layers (see Figure 1). Thus, a model (M1) conforms to a metamodel (M2). Moreover, a metamodel conforms to a metametamodel (M3) which is reflexive (Atkinson & Kuhne 2003). So, 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 that conforms to a metamodel. Then, from these models, it is possible to generate totally or partially the application code (M0 - code) by M2T transformations (Sendall & Kozaczynski 2003). Thus, high-level definitions (models) can be mapped by M2T transformations to specific technologies (target technology). Consequently, the software code can be generated for a specific technological platform, improving technological independence and decreasing error proneness.

Therefore, in order to extend SimulateIoT towards the cloud-to-thing continuum paradigm and to the task scheduling, it is required to work in these metamodelling layers. Specifically, it is required to extend: 1) The metamodel or abstract syntax (M2), 2) The graphical concrete syntax, the element that makes it possible to graphically design models (M1) from the metamodel (M2) and 3) model-to-text transformations, the element that carries out the code generation (M0) from models (M1).

Nevertheless, prior to extending SimulateIoT (Barriga et al. 2021) towards the cloud-to-thing continuum and task scheduling, it is crucial to identify the main concepts of these systems. This characterisation results in a conceptual model, forming the backbone of this study. Aiming to introduce the work carried out in this communication, this model is subsequently introduced. Moreover, an outline of the enhanced SimulateIoT, the resulting simulator after extending it towards this model, is also provided.

Table 1 Feature summary of related works and presented proposal.

Simulator	Task Modelisation	Failure Modelling	Task Optimisation	Task Scheduling	Task Queueing	Network Delay	Network Bandwidth Model	Node Federation	Underlying Simulation Model	Provides Integration API	Focus	Case Use Definition
IFogSim	DAG-based DDFs	Not modelled	Not modelled	Not modelled	Not modelled	Yes	Unidirectional links (different delay depending on origin and destination)	Supports Fog and Cloud federations	Mathematical model	No	IoT systems across all layers (Mist, Edge, Fog, and Cloud)	Graphical Interface
WorkflowSim	DAG-based Workflows	Yes, several types	Yes, limited	Yes, limited	Yes	Yes	Not modelled	Not modelled	Mathematical model	No	Cloud	API
YAFS	DDFs	Yes, network failures model	Not modelled	Yes, user proposal	Yes	Yes	Bi-directional links (same delay regardless of origin or destination)	Supports Fog and Cloud federations	Mathematical model	Yes	IoT systems at Fog and Cloud layers	JSON
ScSF	Workflows	Not modelled	Yes, user proposal	Yes, user proposal	Yes	Not modelled	Not modelled	Not modelled	Mathematical model	No	Processing Environment s	API
SimulateIoT	DAG-based Workflows	Yes, manually induced	Yes, user proposal	Yes, user proposal	Yes	Yes	Unidirectional links (different delay depending on origin and destination)	Supports heterogeneous federations across Edge, Fog, and Cloud	Mathematical model and emulation layer (Infrastructure)	Yes	IoT systems across all layers (Mist, Edge, Fog, and Cloud)	Graphical Interface

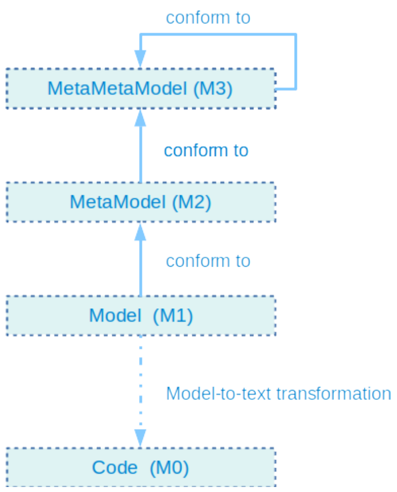


Figure 1 The four layers of metamodeling. In SimulateIoT (Barriga et al. 2021): a) M3 is Ecore, b) M2 is SimulateIoT Metamodel c) M1 is a model conforms to SimulateIoT Metamodel and d) Code is generated using the M2T transformations defined in SimulateIoT approach.

3.1. The Proposed Cloud-to-Thing Continuum and Task Scheduling Model

This section addresses the conceptual model on which the SimulateIoT extension is based. It provides an introduction to the key concepts that constitute this model: `Task`, `Task App`, `Task Node`, `Networking Node`, `Task Processor` and `Task Scheduler`.

3.1.1. Task Task scheduling systems are based on tasks, i.e. in this kind of environment, there are nodes that generate, offload (to the rest of the system), schedule and process tasks. In this communication, tasks are included by means of workflows (Wu et al. 2015; Arunarani et al. 2019b), as is common in literature (Yao et al. 2021; Asghari et al. 2021; NoorianTalouki et al. 2022; Ahmad et al. 2021). So, users can define tasks by means of workflows that the nodes designed for these purposes will generate, offload, schedule or process.

Figure 2 shows a graphical representation of a workflow. This workflow represents four tasks, `Task A`, `Task B`, `Task C` and `Task D`. In this workflow, each node (circle) represents a task and each edge represents the dependency between these tasks.

Concerning task dependency, a dependent task cannot be processed until all other predecessor tasks have been processed, e.g., in the workflow shown in Figure 2, `Task B` and `Task C` cannot be processed until `Task A` has been processed. On the other hand, `Task D` cannot be processed until `Task B` and `Task C` have been processed. This is because once the tasks are processed, they can generate results that may be required by subsequent tasks in the processing pipeline. These results are the basis of the dependency between tasks, represented as

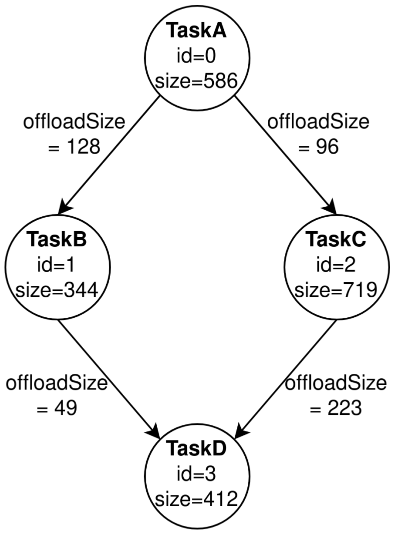


Figure 2 Graphical representation of a workflow.

relationships (edges) between nodes (tasks) in workflows (see Figure 2).

In terms of the attributes of a task, in this communication, each task has a name (e.g. `Task A`), an `id` and a `size` (bytes). Besides, each edge has a source task, a target task and a `offloadSize` which represent the size (bytes) of the data that have to be transmitted from the source task, once processed, to the target task, i.e. from the node that processes the source task to the node that processes the target task. Note that the `offloadSize` attribute represents the offload size (bytes) of the processing results of a task.

3.1.2. Task App In task scheduling scenarios, it is common to deploy applications that provide services at the fog and cloud layers. These applications are the ones that generate and offload tasks (the services they provide are decomposed into tasks or sets of tasks, i.e. workflows). In this model, these applications are included by means of the concept `Task App`, which can be deployed in the different nodes of the fog and cloud layers.

3.1.3. Task Node Traditionally, the edge and mist layers have had a different role than the fog and cloud layers. While the fog and cloud layers have had a role of providing computing resources or services to the end devices of the system (mist and edge layers), the mist and edge layers were constrained in terms of hardware and were limited to consuming these resources and services.

However, there are currently some end devices that do not face the aforementioned hardware constraints (such as mobile phones, personal computers, etc.). Therefore, their use does not have to be limited to consuming the services and resources of the fog and cloud layers but can help these layers in the provision of these services and resources to the rest of the system. Besides, in some situations it provides a better QoS than fog and cloud layers, since these devices are in the mist or edge layer itself and therefore close to the rest of the end devices. So, they are able to provide better latency, request-response time, etc. This new paradigm is called cloud-edge computing (Pan & McElhannon 2018).

In this context, the `Task Node` is designed to include in the task scheduling model this new paradigm of federation between computing layers. Thus, `Task Nodes` are conceived as nodes that belong to the edge and mist layers of the system but can be federated with cloud and fog nodes, thus providing task execution services to `Task Apps` and being able to process the tasks they generate.

On the other hand, as could be edge or mist devices that generate tasks requiring their processing, as `Task Apps`, `Task Nodes` are also designed to generate and offload tasks.

3.1.4. Networking Node

Task scheduling is frequently implemented in environments that operate on a cloud-to-thing continuum infrastructure. In this kind of system, nodes can be federated, acting as a single entity rather than isolated nodes. Federations are addressed in this model by means of `Links`, i.e. the connections between the different nodes that belong to a federation.

The `Networking Node` is the component responsible for managing these links. This model envisages a `Networking Node` for each federated (linked) node. Thus, each `Networking Node` handles the links whose source is the node where the `Networking Node` is integrated. Moreover, links are designed as unidirectional. Consequently, two links are needed to allow two components to interact with each other.

On the other hand, SimulateIoT is a hybrid simulator/emulator of IoT systems. So, SimulateIoT simulations have to be deployed over a real (or virtualised) network. Thus, without the need to simulate the latency and bandwidth of links. However, as aspects such as delay and bandwidth among nodes are critical aspects in task scheduling systems (Jamil et al. 2022), users could require, for testing purposes, specific latency and bandwidth between nodes. Configuring the network where simulations will be deployed could be a tedious, error-prone and costly task. Thus, this model also envisages the possibility to specify the delay and bandwidth of the aforementioned links, being the `Networking Node` the component which will ensure that these networking aspects are met during simulation.

Finally, note that this model envisages fully connected federations, i.e. all nodes belonging to a federation are connected to each other.

3.1.5. Task Processor

The `Task Processor` is the component that performs the processing of tasks. So, the `Task Processor node` is integrated at the deployment (of the simulation) stage into those edge (*Task nodes*), fog and cloud nodes that are modelled by the user to provide task processing services to the rest of the system.

Note that to suitably simulate the processing of tasks, this model envisages the possibility of assigning hardware resources to each node with processing capabilities, i.e. to the `Task Processors`. Thus, users can model aspects related to the hardware of each node, such as their CPU or RAM.

3.1.6. Task Scheduler

This component is the most relevant since it is where the simulator will integrate users' task scheduling proposals, as described in the following sections. Thus, the `Task Scheduler` is the node that receives and schedules (by means of the users' task scheduling proposal), the tasks offloaded to the system.

Since task scheduling algorithms can use several data sources and data types as input to perform their schedules (Bansal et al. 2022), this component is designed to provide users' proposals with resources to request data from several nodes of the simulation. Note that users' proposals can interact with these resources and therefore with the rest of the simulation by means of a REST API (which belongs to the aforementioned provided resources). Thus, the integration of users' proposals with the rest of the simulation is simplified.

This API allows users' proposals to perform four main requests: 1) request the offloaded tasks for their scheduling, 2) request data such as the current delay or available bandwidth between each node (links status), 3) request data related to the hardware usage (CPU, RAM, etc.) of each node, and 4) request the return of the scheduled tasks (once scheduled) to the system for their processing.

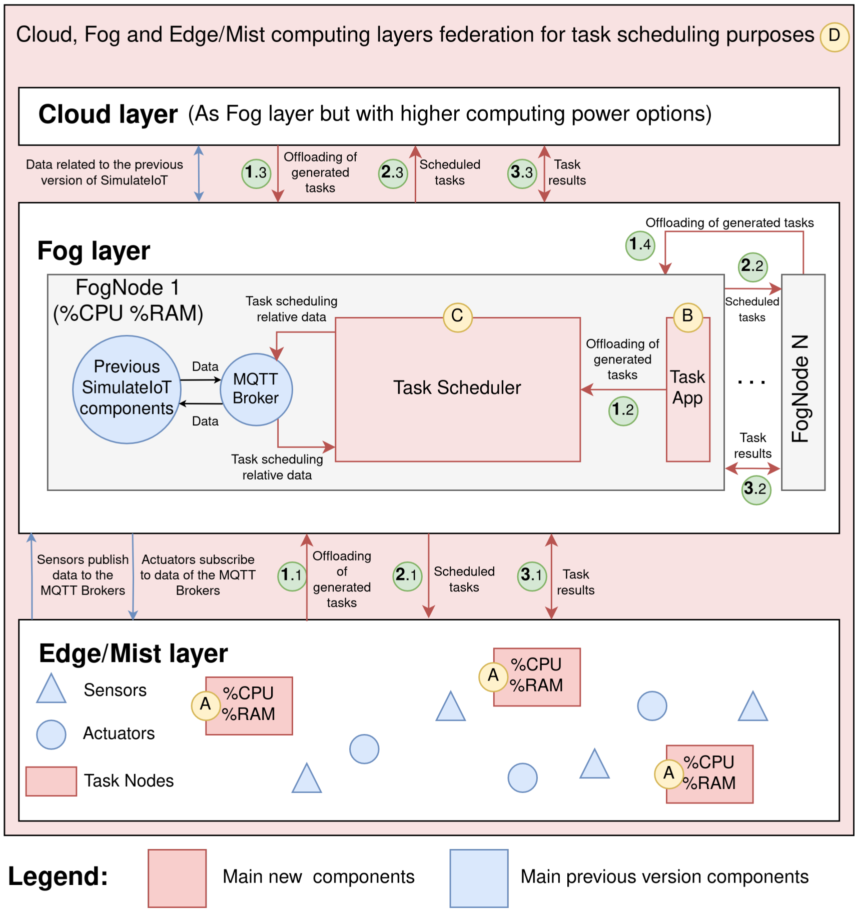
Note that in this model, these requests are limited to the nodes that belong to the same federation, i.e. those nodes that belong to different federations can not interact. On the other hand, this model envisages one `Task Scheduler` per federation. However, the `Task Scheduler` of each federation can integrate a different user scheduling proposal.

Thus, by extending SimulateIoT towards this task scheduling model, users will be able to deploy, analyse, compare, etc. their task scheduling proposals on a simulated IoT system without the need for investment in device acquisition, configuration and deployment. In Section 3.2 that follows, an overview of the resulting simulator after extending it towards this model is provided.

3.2. Overview of the Resulting Simulator

Figure 3, provides a representation of a generic simulation deployment using the extended version of SimulateIoT. This figure also distinguishes between the components developed in this work (marked in red) and those inherited from the original SimulateIoT (marked in blue). The significant enhancements, those related to the concepts belonging to the model defined in Section 3, are explained below in the context of Figure 3. Note that to facilitate referencing the concepts depicted in Figure 3 within the text, the corresponding labels included in the figure will be used (e.g. ①).

Firstly, tasks are generated and offloaded to the system by the `Task Nodes` Ⓐ and the `Task Apps` Ⓑ. The offloading of these tasks is represented in Figure 3 by (1.1) offloading performed by `Task Nodes` to the fog layer; (1.2) and (1.3) offloading performed by `Task Apps` deployed on fog nodes to the fog layer; (1.4) offloading performed by `Task Apps` deployed on



Cloud, Fog and Edge/Mist computing layers federation for task scheduling purposes D
Cloud layer (As Fog layer but with higher computing power options)
Data related to the previous version of SimulateIoT
1.3 Offloading of generated tasks
2.3 Scheduled tasks
3.3 Task results
Fog layer
Offloading of generated tasks 1.4
2.2
FogNode 1 (%CPU %RAM)
Task scheduling relative data
C
B
Previous SimulateIoT components
Data
MQTT Broker
Data
Task Scheduler
Offloading of generated tasks
1.2
Task App
Scheduled tasks
. . .
Task results
3.2
FogNode N
Task scheduling relative data
Sensors publish data to the MQTT Brokers
Actuators subscribe to data of the MQTT Brokers
1.1 Offloading of generated tasks
2.1 Scheduled tasks
3.1 Task results
Edge/Mist layer
Sensors
Actuators
Task Nodes
A %CPU %RAM
A %CPU %RAM
A %CPU %RAM
Legend:
Main new components
Main previous version components

Figure 3 A generic simulation generated by using M2T transformations from a model defined with the proposed simulator.

cloud nodes to the fog layer.

Note that in Figure 3 all the tasks are offloaded to the fog layer as `FogNode 1` is the node where the `Task Scheduler` (C) is deployed.

Once tasks have been offloaded, they are scheduled by the *Task Scheduler* (C) and sent back to the system for their processing (2.1), (2.2) and (2.3).

Then, the `Task Processor` of each fog, cloud and `Task Node`, which is not directly represented in Figure 3, carries out the processing of each task and returns the result to the node that is waiting for it due to the dependency between tasks (3.1), (3.2) and (3.3).

Note that during the simulation, the generation, offloading, scheduling, and processing of each task by each node occur concurrently and adhere to the user-defined design (model).

Finally, it should be highlighted that the behaviour of each node and the overall simulation exhibit a higher level of complexity compared to what is depicted in this overview. For the sake of clarity, there are interactions and concepts that, as in the case of the `Task Processor`, are not directly depicted in Figure 3. For instance, the `Networking Node` or the interactions between the `Task Scheduler` and the nodes belonging to its federation (D) to gather/share their status (available CPU and RAM).

Further details are addressed in following Sections 4 and 5.

4. Extensions of Metamodel and Concrete Syntax

The proposed simulator, as an MDD approach, is composed of three main elements: 1) metamodel or abstract syntax, 2) graphical concrete syntax and 3) model-to-text transformations. This section describes the proposed simulator metamodel and concrete syntax.

4.1. Metamodel Extensions

A Metamodel captures the concepts and relationships in a specific domain in order to model partially reality (Selic 2003). Then, it is possible to design models conforming to this Metamodel. These models can be used to generate the total or partial application code. Thus, the software code could be generated for a specific technological platform, improving its technological independence and decreasing error proneness.

The SimulateIoT metamodel (Barriga et al. 2021) gathers the core concepts and relationships related to the IoT domain, including elements such as sensors, actuators, edge nodes, fog nodes, cloud nodes, databases, complex-event processing services, data definition, topics, message brokers, etc. However, it has not enough expressiveness to simulate IoT systems with task scheduling capabilities and with a cloud-to-thing continuum infrastructure. Therefore, the metamodel of the proposed simulator is an extension of the SimulateIoT metamodel with enough expressiveness to define these kinds of IoT systems (entities and services described in Section 3.1).

Figure 4 shows an excerpt of the proposed simulator metamodel. Note that the new classes and relationships included are numbered and highlighted in blue colour. Besides, note that Figure 11 (Appendix A) shows the complete metamodel, with the elements relating to the extension carried out also highlighted in blue.

This section does not aim to describe how these components work internally, which is addressed in Section 5, where M2T transformations are addressed. Finally, note that in this section, to better describe the elements of the metamodel, the numerical labels shown in Figure 4 are used below as references in the text. These references are used by means of the expression `[class name]` $\textcircled{x}$, where x is the label associated with the `[class]` in Figure 4.

In order to extend the SimulateIoT metamodel towards task scheduling and the cloud-to-thing continuum, first of all, the task generation related components, i.e. the Task Node and the Task App, have been included in the metamodel by means of the classes `TaskNode` $\textcircled{3}$ and `TaskApp` $\textcircled{2}$ respectively. The workflow concept has also been added by means of the `Workflow` class $\textcircled{1}$ and related to the two previously mentioned classes with the aim of allowing the user to model which workflow will be generated by each modelled Task Node and Task App during the simulation. Note that each workflow will be generated every period of time, which can be specified by the `generation_period` (seconds) attribute.

To allow the user to model the different workflows that will be generated, in addition to the `Workflow` class, the `Task` $\textcircled{1.1}$ and `Edge` $\textcircled{1.2}$ classes have also been included. Thus, the `Task` class represents a workflow node and the `Edge` class represents the dependency between these tasks. Note that in these classes the user can also model the size of each task (`size` attribute) and the size of the processing results of each task (`offload_size` attribute).

On the other hand, the metamodel is extended from the `Hardware_specification` $\textcircled{5}$ class to allow the user to model in a more detailed manner the hardware resources of each modelled node. To this end, this class is related to the `CPU` $\textcircled{5.1}$ and `RAM` $\textcircled{5.2}$ classes, also included as part of this extension to allow the user to model these hardware aspects for each node.

Finally, the metamodel is extended with the `Federation` $\textcircled{4}$ class, which allows the user to federate the different nodes of the modelled system. To this end, federations are composed of `Links` $\textcircled{4.1}$, a class that allows modelling the characteristics of the different links between the federated nodes. Particularly, the classes `Delay_specification` $\textcircled{4.2}$ and `Bandwidth_specification` $\textcircled{4.3}$ allow the user to model these characteristics for each link.

Extending the metamodel of SimulateIoT with these classes and relationships, the concepts related to the task scheduling model defined in 3.1 are gathered by the metamodel. Therefore, the required expressiveness to model IoT systems with task scheduling capabilities is achieved.

4.2. Graphical Concrete Syntax and Validator Extensions

Model-driven development allows designing models conforming to a metamodel by means of concrete syntax. Concrete syntax refers to the specific notation used to depict instances of a model. The concrete syntax can visually or textually represent the abstract notions and relationships that are defined within the metamodel. It could be defined as graphical concrete syntax (e.g. using GMF, Eugenia or Sirius) or textual concrete syntax (e.g. using xText). In our approach the Eugenia tool (Kolovos et al. 2015) is used and it makes it possible to generate a graphical concrete syntax conforming to a metamodel.

A graphical syntax is often more intuitive and easier to comprehend at a glance, especially when dealing with complex systems. It provides a high-level overview of a system, which facilitates an understanding of the structure and relationships within the system. However, graphical syntax can become more difficult to manage in terms of automated processing.

In contrast, the textual syntax can be seamlessly processed and integrated with other systems. However, a textual syntax can be more difficult to understand and navigate, especially for individuals who are not familiar with the specific language or notation. It may require more time to interpret and does not provide a visual overview of the system.

Although, both concrete syntax could be developed to define models conform to the metamodel proposed.However, in view of the above, given the complexity of the concepts to be represented, the models to be designed (IoT systems) and the nature of the resulting tool (a simulator), it has been considered that a graphical syntax is more appropriate than a textual one. In addition, the graphical concrete syntax generated for the proposed simulator metamodel is an extension of the graphical concrete syntax defined in SimulateIoT, which is based on Eclipse GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Tools). Figure 5 shows an excerpt from this graphical editor.

It helps users to improve their productivity allowing not only defining models conforming to the proposed simulator metamodel but also their validation. In this respect, metamodels can be extended with constraints based on the Object Constraint Language (OCL) (OMG 2012). OCL is a declarative language developed by the Object Management Group (OMG) for describing rules applicable to the models designed by users conforming

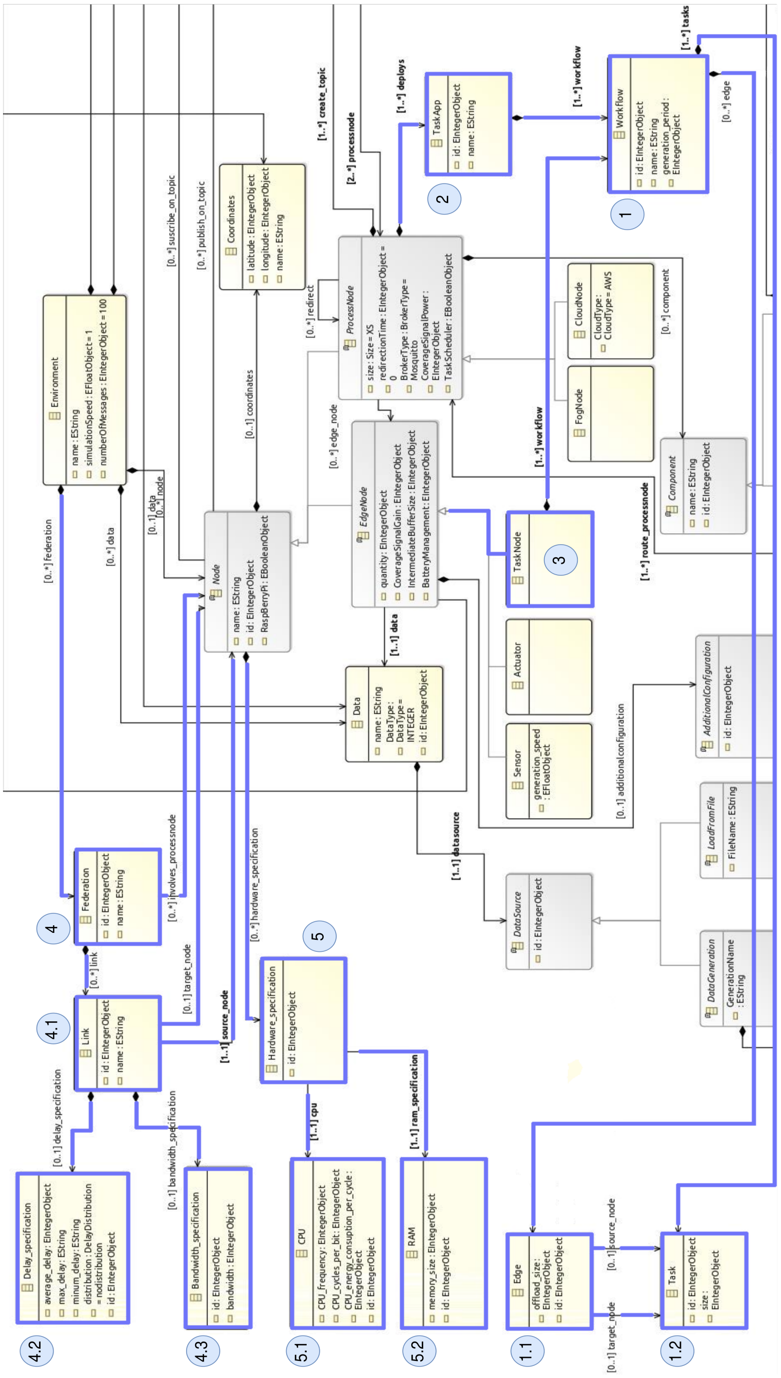


Figure 4 Excerpt of the proposed simulator Metamodel focusing on the task scheduling concepts. The complete metamodel can be found in Appendix A in Figure 11.

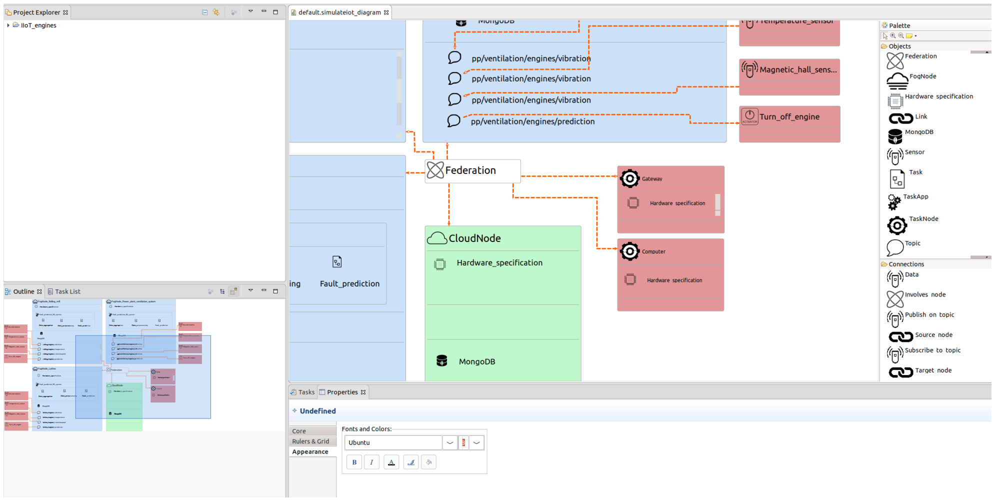


Figure 5 Graphical editor based on Eclipse to graphically design models conforming to the metamodel proposed for the simulator.

to a metamodel. As an example, some of the OCL restrictions defined are shown below.

```
context Task
invariant UniqueId: Task.allInstances()->forAll(
    t1, t2 | t1 <> t2 implies t1.id <> t2.id)
```

This OCL constraint ensures that each task has a unique ID.

```
context Delay_specification
invariant MaxDelayGreaterThanMinDelay: self.
    max_delay >= self.minum_delay
```

This invariant ensures that the maximum delay (max_delay) is always greater than the minimum delay (minum_delay) in each Delay_specification instance.

```
context Bandwidth_specification
invariant ValidBandwidth: self.bandwidth <> null
    and self.bandwidth > 0
```

This invariant ensures that the bandwidth specification is not null and is greater than 0.

To sum up, the graphical concrete syntax developed offers a suitable way to model and validate the IoT environment by using the high-level concepts defined in the SimulateIoT metamodel (Figure 4).

5. Extensions of M2T Transformations

Once the models have been defined and validated conforming to the proposed simulator metamodel (example of a model in Figure 10), a M2T transformation defined using Acceleo (Obeo 2012) can generate the IoT system modelled. This section describes the main extensions included in the M2T transformations of SimulateIoT in order to generate IoT systems simulations with task scheduling capabilities.

For the sake of clarity, this section is divided into the domain-specific concepts identified in Section 3.1 (as in Section 4.1). In this way, each subsection includes the contributions that make it possible to generate the code of each component related to these task scheduling concepts. However, there are no sections dedicated to the `Networking Node` and the `Task Processor`. This is because these components are subcomponents of other elements, such as the Task Node. As a result, they are addressed in the sections pertaining to these primary components.

Acronym	Meaning
TApp	Task App
NN	Networking Node
TN	Task Node
TP	Task Processor
TS/TSN	Task Scheduler Node
SSA	System Status Agent

Table 2 Acronyms used in figures of Section 5.

Note that the descriptions of the components, from a high level of abstraction, are already covered in Section 3 and 4.1. Therefore, the current section omits this high-level description and exclusively delves into the inner workings of the components, offering a low-level perspective.

Finally, note that Table 2 summarises the acronyms used in several of the figures included in this section.

5.1. Task

Section 4.1 describes the extensions carried out to make it possible to model the tasks that will be generated, offloaded and processed by the IoT system. However, the task concept does not become a concrete component or service but is integrated as part of the logic of the rest of the components, e.g. in the `Task App` or the `Task Node`, which need the logic to generate, offload, receive or process tasks. Therefore, the transformations related to this concept are addressed below, in the sections that explore the M2T transformations of the components that are related to tasks, such as the aforementioned.

However, since tasks are managed by these components using the JSON notation, below are the different JSON codes that can be exchanged between components during a simulation.

To illustrate how tasks are generated by the `Task App` or the `Task Node` (components that can generate tasks), Listing 1 shows an example of the JSON code related to the set of tasks that constitute the workflow shown in Figure 2. This JSON code example includes all the necessary fields to represent this workflow, such as the *id* and *name* of the workflow, the node and the component that generate it (field *"generatedBy"*), the tasks that constitute the workflow (field *"nodes"*), the edges (field *"edges"*) and all the data related to these elements. Note that Appendix B, Listing 5, shows the Acceleo code related to the M2T transformations of these tasks.

```
{
  "workflow": {
    "id": 0,
    "name": "exampleWorkflow",
    "generatedBy": {
      "nodeId": "fogA0",
      "generatorId": "taskAppA0",
      "generationId": "0"
    },
    "nodes": [{
        "task": {
          "name": "TaskA",
          "id": "0",
          "size": "586"
        }
      },{
        "task": {
          "name": "TaskB",
          "id": "1",
          "size": "344"
        }
      },{
        "task": {
          "name": "TaskC",
          "id": "2",
          "size": "719"
        }
      },{
        "task": {
          "name": "TaskD",
          "id": "3",
          "size": "412"
        }
      }
        ],
    "edges": [{
        "edge": {
          "id": "0",
          "sourceTaskId": "0",
          "targetTaskId": "1",
          "offloadSize": "128"
        }
            },{
        "edge": {
          "id": "1",
          "sourceTaskId": "0",
          "targetTaskId": "2",
          "offloadSize": "96"
        }
      },{
        "edge": {
          "id": "2",
          "sourceTaskId": "1",
          "targetTaskId": "3",
          "offloadSize": "49"
        }
      },{
        "edge": {
          "id": "3",
          "sourceTaskId": "2",
          "targetTaskId": "3",
          "offloadSize": "223"
        }
      }]}}
```

Listing 1 JSON code to represent a workflow. Specifically, the workflow illustrated in Figure 2.

On the other hand, workflows are sent to the `Task Scheduler` for scheduling purposes. In this regard, the `Task Scheduler` decomposes the workflow to schedule the processing of its tasks. Listing 2 shows the JSON code related to the scheduling of `Task C`, which belongs to the workflow shown in Figure 2 and in Listing 1.

Among the fields of this JSON code, there are data related to the task itself (*id, name, size, offloadSize*), data related to the node that generated it (*generatedBy*), as well as data related to the scheduling of the task (*schedulingData*). Data relating to the scheduling of the task includes the node that has to process it (*processAt*) and at what time its processing has to be performed (*processAt*). Furthermore, since the task scheduling model (Section 3.1) envisages dependency between tasks, this JSON code also includes to which node the processing results have to be sent (*resultsTo*) and also whether the task depends on the processing results of another task (*waitForResults*).

```
{
  "scheduledTask": {
    "id": "2",
    "name": "TaskC",
    "size": "719",
    "offloadSize": "49",
    "generatedBy": {
      "nodeId": "fogA0",
      "generatorId": "taskAppA0",
      "generationId": "0"
    },
    "schedulingData": {
      "processIn": "TaskNodeA",
      "processAt": "2023-1-25 11:29:52",
      "resultsTo": "fogC",
      "waitForResults": {
        "taskId": "1",
        "taskName": "TaskA"
      }
    }}}
```

Listing 2 JSON code related to the scheduling of TaskC of the workflow illustrated in Figure 2.

Once a node performs the processing of a task, it includes in its JSON code some fields related to the results of its processing. Listing 3 shows the JSON code fields added to `Task C` after its processing. Among these fields, there is the time at the task reached the `Task Processor` (*arrivedToTaskProcessorAt*), the time at the processing of the task started (*processingStartedAt*) and the time at the processing of the task finished (*processingFinishedAt*). In short, it includes data that could be useful for the user in the analysis stage.

```
{
        .

    "processingResults": {
        "arrivedToTaskProcessorAt": "2023-1-25 11:28:38"
        "processingStartedAt": "2023-1-25 11:29:54",
        "processingFinishedAt": "2023-1-25 11:29:57"
    }
        .
}
```

Listing 3 JSON code excerpt that shows the fields related to the processing results of TaskC of the workflow illustrated in Figure 2.

As discussed above, since the task scheduling model (Section 3.1) envisages dependency between tasks, once a task is processed, the processing results are sent to the node that is waiting for them, i.e. the node that has to process a task that depends on these results.

Thus, when a node receives processing results, it includes them in the JSON code of the task that is waiting for them. In this way, the exit (last) task of a workflow will include the processing results of the rest tasks of the workflow. Note that this JSON code is the processing result that is returned to the node that generated and offloaded the workflow to the system for its processing (`Task App` or `Task Node`).

Listing 4 shows the JSON code fields related to the processing results of the predecessors of `Task C` (in this example, `Task A`). The data included in this case are the received results related to the processing of the task together with its *id* and *name*. Note that, if a task includes results related to the processing of another task (as is the case in this example where Task C contains the results of Task A) when offloading its processing results, it also includes the results of its predecessors. So, in this example, `Task C` will offload its processing results together with the processing results of `Task A`.

```
{
        .

    "predecessorsProcessingResults": [{
        "task": {
            "id": "0",
            "name": "TaskA",
            "processedIn": "FogA"
            "arrivedToTaskProcessorAt": "2023-1-25 11:26:
   18",
            "processingStartedAt": "2023-1-25 11:28:34",
            "processingFinishedAt": "2023-1-25 11:28:47"
        }
    }]

        .
}
```

Listing 4 JSON excerpt that shows the fields related to the processing results of the predecessor tasks of Task C of the workflow illustrated in Figure 2.

5.2. Task App

Figure 6 shows a generic `Task App` node Ⓑ (represented by a red box) and its main component (element within the red box), the `Workflow Generator` Ⓒ. Figure 6 also shows how the `Task App` is deployed on a fog/cloud node and the interactions that its component could perform with other artefacts of the fog/cloud node and with the rest of the IoT system. Below, the `Task App` node is illustrated by describing its component and its interactions with the rest of the system.

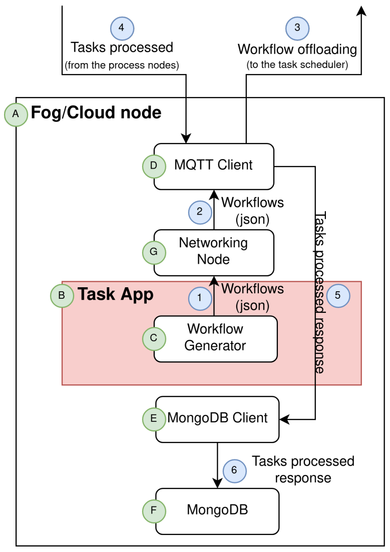


Figure 6 Task App component.

`Workflow Generator` Ⓒ As described in Section 3.1.2, the `Task App` can generate tasks by means of workflows and offload them to the rest of the system. The `Workflow Generator` is the component of the `Task App` whose aim is to generate and offload ① the workflows modelled by the users in the modelling of the system stage.

The `Task App` is not composed of additional components. However, to provide a comprehensive understanding of its role and impact within the entire system, a description of the components associated with the `Task App` is presented below.

`Networking Node` Ⓖ This component simulates the network aspects related to the *bandwidth* and *delay* modelled by users for each `Link`, i.e. the unidirectional connection between two federated nodes. So, as the workflows generated have to be offloaded to the system through a `Link`, the `Networking Node` applies to these workflows the *bandwidth* and *delay* constraints modelled for the `Link` through which they have to be offloaded. Note that the `Networking Node` is described in more depth in Appendix C.

`MQTT Client` Ⓓ The MQTT Client is the component that allows publish/subscribe communication between the components of the system through the MQTT protocol. In this context, it is implied in the offloading of the generated workflows to the rest of the system (interactions ② and interactions ③) and in the reception of the processing results of the tasks (interactions ④ and ⑤).

`MongoDB` Ⓕ and `MongoDB Client` Ⓔ To provide data storage services to the IoT systems simulations, seamlessly to the user, a MongoDB database Ⓕ is deployed on each modelled fog/cloud node during the deployment stage of the simulation. In the same way, a `MongoDB Client` Ⓔ is also deployed to carry out the needed interaction between this database and the task scheduling components. In this context, these two components are employed to store the processing results of the tasks (interactions ④, ⑤ and ⑥).

5.3. Task Node

Figure 7 shows a generic `Task Node` Ⓑ (represented by a red box) and all its components (elements within the red box) and their interactions. The main components of the `Task Node` are the `Workflow Generator` Ⓔ, the `Task Processor` Ⓕ, the `Networking Node` Ⓓ and the `MQTT Client` Ⓒ. Besides, Figure 6 also shows how the `Task Node` is deployed as part of the mist/edge layer and the interactions that its components could perform among them and with other artefacts of the system. Note that in this case, components such as the `Networking Node` or the `MQTT Client` are part of the `Task Node`. In contrast to the `Task App`, the `Task Node` is not deployed on a fog/cloud node (which provides with a `MQTT Client`, etc. to the `Task App`), the `Task Node` is a device itself belonging to the edge layer. Below, the `Task Node` is illustrated by describing each of its components and their interactions with the rest of the system.

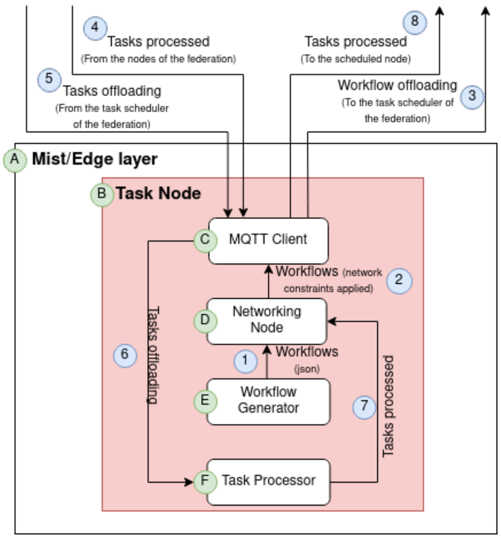


Figure 7 Task Node components.

`Workflow Generator` Ⓔ, `Networking Node` Ⓓ and `MQTT Client` Ⓒ As the `Task App`, the `Task Node` also generates and offloads workflows to the rest of the system. In this context, the behaviour of these components (interactions ①, ②, ③ and ④) is the same as the behaviour already explained in the `Task App` section (Section 5.2).

`Task Processor` Ⓕ `Task Nodes` Ⓑ belong to the edge/mist layer, however, they can be part of the computing power of a federation, thus being able to process tasks. The `Task Processor` is the component of the `Task Node` that performs the processing of tasks. In this way, the `Task Scheduler` could schedule workflows and assign the processing of its tasks to a `Task Node` (interaction ⑤). As these scheduled tasks are offloaded via MQTT protocol, the first to receive them is the `MQTT Client` of the `Task Node` (interaction ⑤). Next, the `MQTT Client` forwards these tasks to the `Task Processor` (interaction ⑥), which performs their processing. Once processed, the `Task Processor` returns the processing results of these tasks to the `MQTT Client`, which sends them to their target nodes, i.e. the nodes waiting for the results of the processed tasks (specified by the field *resultsTo* of the JSON code illustrated in Listing 2). Note that as outgoing data, network constraints are also applied in this context. This flow of data is represented by the interactions ⑦, ② and ⑧. Finally, note that the behaviour of the `Task Processor` is described in more depth in Appendix D.

5.4. Task Scheduler

Figure 8 shows a generic `Task Scheduler` Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the `Task Scheduler` are the `Workflow Buffer` Ⓓ, the `System Status Agent` Ⓖ, the `Task Scheduler API` Ⓔ and the `Task Scheduling Proposal` Ⓕ. Besides, Figure 8 also shows how the `Task Scheduler` is deployed on a fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the fog/cloud node and with the rest of the IoT system. Below, the `Task Scheduler` is illustrated by describing each of its components and their interactions.

`Workflow Buffer` Ⓑ The `Task Apps` and the `Task Nodes` generate and offload workflows to the system. The `Task Scheduler` Ⓑ is the component that first receives these workflows ① with the aim of scheduling the processing of these tasks. In this context, the `Workflow Buffer` holds these incoming workflows ②. Hence, when the `Task Scheduler Proposal` Ⓕ is ready to schedule the processing of a workflow, it retrieves it from this buffer by means of the `Task Scheduler API` (interactions ③, ④, ⑤ and ⑫).

`System Status Agent` Ⓓ The `System Status Agent` aims to gather the status of both the nodes that belong to the same federation as the `Task Scheduler` and the network of the federation. In this context, first, the `Task Scheduling Proposal` requests this data to the `System Status Agent` by means of the `Task Scheduler API` ① (interactions ③ and ⑥). Then, the `System Status Agent` component requests the status of the federation network (Links that connect the nodes of the federation) to the `Network Status Reporter` component of each `Networking Node` (see Appendix C, Figure 12). Moreover, the `System Status Agent` also requests to the `Task Processor Status Reporter` component (see Appendix D, Figure 13) the status related to the hardware usage of each `Task Processor` (i.e. `Task Nodes` and other fog/cloud nodes) that belong to the same federation that the

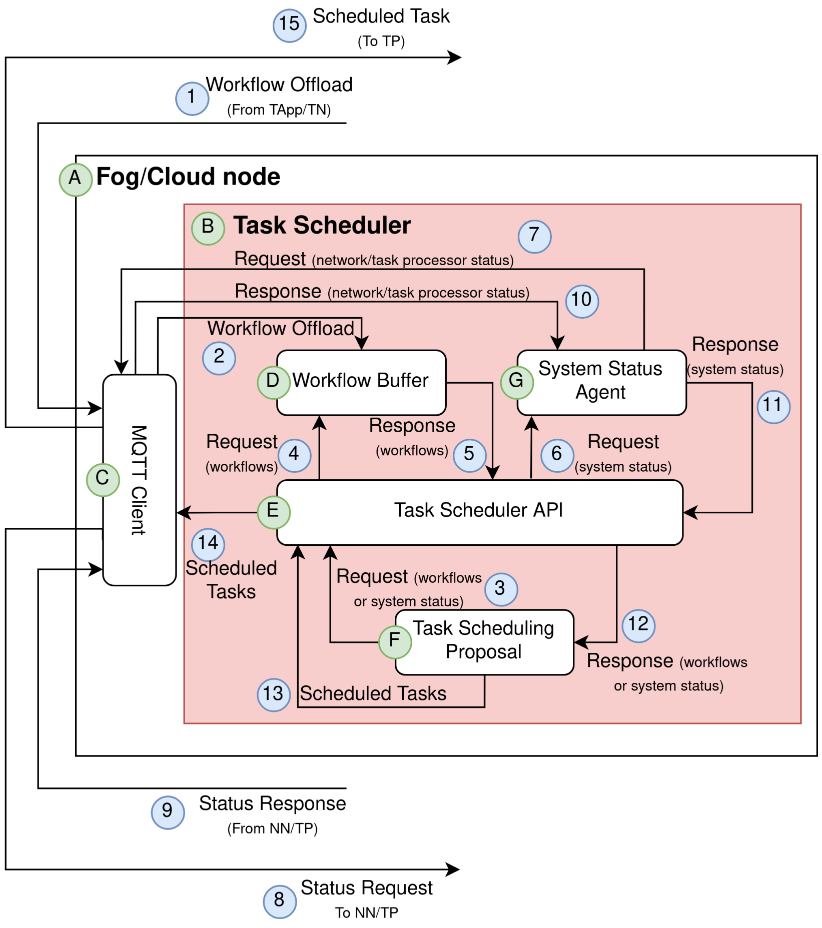


Figure 8 Task Scheduler components.

Task Scheduler. These two aforementioned requests are represented by the interactions ⑦ and ⑧. When the status data is received ⑨, the System Status Agent provides this data to the Task Scheduling Proposal ⑪ and ⑫. Thus, the Task Scheduling Proposal can use this data as input for task scheduling purposes.

Task Scheduler API Ⓔ The Task Scheduling Proposal is the task scheduling approach that the user can deploy into the simulation for testing and analysis purposes. The Task Scheduler API has been conceived as a middleware service to integrate the Task Scheduling Proposal with the simulated system. Thus, the Task Scheduler API is able to 1) gather key data about the status of the simulated system and provide it to the Task Scheduling Proposal, 2) provide the Task Scheduling Proposal with the workflows that have been offloaded to the Task Scheduler Ⓑ and 3) return the tasks of a workflow scheduled to the system for their processing. Note that the request/response interactions related to 1) and 2) (③, ④, ⑤, ⑥, ⑪, ⑫) have been already addressed through the explanation of the Workflow Buffer Ⓓ and the System Status Agent Ⓖ. Hence, with the offloaded workflows and the status of each node and link of the federation, the Task Scheduling Proposal can perform the schedule of each task. Once the Task Scheduling proposal has finished the schedule, returns the tasks scheduled to the system for their processing, also by means of the Task Scheduler API (interaction ⑬, ⑭ and ⑮).

Task Scheduling Proposal Ⓔ The Task Scheduling Proposal implements the task scheduling approach that the user can deploy into the simulation for testing and analysis purposes. The aim of the Task Scheduling Proposal is to schedule the offloaded workflows. The `Task Scheduling Proposal` only interacts with the `Task Scheduler API` Ⓔ. This is because the `Task Scheduling Proposals` is developed by the user, and will not necessarily have been developed to be tested in the simulator proposed in this communication. Therefore, to facilitate their integration with the simulator, the `Task Scheduler API` Ⓔ is used as an interoperability layer. By means of a series of requests, the `Task Scheduling Proposal` can interact with the simulated system receiving the offloaded workflows, gathering data related to the status of the system to carry out the schedule of each task and return them scheduled to the system for their processing. Note that all these interactions have been described above.

5.5. Deployment Infrastructure

The deployment of simulations is carried out through container orchestration. Thus, the system components are wrapped in Docker (Merkel 2014) containers as shown in Figure 9, and subsequently deployed in different clusters (orchestration). Regarding the wrapping of the components, Figure 9 uses the Docker logo within the represented boxes to indicate that said component or set of components is wrapped in a Docker container.

On the other hand, the default supported orchestrator is Docker Swarm, although since the deployment is carried out using `Docker-Compose`, it is possible to perform the deployment on Kubernetes (*Kubernetes Documentation* 2023) by automatically generating the deployment files for this orchestrator using the Kompose tool (*Kompose* 2023).

As for communication between containers, it is carried out through DNS (name + id of the component, specified in the modelling stage). Communication through DNS is supported by the overlay network that is generated to support the deployment.

Finally, orchestration takes into account the fog and cloud nodes modelled as part of the system. In this way, a "cluster" is generated for each fog or cloud node modelled. Subsequently, all the Docker containers related to the fog or cloud node are deployed on this "cluster", as well as those components belonging to the mist or edge layer that interact with it (e.g. Task Nodes).

Note that all these specifications are low-level specifications that, although not previously mentioned, are part of the simulator, in this case of the files generated through M2T transformations to perform the simulation deployment.

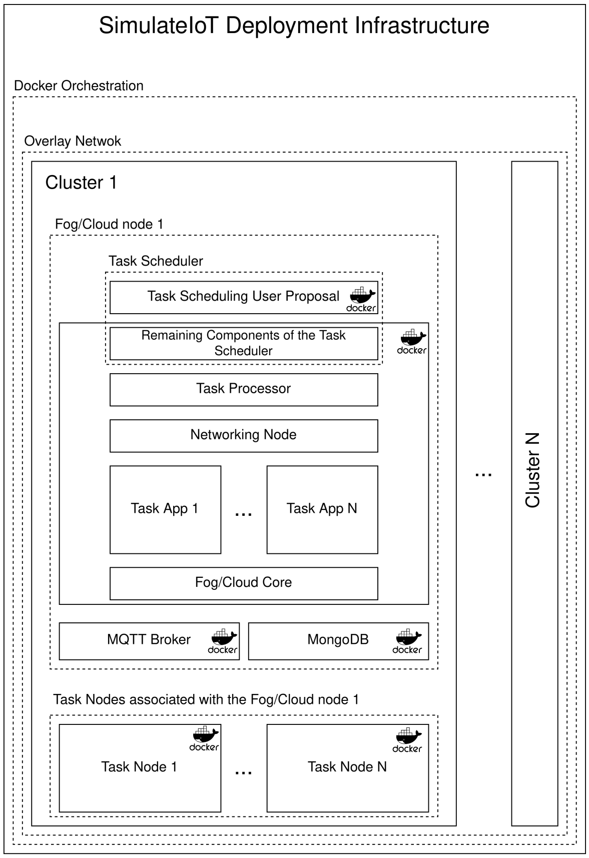


Figure 9 Deployment infrastructure of the extended version of SimulateIoT.

6. Simulation Outputs, Tests and Assessments

The primary motivation for the simulation and testing of an IoT system is to derive valuable insights, enabling the assessment of its behaviour or the optimisation of its performance. Consequently, the advantages that can be drawn from an IoT simulator rely on the variety and depth of tests and evaluations it supports. The SimulateIoT extension carried out in this communication facilitates several options for conducting tests and evaluations from a task scheduling perspective. The most significant tests and assessments that can be performed are listed below.

– Task distribution: How, in general, the task scheduling proposal distributes tasks among devices.
– Overload avoidance: The task scheduling proposal's ability to prevent any single device from becoming overloaded with tasks.
– Dynamic load balancing: How the task scheduling proposal can adjust task distribution as the load changes.
– Task prioritisation: If included in the policies of the task scheduling proposal, how it balances the load while considering task priorities.
– Response time load balancing: If included in the policies of the task scheduling proposal, how it can maintain uniform response times by effectively balancing the load.
– Queue length: The task scheduling proposal's ability to maintain balanced queue lengths across all devices.
– Task rebalancing: How effectively the task scheduling proposal can rebalance tasks when new devices join or existing devices leave the system.
– Network load variation: How the task scheduling proposal balances the load under varying network conditions.

- Workflow response time: The elapsed time from when a workflow is generated by a device, processed by the system, to the delivery of the processing results back to the device. This can also be applied in the context of tasks.
- Workflow processing rate: The number of workflows processed by a device over a specific period of time. This can also be applied in the context of tasks.
- Workflow processing throughput: The maximum number of workflows a device can process in a given time frame. This can also be applied in the context of tasks.
- Hardware consumption: How the system or a device uses its resources (CPU, memory, etc.) during the whole simulation or at a specific time.
- Resource allocation: Whether the resource allocation strategy followed during the design stage of the system was appropriate.
- Bottleneck identification: Finding points in the system where bottlenecks occur that could limit the system's performance.
- Workflow response time, processing rate and throughput scalability: Evolution of these parameters as the workload on the system is increased or reduced.
- Device scalability: How the system handles an increasing number of IoT devices to check if the system can maintain performance as the network grows.
- Network traffic scalability: How the system performs under different levels of network traffic.
- Fault tolerance: How the system behaves, in general, when faced with hardware or software faults. Note that this proposal does not include a specific model to induce failures during simulations. However, a simple script can stop (and redeploy later if required) the components of the simulated system (Docker containers). Thus, thanks to the components notifying the status of each node to the `Task Scheduler`, it can notice if a device is available or not (lacked response when stopped) and take it into account.
- Fault recovery: How the system handles recovery processes in the event of a failure, ensuring no task or data loss.
- Fault tolerance load balancing: The task scheduling proposal's ability to redistribute tasks when a device fails or becomes unavailable.
- Power consumption: The power consumed by the IoT system during a simulation or in a specific period of time. Note that a model related to the power consumption of devices has not been included in this work, however, it can be assessed from the hardware consumption.
- Workflow-specific energy consumption: The amount of energy consumed for each specific workflow.
- Device-specific energy consumption: The energy consumption of individual devices under different task loads.

As can be seen, the extension developed for SimulateIoT in this study provides users with a wide spectrum of testing and analysis opportunities for their task scheduling proposals. Thus, providing users with a holistic understanding of their IoT systems design and their task scheduling proposals performance.

7. Case Study: An IIoT System Applied to the Steel Industry for Predictive Maintenance

In this section, a study case focused on the IIoT applied to the steel industry for predictive maintenance is illustrated.

7.1. Motivation

Task scheduling techniques can be applied to any IoT system to optimise the processing of their tasks (Potluri & Rao 2020). However, their application is particularly appealing in the so-called critical IoT systems, i.e. IoT systems on which the safety of users depends and IoT systems on which specific response times, fault tolerance, etc. have to be met in order to perform suitably. Some of these critical IoT systems could be those focused on healthcare, traffic safety and control (IoV) or industry (IIoT) (Andersson et al. 2016).

In industry, optimal maintenance of production equipment and facilities is one of the keys to global competitiveness and survival (Zhao et al. 2022). Over time, different maintenance strategies, such as corrective and preventive maintenance, have been developed and applied (Lie & Chun 1986; Hao et al. 2010). Nowadays, with the possibility of continuous monitoring of equipment and facilities provided by the IoT and machine learning, a new maintenance strategy is being developed and implemented, predictive maintenance (Çınar et al. 2020; Cheng et al. 2020). This type of maintenance is based on predicting equipment failures or breakdowns, allowing maintenance work to be carried out before the equipment suffers further damage and causes a more negative impact on production (Carvalho et al. 2019).

In this context, electric motors are one of the most widely used tools in industry (Cakir et al. 2021). Their applications are varied, primarily including blowers, turbines, pumps, compressors, alternators, rolling mills, movers, etc. Thus, for the reasons outlined above, proper maintenance of these engines is crucial for companies to be competitive.

Given the need for predictive maintenance of electric motors, as well as the suitability of the application of IoT and task scheduling to achieve this purpose, it has been considered appealing to show the application of the proposed simulator in this context. Thus, this case study illustrates how the proposed simulator can assist in the design, development and implementation of an IoT system for monitoring and predicting the failure of electric motors in a steel company.

7.2. Overview

The use case presented in this Section is based on those works referenced in the above Section 7.1. The aim of the modelled system (using the metamodel presented in this communication) shown in Figure 10 is to provide a steel company with the capability to perform predictive maintenance on their electric engines. On the other hand, the aim of the use case is to test whether the system identifies and stops faulty engines below a time threshold. For example, if at any point during the entire simulation, the system takes no more than 10 seconds (hypothetical time threshold) from the moment an engine starts malfunctioning until the system identifies this situation and stops the engine. Note that these time thresholds can not be specified as part of the system as it is not supported by the DSL proposed. Instead, the user is who has to model the system and later, analyse the logs of the simulation to check the behaviour of the system. Continuing with the proposed example, check whether the time threshold is met during the entire simulation.

For this purpose, a set of sensors has been included in the edge layer of the system (red-coloured components) in order to continuously monitor each electric engine. Two `Task Nodes` have also been added to the edge layer, a gateway, which carries out the aggregation of the publishing data for each sensor (note that this is a task included in the simulation by means of a workflow), and a computer, which is used only to provide support for the processing of the tasks to be carried out in the system. In addition, an actuator has also been included in the edge layer to stop the operation of those engines whose failure has been predicted.

As for the fog layer, the modelled system includes several fog nodes that provide the different deployed edge nodes with topics for subscribing or publishing their data. Besides, these fog nodes also perform two tasks in the system, the pre-processing of the received data (aggregated by the gateway) and the failure prediction of each monitored engine (from this pre-processed data). Note that these two tasks have been also included in the simulation by means of workflows.

Furthermore, a cloud node has been added to the system. This cloud node aims to provide additional hardware resources to the system if needed.

Finally, note that the simulation related to this use case has been deployed on a personal computer with the following specifications: Model: MSI GP63 Leopard 8RE; CPU: Intel Core i7-8750H; Graphics: GeForce GTX 1060 with 6GB GDDR5; RAM Memory: 16GB DDR4-2400.

7.3. Model Definition

Figure 10 shows an excerpt of the IIoT system model. For the purpose of explaining this model, it includes numerical references for each node, which are referenced below when describing each component of the case study. Besides, for the sake of clarity, note that the description of the case study is divided into three parts: 1) edge layer (red nodes), 2) fog layer (blue nodes), and 3) cloud layer (green nodes).

7.3.1. Edge Layer The edge layer of the IIoT system modelled for this case study is comprised of three kinds of devices: sensors (S.X), actuators (A.X) and `Task Nodes` (T.X). The sensors included are accelerometers (S.1, S.4 and S.7), thermometers (S.2, S.5 and S.8) and (magnetic) Hall sensors (S.3, S.6 and S.9). The accelerometers gather vibration data, i.e. the vibration that the bearings of the engine produce, the thermometers gather the temperature of the engine, and 3) the (magnetic) Hall sensors collect data related to the rotational speed of the engines. Note that these sensors and the data they collect are often used to predict failures in electric engines (Cakir et al. 2021).

These sensors collect this data and publish it on the topics To.X provided by the fog nodes F.X for further use. Accelerometers publish their data in the topic *x/.../engines/vibration*, thermometers in the topic *x/.../engines/temperature* and (magnetic) Halls sensors in the topic *x/.../engines/rotationspeed*. Note that the data gathered and published by these sensors has been modelled by the user with the expressiveness already provided by the previous version of SimulateIoT.

The actuators included A.X aim to stop the operation of those engines whose failure has been predicted. Thus, they subscribe to the topic *x/.../engines/prediction*, where the nodes that carry out the prediction (fog F.X and Cloud C.X nodes) of the failure of the engines publish their predictions. In this way, when an engine failure prediction is published in this topic, the actuators receive it and stop the operation of the engine.

Finally, two `Task Nodes` Tn.X have been modelled. On the one hand, a gateway Tn.1, which has a `Hardware_specification` H.5 where the `CPU` and `RAM` of the device have been modelled. This device has been modelled to take advantage of its hardware for task scheduling purposes. Moreover, this `Task Node` performs a task, the data aggregation T.7 of the data published by the modelled sensors. In this way, when the data reach the topics and the fog nodes, it is already aggregated. Note that this task has been modelled by means of a `Workflow` in the *properties* of this component. On the other hand, a personal computer has been added as a `Task Node` of the system, being part of the edge layer and providing the rest of the system with more computing power. This `Task Node` have also a `Hardware_specification` H.6.

Note that each sensor, actuator and `Task Node` have an attribute named `quantity` (included in the previous version of SimulateIoT) where the user can specify the quantity of each node. In this case study, the `quantity` attribute is ten for sensors, three for actuators, three for the gateway `Task Node` and one for the computer `Task Node`.

7.3.2. Fog Layer The fog layer of the IIoT system modelled for this case study is comprised of three fog nodes, `FogNode_rolling_mill` F.1, `FogNode_power_plant_ventilation_system` F.2 and `FogNode_lathes` F.3. The `FogNode_rolling_mill` is the fog node deployed near the rolling mill, the `FogNode_power_plant_ventilation_system` is the fog node deployed near the ventilation system of the power plant of the company, and the `FogNode_lathes` represents the fog node deployed near the lathes of the company. The aim of these fog nodes is to provide services to the rest of the system.

In this regard, the `FogNode_rolling_mill` provides topics To.1 to subscribe or publish data to those sensors installed on the engines of the `FogNode_rolling_mill` of the company. Besides, this node carries out two tasks, 1) the data pre-processing T.1 of the received data (already aggregated by the gateway `Task Node`) to send this data in a suitable way to the machine learning model of the system, which use it as input to make predictions, and 2) The fault prediction T.2 of the engines of the

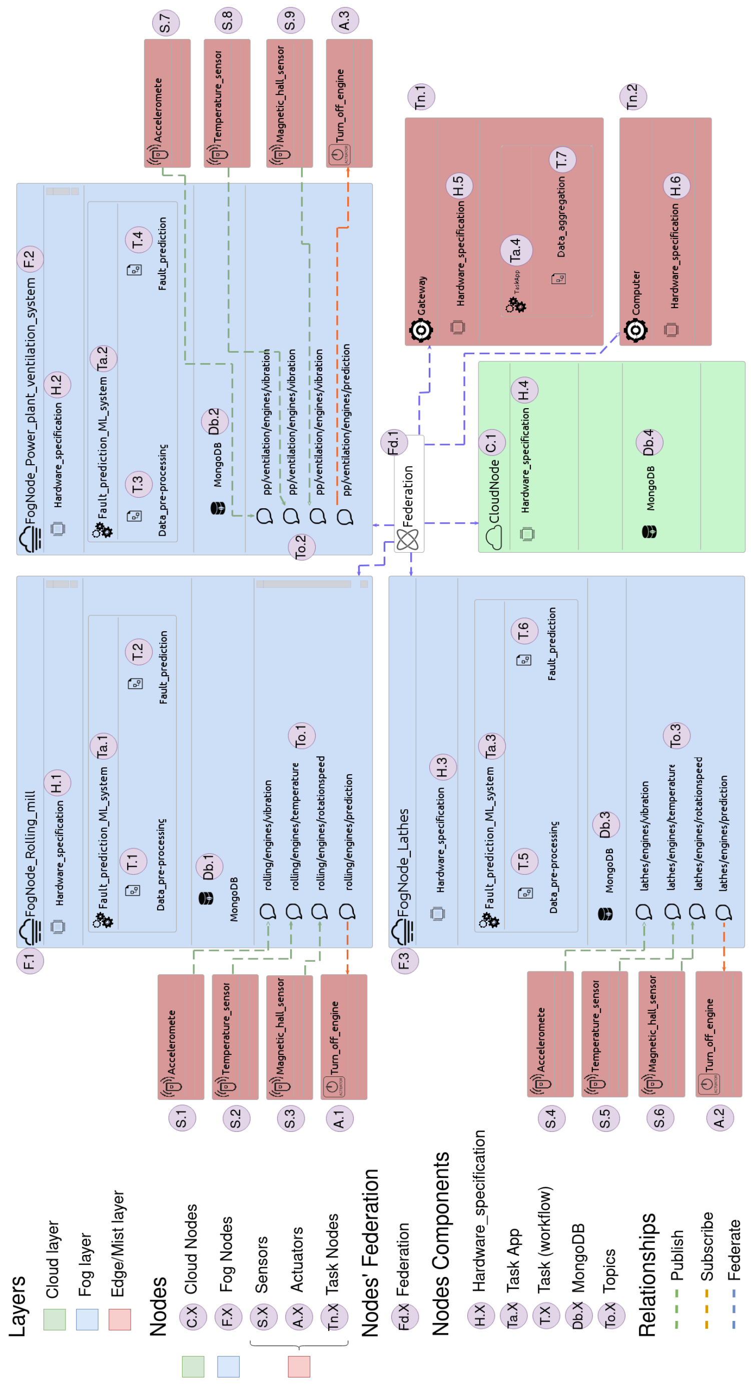


Figure 10 Model conforms to the proposed simulator metamodel. An IIoT system applied to the steel industry for predictive maintenance of electric engines.

rolling mill. These predictions are later published in the topic *x/.../engines/prediction* where the actuators responsible to stop the engines are subscribed. Note that these tasks have been modelled by means of `Workflows` in the *properties* of these components. This fog node has also a `hardware_specification` (H.1) and a mongo database (Db.1).

The other two fog nodes, the `FogNode_power_plant_ventilation_system` and the `FogNode_lathes` have a similar configuration to the `FogNode_rolling_mill`. The differences are 1) The `Workflows` modelled in the tasks defined in each of them are different 2) Each fog node represents the implementation of the fault prediction system in the electric engines of the different equipment and facilities of the company.

7.3.3. Cloud Layer

The cloud layer has been included with one aim, to provide hardware resources to the rest of the system. Thus, one cloud node has been modelled (C.1). In this regard, this cloud node has a `hardware_specification` greater than the `hardware_specification` of the fog nodes, and a Mongo database (Db.4).

7.3.4. Cloud-Fog-Edge Federation

Although the cloud-fog-edge `Federation` (Fd.1) is not a layer, it allows each cloud, fog and edge node modelled to operate as a single entity instead of isolated nodes. This component has been modelled federating each fog, cloud and `Task Node` modelled. Thus, all the nodes with computing power (a `hardware_specification`) can cooperate for task scheduling purposes. Note that in the properties of this component, the features of each `Link` that interconnects the nodes belonging to the `Federation` (`bandwidth` and `delay`) have been modelled.

Finally, note that although in this case study has not been modelled, additional nodes could be defined to build other federations.

7.4. M2T Transformations

Once the model has been defined, the M2T transformations are applied with the following goals:

- i) to generate Java, Python, NodeJs, etc. code that wraps each device behaviour.
- ii) to generate configuration code to deploy all the generated services, such as the message brokers necessary, including the *topic* configurations defined, the gateway configurations, etc.
- iii) to generate the code and deployment configuration files of the architecture that supports task scheduling (`Task Apps`, `Task Nodes`, `Networking Nodes`, `Task Processors` and the `Task Schedulers`).
- iv) to generate the code and deployment configuration of the users' task scheduling proposal and their integration with the simulation.
- v) 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.
- vi) to generate for each cloud, fog and edge node a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm*.

Consequently, each edge node, fog node and cloud node and their related components are generated following the software architecture defined in Section 5 where the M2T transformations have been defined.

Finally, note that to generate a part of the code in a target language/infrastructure different from the one supported, users will need to make the following efforts: 1) Understand the metamodel or the concepts related to the component/s they wish to update; 2) Understand the M2T transformations related to the component/s to be modified; 3) Develop the code from scratch in their target language; 4) Integrate it into Acceleo; 5) Conduct sufficient tests and trials to confirm the successful update of the component/s.

7.5. Simulation Analysis

The benefits that can be obtained from a simulator come from the outputs, data, etc. from the simulations and tests performed. In this section, a set of simulations and different tests based on the model described above are carried out, illustrating the possibilities and benefits provided by the proposed simulator. To carry out these simulations and tests, the HEFT algorithm (Topcuoglu et al. 2002), one of the most widely used and extended algorithms in the field of task scheduling, on which some recent algorithms are based (Ojha et al. 2020; Divyaprabha et al. 2018; Faragardi et al. 2020), has been integrated into the simulator. Consequently, the `Task Scheduler` applies the aforementioned task scheduling algorithm (HEFT) to process the workflows during simulations. Note that the M2T transformations only need the task scheduling proposal (in this case the HEFT algorithm) to be on the same path as the generated components (by the M2T transformations) in order to automatically integrate it into the simulation.

As for the test, first is desired to determine the average and maximum time that the modelled IIoT system needs to predict the failure of an engine. This involves the time it requires to aggregate and pre-process the data related to each engine and the time it requires to carry out the prediction (failure or not).

For this purpose, the simulation was run for 120 seconds. Once completed, the average time reported to predict the failure of an engine (any engine) is 4.391 seconds, and the maximum time is 6.384 seconds. The maximum time was related to the engines of the rolling mill. This was expected since the workflows related to failure prediction for this kind of engine are more complex (higher amount of bytes to process and to offload) than for the others.

At this point, the user has to determine whether this response time satisfies his performance needs i.e., if the response time is lower than expected, the user could reduce the hardware of the designed system, thus saving costs. If, on the other hand, the response time exceeds the estimated time to avoid severe engine damage, the system has to be upgraded, either by software or hardware.

In this use case, it has been determined to carry out a software upgrade. The HEFT algorithm includes an insertion policy, i.e., the idle slots of each processor (time in which the processor is not used between processing each task) can be used to process tasks. However, in the previous test, a modified HEFT algorithm was used in which the use of idle slots had been restricted. The simulation is then re-deployed with the HEFT algorithm and its insertion policy enabled.

For this purpose, again, the simulation was run for 120 seconds. Once completed, the average time reported to predict the failure of an engine (any engine) is 3.637 seconds, and the maximum time 6.314 seconds. Although the average execution time has improved (4.391 to 3.637), the maximum execution time has remained the same. This is because, due to the number of nodes in the federation, their hardware configuration and the workflows related to the tasks to be processed in the system, at a specific time, the insertion policies cannot be applied as there are no idle slots available.

Thus, in order to reduce the maximum processing time for the failure prediction of the engines of the rolling mill, this software improvement is not enough, so it is determined to double the computational capacity of the fog nodes. After this improvement, the maximum processing time has been reduced to 3.259 seconds, about half.

Analysing why the processing of worst-case tasks is reduced a half, the simulator logs reported that the worst case (maximum execution time) occurs at an instant when the tasks (related to the worse case) are processed by fog nodes. Specifically for the fog node deployed near the rolling mills. Thus, by doubling its computing power, the processing of these tasks is reduced by about half.

These are some of the tests that the proposed simulator allows to carry out. These tests allow users to analyse the performance of their IoT systems and re-design them until reaching an optimal configuration that satisfies their performance requirements In this case study, users could have tested the impact of other adjustments, such as the modification of the system architecture (adding or subtracting nodes), modifying the workflows of each task, the features of the links that inter-connects each node to the federation, etc.

8. Conclusions and Future Work

Model-driven development (MDD) offers an effective solution for dealing with the technological complexity of domains where diverse technologies are used. The key of MDD lies in its emphasis on the creation of abstractions of the application domain using the four-layer metamodel architecture. This architecture facilitates a structured approach towards system design.

Once these models are established, we can proceed to the model-to-text (M2T) transformation stage, where the developed models are transformed into executable code specific to the technology in use. This approach effectively mitigates the risks and challenges of manual coding, enhancing productivity and reducing the margin for error.

This paper proposes an extension of the SimulateIoT domain-specific language (DSL) towards IoT simulation in the context of task scheduling and the cloud-to-thing continuum paradigm. This DSL extension aids users in conceptually framing their task scheduling proposals within their IoT system designs based on the cloud-to-thing continuum paradigm. By using this language, users can propose, simulate, and evaluate various IoT system designs and task scheduling solutions, thereby working towards a system that fulfils their specific requirements, such as Quality of Service (QoS) or Service Level Agreements (SLAs).

The system design's components can include a variety of elements including cloud, fog, edge, and mist nodes. These nodes can be federated to create an integrated system. Additionally, devices and applications that generate and offload workflow-based tasks can be incorporated, along with the necessary architecture for processing these tasks. This broad range of possibilities allows users to model realistic IoT systems where task scheduling plays a pivotal role.

Finally, once the IoT system is modelled and simulated, the simulation's outputs can be gathered, analysed, and leveraged for the purpose of system refinement and optimisation. This iterative process of design, simulation, deployment, and analysis serves as a feedback loop, enabling continuous improvement of the system in line with the user's evolving needs and technological advancements. This process, founded on the principles of the MDD, ensures a systematic, rigorous approach to designing and test task scheduling proposals and complex IoT systems based on the cloud-to-thing continuum paradigm.

Regarding the limitations of our extension, there are several points to consider. While we offer valuable logs pertaining to the energy consumption of the simulated IoT system, the extension does not encompass a comprehensive model that directly indicates the system's energy usage. Given the emphasis on energy efficiency in the current climate change scenario and the rising trend of energy-awareness task scheduling algorithms (green IoT) (Ghafari et al. 2022), this absence is a notable limitation of our tool.

Additionally, although our extension provides logs concerning the hardware consumption of each simulated component, potentially allowing for cost inference for deploying these components on private clouds like AWS or Google Cloud, it does not furnish a full-fledged pricing model. As many contemporary task scheduling algorithms consider the costs associated with IoT system deployments (Shu et al. 2021; Yuan et al. 2020), this omission is another significant limitation.

Moreover, since SimulateIoT emulates the infrastructure of modelled IoT systems rather than merely simulating it, the hardware demands for running a simulation are higher than for simulators reliant solely on mathematical models. This increases the hardware requirements to run simulations, which can pose challenges for scalability, especially when simulating large IoT systems.

Lastly, while SimulateIoT is designed to simplify the process of integrating and testing user task scheduling proposals, manual effort is still required to carry out this integration.

As for future work, there are several extensions that could be interesting to develop:

– Mobility: The proposed simulator does not support device mobility. Currently, some works in the literature focus on the study of the federation of an edge layer composed of mobile devices, the mobile edge computing paradigm (Mao et al. 2017; Maray & Shuja 2022). In this computing paradigm, mobile nodes belonging to the edge layer can leave or join the federation. This dynamic property of the edge layer requires an architecture to support it and the corresponding software to handle it. Thus, the inclusion of this paradigm in the simulator could be an advantage for work that focuses on the development of task scheduling techniques for this kind of system (Ma et al. 2021; Wang et al. 2021).

- Energy consumption: Currently, many task scheduling proposals focus on the sustainable development of the IoT, thus prioritising energy consumption optimisation over the makespan of the tasks to be processed (Ghafari et al. 2022). Introducing the concept of devices' batteries or system energy consumption to the proposed simulator could help those users who require test their task scheduling proposals for energy optimisation.
- Cost of use: Given that several cloud platforms offer their services for a certain price and also that energy has a monetary cost, in literature there are several task scheduling proposals focused on optimising the use of system resources (Shu et al. 2021; Yuan et al. 2020). Integrating the concept of resources cost or the model of *pay-as-you-use* could be interesting to allow these users to test their proposals.
- A textual concrete syntax will be developed in order to facilitate the modelling of this kind of system by using a textual notation.
- Taking into account that currently users have to manage the QoS and SLAs manually, it could be interesting to consider an extension of the proposed metamodel to define QoS and SLAs. Thus, facilitating users the modelling and handling of such concepts for each model.

Acknowledgments

This work was funded by project TED2021-129194B-I00 funded by MCIN/ AEI/ 10.13039/501100011033 and for European Union NextGenerationEU/ PRTR; the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the projects IB20058, and by the European Regional Development Fund (ERDF).

Proyecto TED2021-129194B-I00 financiado por:

Consejería de Economía, Ciencia y Agenda Digital

References

Aazam, M., Zeadally, S., & Harras, K. A. (2018). Deploying fog computing in industrial internet of things and industry 4.0. *IEEE Transactions on Industrial Informatics*, *14*(10), 4674-4682. doi: 10.1109/TII.2018.2855198

Ahmad, Z., Jehangiri, A. I., Ala'anzy, M. A., Othman, M., Latip, R., Zaman, S. K. U., & Umar, A. I. (2021). Scientific workflows management and scheduling in cloud computing: taxonomy, prospects, and challenges. *IEEE Access*, *9*, 53491–53508.

Alizadeh, M. R., Khajehvand, V., Rahmani, A. M., & Akbari, E. (2020). Task scheduling approaches in fog computing: A systematic review. *International Journal of Communication Systems*, *33*(16), e4583.

Al-Maytami, B. A., Fan, P., Hussain, A., Baker, T., & Liatsis, P. (2019). A task scheduling algorithm with improved makespan based on prediction of tasks computation time algorithm for cloud computing. *IEEE Access*, *7*, 160916–160926.

Andersson, M. A., Özçelikkale, A., Johansson, M., Engström, U., Vorobiev, A., & Stake, J. (2016). Feasibility of ambient rf energy harvesting for self-sustainable m2m communications using transparent and flexible graphene antennas. *IEEE Access*, *4*, 5850-5857. doi: 10.1109/ACCESS.2016.2604078

Arunarani, A., Manjula, D., & Sugumaran, V. (2019a). Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, *91*, 407-415. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167739X17321519 doi: https://doi.org/10.1016/j.future.2018.09.014

Arunarani, A., Manjula, D., & Sugumaran, V. (2019b). Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, *91*, 407–415.

Asghari, A., Sohrabi, M. K., & Yaghmaee, F. (2021). Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel sarsa reinforcement learning agents and genetic algorithm. *The Journal of Supercomputing*, *77*, 2800–2828.

Atkinson, C., & Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE software*, *20*(5), 36–41.

Bansal, S., Aggarwal, H., & Aggarwal, M. (2022). A systematic review of task scheduling approaches in fog computing. *Transactions on Emerging Telecommunications Technologies*, e4523.

Barriga, J. A., Clemente, P. J., Pérez-Toledano, M. A., Jurado-Málaga, E., & Hernández, J. (2023). Design, code generation and simulation of iot environments with mobility devices by using model-driven development: Simulateiot-mobile. *Pervasive and Mobile Computing*, *89*, 101751. Retrieved from https://www.sciencedirect.com/science/article/pii/S1574119223000093 doi: https://doi.org/10.1016/j.pmcj.2023.101751

Barriga, J. A., Clemente, P. J., Sosa-Sánchez, E., & Prieto, A. E. (2021). Simulateiot: Domain specific language to design, code generation and execute iot simulation environments. *IEEE Access*, *9*, 92531-92552. doi: 10.1109/ACCESS.2021.3092528

Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., … Rana, O. (2018). The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things*, *3*, 134–155.

Cakir, M., Guvenc, M. A., & Mistikoglu, S. (2021). The experimental application of popular machine learning algorithms on predictive maintenance and the design of iiot based condition monitoring system. *Computers & Industrial Engineering*, *151*, 106948. Retrieved from https://www.sciencedirect.com/science/article/pii/S0360835220306252 doi: https://doi.org/10.1016/j.cie.2020.106948

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., & Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, *41*(1), 23–50.

Carvalho, T. P., Soares, F. A., Vita, R., Francisco, R. d. P., Basto, J. P., & Alcalá, S. G. (2019). A systematic literature review of machine learning methods applied to predictive maintenance. *Computers & Industrial Engineering*, *137*, 106024.

Chen, W., & Deelman, E. (2012). Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *2012 ieee 8th international conference on e-science* (p. 1-8). doi: 10.1109/eScience.2012.6404430

Cheng, J. C., Chen, W., Chen, K., & Wang, Q. (2020). Data-driven predictive maintenance planning framework for mep components based on bim and iot using machine learning algorithms. *Automation in Construction*, *112*, 103087.

Chernyshev, M., Baig, Z., Bello, O., & Zeadally, S. (2018). Internet of things (iot): Research, simulators, and testbeds. *IEEE Internet of Things Journal*, *5*(3), 1637-1647. doi: 10.1109/JIOT.2017.2786639

Çınar, Z. M., Abdussalam Nuhu, A., Zeeshan, Q., Korhan, O., Asmael, M., & Safaei, B. (2020). Machine learning in predictive maintenance towards sustainable smart manufacturing in industry 4.0. *Sustainability*, *12*(19), 8211.

Ding, D., Fan, X., Zhao, Y., Kang, K., Yin, Q., & Zeng, J. (2020). Q-learning based dynamic task scheduling for energy-efficient cloud computing. *Future Generation Computer Systems*, *108*, 361–371.

Divyaprabha, M., Priyadharshni, V., & Kalpana, V. (2018). Modified heft algorithm for workflow scheduling in cloud computing environment. In *2018 second international conference on inventive communication and computational technologies (icicct)* (p. 812-815). doi: 10.1109/ICICCT.2018.8473237

Erazo, M. A., & Liu, J. (2013). Leveraging symbiotic relationship between simulation and emulation for scalable network experimentation. In *Proceedings of the 1st acm sigsim conference on principles of advanced discrete simulation* (pp. 79–90).

Faragardi, H. R., Saleh Sedghpour, M. R., Fazliahmadi, S., Fahringer, T., & Rasouli, N. (2020). Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds. *IEEE Transactions on Parallel and Distributed Systems*, *31*(6), 1239-1254. doi: 10.1109/TPDS.2019.2961098

Fisher, A., Rudin, C., & Dominici, F. (2019). All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *J. Mach. Learn. Res.*, *20*(177), 1–81.

Gazori, P., Rahbari, D., & Nickray, M. (2020). Saving time and cost on the scheduling of fog-based iot applications using deep reinforcement learning approach. *Future Generation Computer Systems*, *110*, 1098–1115.

Ghafari, R., Kabutarkhani, F. H., & Mansouri, N. (2022). Task scheduling algorithms for energy optimization in cloud environment: a comprehensive review. *Cluster Computing*, *25*(2), 1035–1093.

Girs, S., Sentilles, S., Asadollah, S. A., Ashjaei, M., & Mubeen, S. (2020). A systematic literature study on definition and modeling of service-level agreements for cloud services in iot. *IEEE Access*, *8*, 134498–134513.

Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., & Razafindralambo, T. (2011). A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, *49*(11), 58-67. doi: 10.1109/MCOM.2011.6069710

Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K., & Buyya, R. (2017). ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, *47*(9), 1275–1296.

Gupta, S., Iyer, S., Agarwal, G., Manoharan, P., Algarni, A. D., Aldehim, G., & Raahemifar, K. (2022). Efficient prioritization and processor selection schemes for heft algorithm: A makespan optimizer for task scheduling in cloud environment. *Electronics*, *11*(16), 2557.

H., S., & V., N. (2021). A review on fog computing: Architecture, fog with iot, algorithms and research challenges. *ICT Express*, *7*(2), 162-176. Retrieved from https://www.sciencedirect.com/science/article/pii/S2405959521000606 doi: https://doi.org/10.1016/j.icte.2021.05.004

Hao, Q., Xue, Y., Shen, W., Jones, B., & Zhu, J. (2010). A decision support system for integrating corrective maintenance, preventive maintenance, and condition-based maintenance. In *Construction research congress 2010: Innovation for reshaping construction practice* (pp. 470–479).

Hosseinioun, P., Kheirabadi, M., Kamel Tabbakh, S. R., & Ghaemi, R. (2022). atask scheduling approaches in fog computing: A survey. *Transactions on Emerging Telecommunications Technologies*, *33*(3), e3792. Retrieved from https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3792 (e3792 ETT-19-0285.R1) doi: https://doi.org/10.1002/ett.3792

Jamil, B., Ijaz, H., Shojafar, M., Munir, K., & Buyya, R. (2022). Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions. *ACM Computing Surveys (CSUR)*.

Kar, B., Yahya, W., Lin, Y.-D., & Ali, A. (2022). A survey on offloading in federated cloud-edge-fog systems with traditional optimization and machine learning. *arXiv preprint arXiv:2202.10628*.

Kolovos, D. S., García-Domínguez, A., Rose, L. M., & Paige, R. F. (2015). Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 1–27.

Kompose. (2023). https://kompose.io/. ([Online; accessed 09-Oct-2023])

Kubernetes Documentation. (2023). https://kubernetes.io/docs/home/. ([Online; accessed 09-Oct-2023])

Lera, I., Guerrero, C., & Juiz, C. (2019). Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, *7*, 91745-91758. doi: 10.1109/ACCESS.2019.2927895

Lie, C. H., & Chun, Y. H. (1986). An algorithm for preventive maintenance policy. *IEEE Transactions on Reliability*, *35*(1), 71-75. doi: 10.1109/TR.1986.4335352

Ma, X., Zhou, A., Zhang, S., Li, Q., Liu, A. X., & Wang, S. (2021). Dynamic task scheduling in cloud-assisted mobile edge computing. *IEEE Transactions on Mobile Computing*.

Mao, Y., You, C., Zhang, J., Huang, K., & Letaief, K. B. (2017). A survey on mobile edge computing: The communication perspective. *IEEE communications surveys & tutorials*, *19*(4), 2322–2358.

Maray, M., & Shuja, J. (2022). Computation offloading in mobile cloud computing and mobile edge computing: survey, taxonomy, and open issues. *Mobile Information Systems*, *2022*.

McGregor, I. (2002). The relationship between simulation and emulation. In *Proceedings of the winter simulation conference* (Vol. 2, p. 1683-1688 vol.2). doi: 10.1109/WSC.2002.1166451

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

Mijuskovic, A., Chiumento, A., Bemthuis, R., Aldea, A., & Havinga, P. (2021). Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification. *Sensors*, *21*(5). Retrieved from https://www.mdpi.com/1424-8220/21/5/1832 doi: 10.3390/s21051832

Mishra, S., Mishra, S., Kayal, A., & Chudi, S. R. (2012). Simulation in wireless sensor networks. *International Journal of Electronics Communication and Computer Technology (IJECCT)*, *2*(4), 176.

NoorianTalouki, R., Shirvani, M. H., & Motameni, H. (2022). A heuristic-based task scheduling algorithm for scientific workflows in heterogeneous cloud computing platforms. *Journal of King Saud University-Computer and Information Sciences*, *34*(8), 4902–4913.

Obeo. (2012). *Acceleo project http://www.acceleo.org.*

Ojha, S. K., Rai, H., & Nazarov, A. (2020). Enhanced modified heft algorithm for task scheduling in cloud environment. In *2020 2nd international conference on advances in computing, communication control and networking (icacccn)* (p. 866-870). doi: 10.1109/ICACCCN51052.2020.9362975

OMG. (2012, January). *OMG Object Constraint Language (OCL), Version 2.3.1.* Retrieved from http://www.omg.org/spec/OCL/2.3.1/

Pan, J., & McElhannon, J. (2018). Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, *5*(1), 439-449. doi: 10.1109/JIOT.2017.2767608

Potluri, S., & Rao, K. S. (2020). Optimization model for qos based task scheduling in cloud computing environment. *Indonesian Journal of Electrical Engineering and Computer Science*, *18*(2), 1081–1088.

Qian, L., Luo, Z., Du, Y., & Guo, L. (2009). Cloud computing: An overview. In *Cloud computing: First international conference, cloudcom 2009, beijing, china, december 1-4, 2009. proceedings 1* (pp. 626–631).

Rashid, A., & Chaturvedi, A. (2019). Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, *7*(2), 421–426.

Rodrigo, G. P., Elmroth, E., Östberg, P.-O., & Ramakrishnan, L. (2018). Scsf: A scheduling simulation framework. In D. Klusáček, W. Cirne, & N. Desai (Eds.), *Job scheduling strategies for parallel processing* (pp. 152–173). Cham: Springer International Publishing.

Saltelli, A., & Funtowicz, S. (2014). When all models are wrong. *Issues in Science and Technology*, *30*(2), 79–85.

Samann, F. E. F., Zeebaree, S. R., & Askar, S. (2021). Iot provisioning qos based on cloud and fog computing. *Journal of Applied Science and Technology Trends*, *2*(01), 29–40.

Sandhu, M. M., Khalifa, S., Jurdak, R., & Portmann, M. (2021). Task scheduling for energy-harvesting-based iot: A survey and critical analysis. *IEEE Internet of Things Journal*, *8*(18), 13825–13848.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, *20*(5), 19–25.

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE software*, *20*(5), 42–45.

Shu, W., Cai, K., & Xiong, N. N. (2021). Research on strong agile response task scheduling optimization enhancement with optimal resource usage in green cloud computing. *Future Generation Computer Systems*, *124*, 12–20.

Singh, P., Dutta, M., & Aggarwal, N. (2017, Jul 01). A review of task scheduling based on meta-heuristics approach in cloud computing. *Knowledge and Information Systems*, *52*(1), 1-51. Retrieved from https://doi.org/10.1007/s10115-017-1044-2 doi: 10.1007/s10115-017-1044-2

Siow, E., Tiropanis, T., & Hall, W. (2018). Analytics for the internet of things: A survey. *ACM Computing Surveys (CSUR)*, *51*(4), 74.

Sterman, J. D. (2002). All models are wrong: reflections on becoming a systems scientist. *System Dynamics Review: The Journal of the System Dynamics Society*, *18*(4), 501–531.

Topcuoglu, H., Hariri, S., & Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, *13*(3), 260–274.

Wang, W., Lu, B., Li, Y., Wei, W., Li, J., Mumtaz, S., & Guizani, M. (2021). Task scheduling game optimization for mobile edge computing. In *Icc 2021-ieee international conference on communications* (pp. 1–6).

Wu, F., Wu, Q., & Tan, Y. (2015). Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, *71*(9), 3373–3418.

Yao, F., Pu, C., & Zhang, Z. (2021). Task duplication-based scheduling algorithm for budget-constrained workflows in cloud computing. *IEEE Access*, *9*, 37262–37272.

Yu, C., Lin, B., Guo, P., Zhang, W., Li, S., & He, R. (2018).

Deployment and dimensioning of fog computing-based internet of vehicle infrastructure for autonomous driving. *IEEE Internet of Things Journal*, *6*(1), 149–160.

Yuan, H., Liu, H., Bi, J., & Zhou, M. (2020). Revenue and energy cost-optimized biobjective task scheduling for green cloud data centers. *IEEE Transactions on Automation Science and Engineering*, *18*(2), 817–830.

Zhao, J., Gao, C., & Tang, T. (2022). A review of sustainable maintenance strategies for single component and multicomponent equipment. *Sustainability*, *14*(5). Retrieved from https://www.mdpi.com/2071-1050/14/5/2992 doi: 10.3390/su14052992

About the authors

José A. Barriga obtained his BSc degree in 2018 at the University of Extremadura and his MSc degree in 2019 at the International University of La Rioja. Currently, he is a PhD student at the University of Extremadura. He has been working for five years in the areas of IoT systems simulation, model-driven development applied to the IoT and machine learning applied to agriculture. You can contact the author at jose@unex.es.

José M. Cháves-González Jose M. Cháves-González is an Associate Professor of the Computer Science Department at the University of Extremadura (Spain). He received his BSc in Computer Science from the University of Extremadura in 2005 and a PhD in Computer Science in 2011. His research activity focuses on problem solving in the field of bioinformatics, the design and development of evolutionary and bio-inspired algorithms, multi-objective optimisation and the optimisation of algorithms using parallelism techniques. You can contact the author at jm@unex.es.

Arturo Barriga is a junior researcher in the Quercus Software Engineering Group at the University of Extremadura. He obtained his BSc degree from the University of Extremadura, Spain, 2022. Currently, he is a MSc student at the International University of La Rioja. His research focuses on the fields of digital twins and machine learning applied to agriculture. You can contact the author at arturobc@unex.es.

Pablo Alonso is a junior researcher in the Quercus Software Engineering Group at the University of Extremadura. He obtained his BSc degree in computer science from the University of Extremadura, Spain, in 2022. Currently, his research focuses on the fields of digital twins and simulation of Internet of Things (IoT) environments. You can contact the author at pabloap@unex.es.

Pedro J. Clemente is an Associate Professor of the Computer Science Department at the University of Extremadura (Spain). He received his BSc in Computer Science from the University of Extremadura in 1998 and a PhD in Computer Science in 2007. He has published numerous peer-reviewed papers in international journals, workshops, and conferences. His research interests include component-based software development, aspect orientation, service-oriented architectures, business process modelling, and model-driven development. He is involved in several research projects. He has participated in many workshops and conferences as speaker and member of the program committees. You can contact the author at pjclemente@unex.es
.

A. Appendix: The complete metamodel of the proposed simulator

This Section shows in Figure 11 the complete metamodel of the proposed simulator. This metamodel is composed of the SimulateIoT metamodel and the extension carried out (highlighted in blue). The description of the classes and relationships that are not part of the extension (and that have not been addressed in this article), can be found in the article (Barriga et al. 2021) Section IV, subsection A.

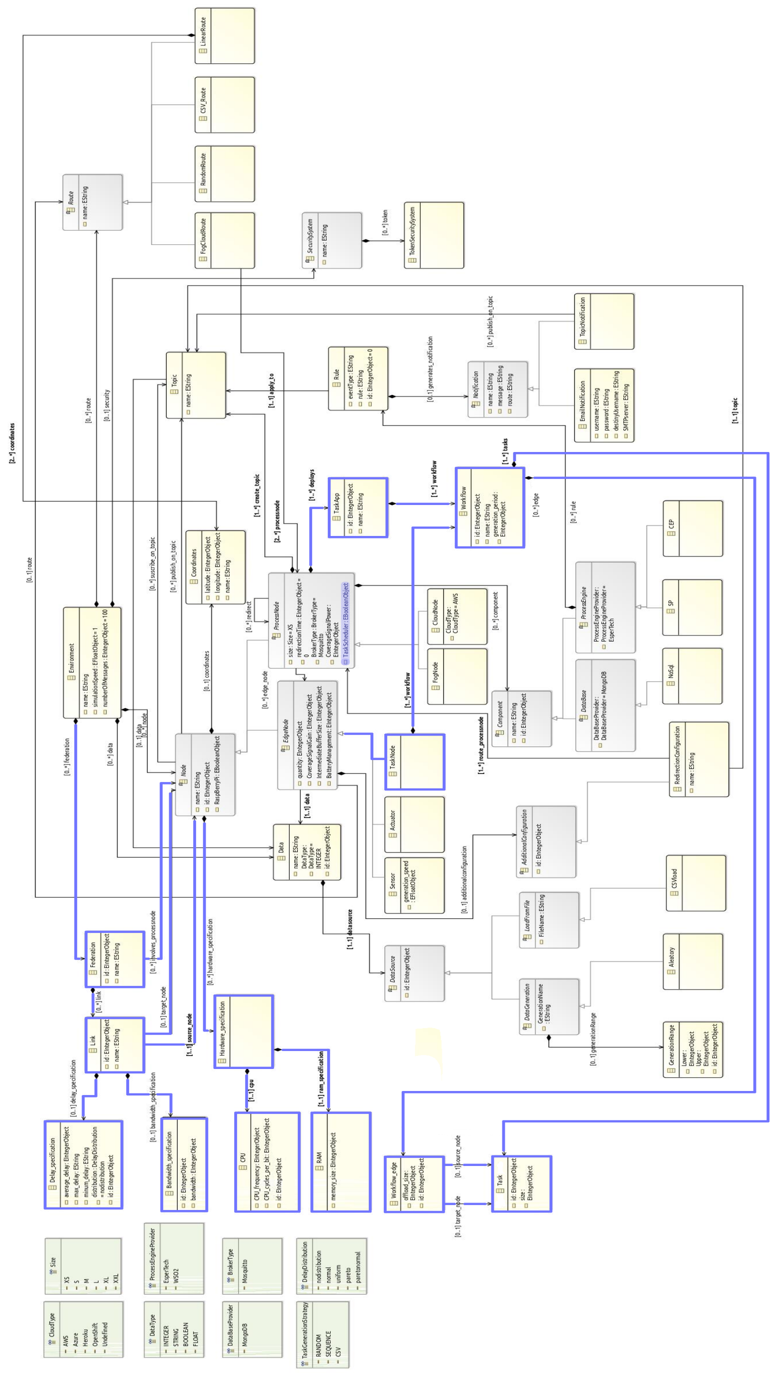


Figure 11 Complete SimulateIoT metamodel with the extension concepts and relationships included.

B. Appendix: Acceleo M2T example

```
[template public generateWorkflow(anEnvironment : Environment)]
[comment @main/]
[for (tnode : TaskNode | anEnvironment.node->filter(EdgeNode)->filter(TaskNode))]
  [for (workflow : Workflow | tnode.workflow)]
    [file ('/' + tnode.name + tnode.id + '/src/main/resources/wflows/wflow' + workflow.id + '.json', false, 'UTF-8')]

{
    "workflow": {
        "id": "[workflow.id/]",
        "name": "[workflow.name/]",
        "generatedBy": {
            "nodeId": "TaskNode[tnode.name + tnode.id/]",
            "generatorId": "[tnode.name + tnode.id/]",
            "generationId": "" // This ID acts as a counter within the component responsible for generating workflows,
 and is added during simulations.
        },
        "nodes":
            [for (task : Task | workflow.task) before('[') separator(', ') after(']')]
            {
                "task": {
                    "name": "[task.name/]",
                    "id": "[task.id/]",
                    "size": "[task.size/]"
                }
            }
      [/for]
    [if (workflow.edge->size()>0)]
    "edges":
    [/if]
    [for (edge : Edge | workflow.edge) before('[') separator(', ') after(']')]
            {
                "edge": {
                    "id": "[edge.id/]",
                    "sourceTaskId": "[edge.source.id/]",
                    "targetTaskId": {
            [for (target : Task | edge.target) separator(', ')]
            "target[i/]" : "[edge.id/]"
            [/for]
          },
                    "offloadSize": "[edge.offload_size/]"
                }
            }
        [/for]
    }
}
    [/file]
  [/for]
[/for]
[/template]
```

Listing 5 M2T developed in Acceleo for the generation of the workflows that the Task Nodes will offload to the system during the simulation.

C. Networking Node

Figure 12 shows a generic `Networking Node` Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the `Networking Node` are the `Delay Controller` Ⓒ, the `Synthetic Delay Generator` Ⓓ, the `Bandwidth Controller` Ⓔ and the `Network Status Reporter` Ⓖ. Besides, Figure 12 also shows how the `Networking Node` is deployed on an edge (`Task Node`), fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the edge/fog/cloud node and with the rest of the IoT system. Below, the `Networking Node` is illustrated by describing each of its components and their interactions.

`Delay Controller` Ⓒ The `Delay Controller` aims to apply the delay of the `Links` that connect the nodes belonging to a federation. The delay is applied from the source node to the target node, i.e. the `Delay Controller` applies the delay constraints modelled to the outgoing traffic. Note that, as there is one `Networking Node` per node belonging to a federation, the `Delay Controller` applies the delay modelled to those `Links` whose source node is the edge/fog/cloud node where it is deployed.

In this regard, tasks are transmitted by means of workflows in JSON format (Listing 1). This JSON code has among its fields the target node of the workflow (Listing 1, field *generatedBy*), i.e. where the workflow has to be sent. In this way, when the `Delay Controller` receives an outgoing workflow (interaction ①), it is able to identify the `Link` through which the workflow has to be sent. This also applies to the outgoing tasks processed in the node where the `Networking Node` is integrated (interaction ②).

Thus, with this information, the `Delay Controller` request to the `Synthetic Delay Generator` Ⓓ the current delay of the `Link` (interaction ③). Note that the delay is generated synthetically following user modelling. Once the response is received (interaction ④), the `Delay Controller` holds tasks during the received delay, thus simulating it. Finally, when the delay has been simulated, the traffic is forwarded to the `Bandwidth Controller` Ⓔ (interaction ⑤).

`Synthetic Delay Generator` Ⓓ Users can model the delay of each `Link` (average, minimum, maximum, etc.) that connects each node in a federation. Thus, the aim of the `Synthetic Delay Generator` is to generate the delay of each `Link` during simulation.

Thus, the `Synthetic Delay Generator` interacts with the `Delay Controller` Ⓒ and with the `Network Status Reporter` Ⓖ of the same `Networking Node`. In this way, when any of these components need to know the delay of a specific `Link`, they request it to the `Synthetic Delay Generator` (interaction ③ and ⑩). Then, the `Synthetic Delay Generator` responds to them with the delay of the `Link` requested (interaction ④ and ⑪).

`Bandwidth Controller` Ⓔ The `Bandwidth Controller` aims to apply the bandwidth constraints of the `Links` that connect the nodes belonging to a federation. Thus, the bandwidth is applied from source to target, i.e. the `Bandwidth Controller` applies the bandwidth constraints modelled to the outgoing traffic. Note that, as in the case of delay, the `Bandwidth Controller` applies the bandwidth modelled from source, to target.

Thus, the `Bandwidth Controller` receives traffic (workflows or results related to a processed task) from the `Delay Controller` (interaction ⑤). If traffic t_i is received and no traffic is being transmitted, the `Bandwidth Controller` holds the traffic t_i for the time resulting from applying the mathematical Expression 1. Thus, simulating the time that the traffic would have needed to be transmitted in a real environment.

$$TT_{t_i} = \frac{TS_{t_i}}{LB_{t_i}} \tag{1}$$

Where TT_{t_i} is the transmission time (seconds) required to send a traffic t_i of a specific size TS_{t_i} (bytes) through a `Link` with a bandwidth LB_{t_i} (bytes/seconds).

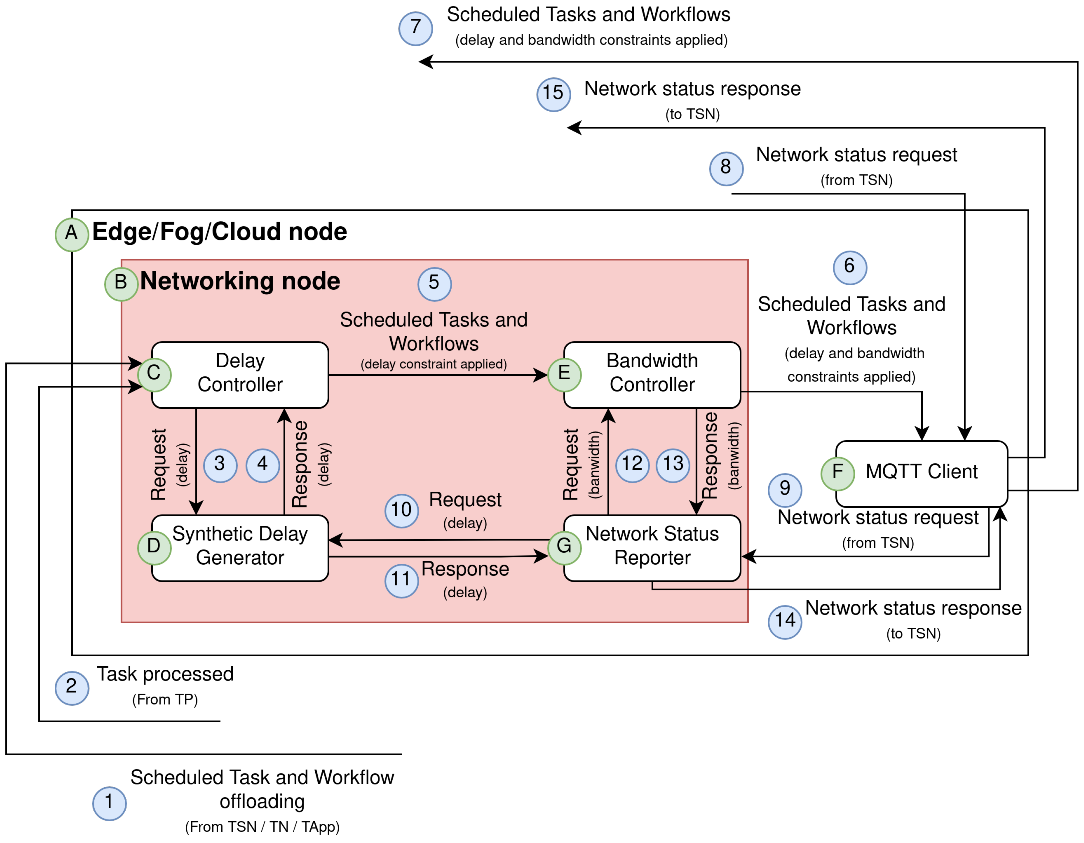
On the other hand, if a traffic t_n arrives at the `Bandwidth Controller`, but there are $n-1$ workflows or processed tasks (traffic) being transmitted or pending to be transmitted through the same `Link` over which the traffic t_n has to be transmitted, traffic t_n is queued in a FIFO (First In First Out) traffic queue. So, in this case, the transmission time of the traffic t_n can be determined by the Expression 2. Thus, simulating the time that the traffic t_n would have needed to be transmitted in a real environment.

$$TT_{t_n} = \left(\sum_{i=1}^{n} \frac{TS_{t_i}}{LB_{t_i}}\right) + \frac{RT_{t_0}}{LB_{t_i}} \tag{2}$$

Where a) TT_{t_n} is the transmission time required to send a traffic t_n with a size of TS_{t_n} bytes through a `Link` with a bandwidth of LB_{t_i} bytes, b) over which a workflow or processed task (traffic) t_0 is being transmitted and RT_{t_0} bytes of this traffic remain to be transmitted (when t_n arrives), and c) a set of $n-1$ workflows or processed tasks (traffic) are pending to be transmitted (queued before t_n).

Thus, once the delay and bandwidth constraints are applied, the traffic is sent to the `MQTT Client` (interaction ⑥), which forwards it to its target node (interaction ⑦). Finally, note that for the sake of clarity, interactions ⑫ and ⑬ are described below as part of the `Network Status Reporter` Ⓖ description.

`Network Status Reporter` Ⓖ The `Task Scheduler` of a federation could need to request the status of the `Links` (delay and bandwidth use) of the federation. Thus, it can use this data as input to perform the scheduling of the offloaded tasks. In this regard, the `Network Status Reporter` of each node belonging to a federation is the node that receives this request and responds to it with the current delay and available bandwidth of each `Link`. Note that, as there is one `Networking Node` per node belonging to a federation, the `Network Status Reporter` responds with the delay and bandwidth of those



7 Scheduled Tasks and Workflows
(delay and bandwidth constraints applied)
15 Network status response
(to TSN)
8 Network status request
(from TSN)
A Edge/Fog/Cloud node
B Networking node
5 Scheduled Tasks and Workflows
(delay constraint applied)
6 Scheduled Tasks and Workflows
(delay and bandwidth constraints applied)
C Delay Controller
E Bandwidth Controller
F MQTT Client
Request (delay)
3
4
Response (delay)
Request (banwidth)
12
13
Response (banwidth)
9 Network status request
(from TSN)
10 Request
(delay)
D Synthetic Delay Generator
G Network Status Reporter
11 Response
(delay)
14 Network status response
(to TSN)
2 Task processed
(From TP)
1 Scheduled Task and Workflow offloading
(From TSN / TN / TApp)

Figure 12 Networking Node component.

Links whose source node is the edge/fog/cloud node where it is deployed.

Since the Task Scheduler carries out these requests through the MQTT protocol, the MQTT Client Ⓕ is the first to receive them (interaction ⑧). Then, the MQTT Client forwards these requests to the Network Status Reporter. Following, the Network Status Reporter requests the current delay of each Link to the Synthetic Delay Generator (interactions ⑩ and ⑪) and the current use of bandwidth of each Link to the Bandwidth Controller Ⓔ (interactions ⑫ and ⑬). Once the Network Status Reporter has gathered all the requested data, it sends this data to the MQTT Client (interaction ⑭), which finally forwards the data to the Task Scheduler.

D. Task Processor

Figure 13 shows a generic Task Processor Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. The main components of the Task Processor are the Task Manager Ⓓ, the Task Performer Ⓔ and the Task Processor Status Reporter Ⓕ. Moreover, Figure 13 also shows how the Task Processor is deployed on an edge (Task Node), fog or cloud node Ⓐ and the interactions that these components could perform with other artefacts of the edge/fog/cloud node and with the rest of the IoT system. Below, the Task Processor is illustrated by describing each of its components and their interactions.

Task Manager Ⓓ The Task Manager aims to ensure that the schedule performed by the Task Scheduler of a federation is followed by the Task Processors that belong to the federation. Note that, as there is one Task Processor per node (with computing power) belonging to a federation, the Task Manager ensures that the schedule performed by the Task Scheduler is followed by the edge/fog/cloud node Ⓐ where it is deployed.

Thus, when tasks reach the Task Processors, as the tasks

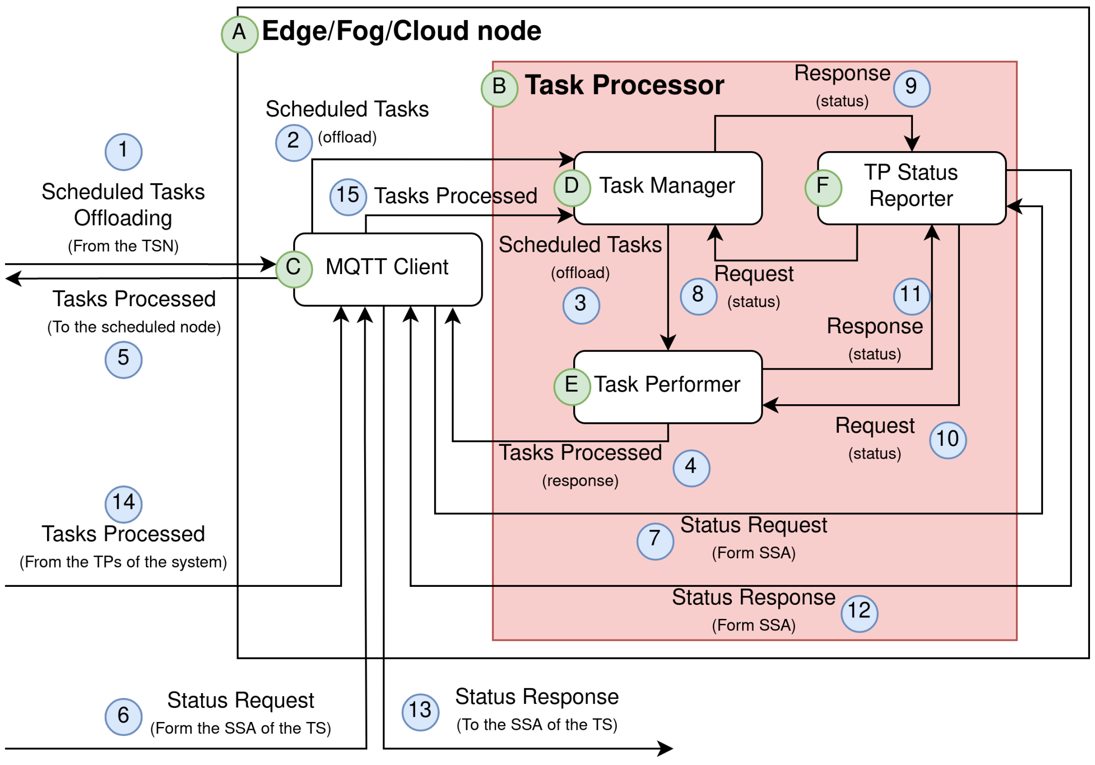


Figure 13 Task Proccessor components.

are sent through the system using the MQTT protocol, the first component that they reach is the `MQTT Client` Ⓒ of the computing nodes (interaction ①). Later, the `MQTT Client` forwards these tasks to the `TaskManager` (interaction ②).

The `Task Manager` Ⓓ receives, stores (buffer) and sends these tasks (following the schedule performed by the `Task Scheduler`) to the `Task Performer` Ⓔ (interaction ③), which performs their processing.

The `Task Manager` also handles the interdependency among tasks. Consequently, 1) the `Task Manager` holds dependent tasks until the arrival of the processing results of the tasks on which these tasks depend, 2) when these results arrive, the `Task Manager` includes in the task the processing results of the tasks on which it depended 3) Finally, following the schedule the `Task Manager` sends the task to the `Task Performer` for its processing (interaction ③).

Finally, note that for the sake of clarity, interactions ⑧ and ⑨ are described below, in the section reserved for the `Task Processor Status Reporter` Ⓕ.

`Task Performer` Ⓔ The `Task Performer` is the component of the `Task Processor` Ⓑ which performs the processing of the tasks. The `Task Performer` simulates the processing of the tasks holding them the time that would be required for their processing in a real environment. For this purpose, the `Task Performer` applies the expression 3.

$$PT_{t_i} = \frac{TS_{t_i}}{CF_{c_i}/CCB_{c_i}} \tag{3}$$

Where PT_{t_i} is the time (seconds) required to process the task t_i which has a size of TS_{t_i} bits on a CPU c_i with CCB_{p_i} cycles per bit (i.e the cycles that the CPU needs to process a bit) and a frequency of CF_{c_i} (i.e. cycles that the CPU can perform per second). Note that the parameters of the CPU are the CPU attributes that the user can specify to model the CPU of each node.

Once a task is processed, as transmitted in JSON, the `Task Performer` includes in it fields data such as the timestamp related to the start of the processing of the task and the timestamp related to the end of the processing of the task.

Then, the `Task Performer` sends the processed task to the `MQTT Client` (interaction ④), which forwards the processed task to their next target node. For the sake of clarity, interactions ⑩ and ⑪ are described below in the section reserved for the `Task Processor (TP) Status Report` Ⓕ.

Task Processor Status Reporter Ⓕ The Task Scheduler of the federation could need to know the status of the task processing, i.e. CPU use of each Task Processor Ⓑ and the tasks pending of processing. So, the Task Processor Status Reporter has the same aim that the Network Status Reporter of the Networking Node (Section C), although in the context of the Task Processor. Thus, in this case, the data that is reported is related to the use of the CPU and RAM of the Task Processor (by the Task Performer Ⓔ) and the status of the Task Manager Ⓓ (Tasks pending to be processed). Thus, the Task Scheduler can use this data as input to perform the scheduling of the offloaded tasks.

The Task Scheduler sends these requests through the MQTT protocol, so the MQTT Client Ⓒ is the first to receive them (interaction ⑥). Then, the MQTT Client forwards these requests to the Task Processor Status Reporter (interaction ⑦). Once the Task Processor Status Reporter receives a request, it requests the tasks pending to be processed (their size, estimated queue time, etc.) to the Task Manager Ⓓ (interaction ⑧), and the status of the use of the CPU and RAM to the Task Performer Ⓔ, (interaction ⑩).

When these components receive the requests from the Task Processor Status Reporter, they respond to it with the requested data (interactions ⑨ and ⑪). So, once the Task Processor Status Reporter gathers all the CPU, RAM and task processing related data, it forwards this data to the MQTT Client (interaction ⑫), which sends it to the Task Scheduler (interaction ⑬).

Chapter 8

Discussion, Conclusion and Future Works

"All we have to decide is what to do with the time that is given to us."

The Fellowship of the Ring (1954)
Tolkien, J. R. R.

8.1 Discussion

This section is dedicated to discussing the results obtained (see Chapter 2) through this Ph.D. Thesis. The discussion is structured around the RQs defined in Section 1.3, which are the foundation of this dissertation. Thereby, for each RQ, the discussion explores the extent to which the different contributions provide answers and insights to these questions. The discussion can be found below.

RQ1. ***To what extent are MDD techniques appropriate for developing tools and languages that can tackle effectively the complexity of IoT systems?***

Through the first contribution, the first release of SimulateIoT was delivered [2]. This first release captures in a metamodel the primary concepts and components of current IoT systems, such as sensors, actuators, cloud, and fog nodes among others. Conform to this metamodel, users are enabled to model IoT systems. Then, from these models, using the defined model-to-text transformations users can generate the code of the system, including the code to deploy and simulate it. Thus, this contribution proved the feasibility of tackling IoT systems' complexity using MDD techniques.

However, this first release encompasses only some of the primary components of IoT systems. Therefore, further research is required to conclusively state that MDD techniques are adequately suited to address the complexity of any IoT system effectively.

To this end, the domain of the IoT should be captured by SimulateIoT. Nevertheless, it is not feasible to cover the whole domain of IoT systems in a DSL due to its broad and ever-evolving nature [135]. For this reason, following the positive insights provided by the first contribution, it was considered to answer this RQ by extending SimulateIoT towards different and diverse IoT domains. Namely, the second contribution was focused on including FIWARE [3]. This was followed by a further extension to encompass IoT mobility in the third contribution [4]. The fourth contribution marked the integration of task scheduling functionalities into SimulateIoT [5]. Finally, in collaboration with the University of Aveiro, the system was further developed to include Big Data systems.

Between these extensions, the second one, the incorporation of FIWARE into SimulateIoT, should be highlighted. This is because this extension replaces the components of SimulateIoT, developed by the author of this Ph.D. Thesis, with third-party components from the FIWARE catalog. This is particularly insightful for this RQ given that these FIWARE components are real components that can be used in real IoT systems and not only in simulations. Moreover, they are not tailored to any IoT system in particular. So, its successful integration within SimulateIoT as target technology shows to what extent MDD can handle the heterogeneous technology of IoT systems, not only modeling but also generating, configuring, and deploying IoT systems.

In summary, the answer to RQ1 reveals the applicability of MDD techniques in the field of IoT systems. The progressive development and extension of SimulateIoT, through various contributions, have underscored the potential of MDD in capturing and managing the heterogeneous nature of IoT systems. While the initial version of SimulateIoT demonstrated the feasibility of using MDD to model and generate primary IoT components, i.e. the foundation of IoT systems, subsequent enhancements have expanded its scope, incorporating diverse IoT domains such as FIWARE, IoT mobility, task scheduling, and Big Data systems. So, these advancements collectively suggest that MDD techniques hold significant promise in effectively addressing the complexities inherent in IoT systems.

RQ2. ***To what degree are MDD techniques adequately suited for generating the simulation code necessary to simulate an IoT system?***

The first contribution of this Ph.D. Thesis outcome is an MDD-based simulator, SimulateIoT, able to handle the complexity of IoT systems. With SimulateIoT, it is possible to model IoT systems and generate the code of their components through the model-to-text transformation defined. Moreover, these model-to-text transformations can not only generate the necessary code to develop each component of the IoT system modeled but also can generate the required code to deploy and simulate it. For instance, synthetic sensor data generation modules simulate the data collection process.

However, similar to the limitations acknowledged in the previous RQ, the initial version of SimulateIoT addresses only a subset of the primary concepts inherent in IoT systems. This limitation underscores the need for ongoing research to conclusively determine the effectiveness of MDD techniques in generating simulation code capable of simulating any IoT system.

To expand upon this research question, the approach taken mirrors that of RQ1. Thus, SimulateIoT was extended to cover additional IoT domains, including the FIWARE IoT platform, IoT mobility, task scheduling, and Big Data systems within the IoT context. These extensions have been instrumental in illustrating the versatility of MDD techniques in generating comprehensive simulation code across diverse IoT domains, thereby reinforcing the validity of MDD techniques for this specific purpose.

In short, the answer to this RQ highlights the significant capability of MDD techniques in generating code to simulate IoT systems. The development of SimulateIoT, evolving through various stages and extensions, has not only addressed the primary concepts of IoT systems but also demonstrated the adaptability of MDD in diverse IoT domains. Thus, underscoring the potential of MDD in effectively creating comprehensive simulation code, a crucial aspect for testing IoT systems, which contributes to the advancement and practical application of IoT technologies.

RQ3. ***In what measure are MDD techniques effective in developing IoT simulation tools that not only offer adaptive integration capabilities and a user-friendly learning curve but also ensure agility in designing IoT systems and cost-efficiency in testing and validating them?***

Firstly, regarding the learning curve of the proposed simulator, note that SimulateIoT is a simulator based on MDD. MDD facilitates a focus on the high-level abstract concepts of the IoT, thereby simplifying the complexity associated with designing and conceptualizing these systems [136]. Additionally, MDD supports graphical modeling, which further reduces the intricacy of system design [136]. Consequently, SimulateIoT features a user-friendly learning curve, making it accessible to users. However, it is essential for users to have a foundational understanding of key IoT concepts and familiarity with MDD tools to fully leverage the simulator's capabilities.

Furthermore, models of IoT systems within SimulateIoT can be effortlessly modified, validated, generated, and simulated. Excluding the modification process, which depends on the user's skills and the extent of the modification, the remaining actions are almost instantaneous. This enables a swift and agile process in testing and redesigning, aiming to optimize the IoT system according to user requirements. This process is outlined in the methodology presented in the first contribution and is evidenced through the subsequent extensions applied to SimulateIoT.

In light of the above, it can be stated that SimulateIoT is a cost-efficient simulator for validating and testing IoT systems. However, it is important to note that this cost-efficiency may be comparatively lower in testing stages than other simulators. This is attributed to the fact that SimulateIoT predominantly relies on real components for conducting simulations, which results in higher hardware consumption costs than simulators based solely on mathematical models.

Lastly, regarding SimulateIoT's adaptative integration capabilities, its extension concerning task scheduling is strategically designed to allow users to incorporate their own task-scheduling algorithms. This feature positions SimulateIoT as an advantageous platform for users to thoroughly test their task-scheduling algorithms and compare them against other existing solutions. The primary requirement for users in this context is to acquire the input for their algorithms from SimulateIoT via API requests. Of course, this methodology is potentially applicable to other domains different from task scheduling. So, it can be stated that SimulateIoT presents high adaptative integration capabilities, as with this method, it can include even users' proposals that are not documented in the literature.

In conclusion, the simulation tools offered by SimulateIoT, which are based on MDD, provide a user-friendly learning curve for the simulation of IoT systems. Additionally, these tools offer agility and cost-effectiveness in both the design and testing stages, despite the potential for higher testing costs compared to alternative simulators. Furthermore, SimulateIoT demonstrates considerable adaptive integration capabilities, particularly shown through the task-scheduling extension. It is therefore possible to respond positively to this RQ in all the aspects it encompasses.

RQ4. ***To what extent is it feasible for a methodological approach grounded in MDD-based simulators to achieve an optimal IoT system design in terms of users' specific needs?***

Through the initial version of SimulateIoT, delivered through the first contribution, a methodology was developed that defines each step required to use SimulateIoT. Note that this methodology is derived primarily from MDD principles, rather than specific features or functionalities of SimulateIoT itself. In terms of methodological steps, it should be noted that MDD approaches enable domain-specific modeling and the generation of text, as code, from model-to-text transformations. SimulateIoT, as an MDD approach to simulate IoT systems, allows modeling these systems and generating all their related code. Moreover, given the nature of the generated code, it also allows deploying and simulating these systems. Thus, this methodology is based on these three stages and allows modeling, generating, and simulating IoT systems through SimulateIoT.

Thereby, this methodology helps users to use SimulateIoT to model, generate, and simulate IoT systems. In addition, it is also useful when redesigning IoT systems to optimize them if considered after analyzing the results of previous simulations. This is because the process is the same in this case, i.e., redesign (modeling), generate, and simulate. So, it is feasible to use this methodology iteratively until the system design that satisfies user requirements is achieved. Note that, although it is not explicitly specified in each of the use cases carried out in the works included in Chapters 4, 5, 6, 7, this methodology has been applied in each of them. Thus, showing its effectiveness.

So, in light of the results achieved, it is possible to answer positively to this RQ.

8.2 Conclusion

The Internet of Things (IoT) is characterized by its complex ecosystems. Firstly, these systems are complex because they comprise a wide array of heterogeneous technologies [135, 137]. Moreover, this complexity is heightened by the ever-evolving nature of the IoT, which leads to the ongoing introduction of new technologies, entailing even wider technological diversity [135, 138]. In addition, the lack of universally accepted standards for the development of these systems further complicates this scenario [139, 138], leading to a landscape where the development of IoT systems can become a major challenge.

Given this context, testing IoT systems designs before putting them into production is a suitable strategy to ensure the system behaves as expected. However, testing IoT systems is a costly and time-consuming process due to several reasons, such as device acquisition, device configuration, and system deployment among others. To avoid these shortcomings, simulation tools capable of simulating IoT systems are used to test them [125, 140]. Nevertheless, the simulation tools documented in literature often tackle simulations from a low-level abstraction, leading to simulators with a prominent learning curve coupled with low agility regarding designing and simulating IoT systems processes. Moreover, these simulators usually offer closed simulation capabilities, i.e. low integration adaptability, making it difficult for users to comprehensively test their systems or own IoT proposals.

Model-driven development (MDD) provides a set of techniques that enables elevating the level of abstraction, focusing on the high-level abstract concepts of a specific domain and on their relationships rather than on low-level details. In this work, an MDD-based IoT simulator called SimulateIoT has been proposed. This simulator takes advantage of the MDD to tackle IoT simulations from a high level of abstraction. With its metamodel, SimulateIoT enables modeling IoT systems and validate them. With its model-to-text transformations, it enables the generation of the code of each modeled component, together with the configuration files and deployment scripts required to set up the system and simulate it. Moreover, a methodology to facilitate and conduct these processes has been also developed.

SimulateIoT offers a comprehensive platform for simulating IoT systems, encompassing a broad spectrum of components such as foundational IoT elements, FIWARE architectures, mobile devices, task-scheduling nodes, processes, and big data within the IoT context. This versatility underscores the potential of MDD techniques not only in managing the inherent complexity of IoT systems through detailed modeling and validation capabilities but also in generating the simulation code required to simulate the modeled systems. Leveraging MDD principles, SimulateIoT approaches simulations with a high level of abstraction, significantly easing the learning curve for users to conduct such simulations. Rising the level of abstraction not only simplifies initial engagement with the tool but also enhances flexibility in the redesign and testing processes, pivotal for refining and achieving optimal IoT system designs. Besides, the developed methodology defines each step to conduct this process properly. Moreover, SimulateIoT presents a high adaptability to integration, particularly evidenced by its task scheduling extension. This extension was strategically designed to foster integration adaptability, enabling users to not only engage with existing simulation parameters but also to test and incorporate their own proposals within the SimulateIoT environment.

Thus, SimulateIoT shows to what extent MDD can be applied and leveraged for the simulation of IoT systems. Essentially, by elevating the level of abstraction, MDD enables SimulateIoT to address the complexity of IoT systems and to bridge the gaps often present on IoT simulators. Thus, delivering a set of simulation tools that enables testing IoT systems without the costs and efforts usually associated with this process.

8.3 Future work

Numerous promising research directions exist where SimulateIoT could be further expanded and explored. They are listed below:

- *Extend mobility simulation capabilities*: Firstly, as SimulateIoT is a simulator, it could be interesting to further improve its current domains of application. For instance, it could be interesting to extend the mobile capabilities by adding mobile sinks [141]. Mobile sinks are devices that move through IoT environments gathering the data collected by stationary sensors. Thus, sensors do not have to publish their data and therefore keep a connection with any device for that purpose, which are some of the most energy-consuming actions of these devices [141]. Measuring the energy savings led by these mobile sinks could be interesting to determine if it is suitable to use them in a specific IoT system.

- *Extend task-scheduling simulation capabilities*: Another interesting extension could be to enhance the task-scheduling capabilities of SimulateIoT. Currently, SimulateIoT enables users to test their own task-scheduling approaches. Nevertheless, it does not allow testing the policies that could govern some aspects of the task-scheduling environment [142], such as the priority in handling the results of a task once processed.

- *Simulation of Digital Twins*: On the other hand, different research directions such as Digital Twins could be also explored [143, 144]. SimulateIoT already simulates the infrastructure on which Digital Twins relies for their functioning, such as the sensors from which they gather data, the fog and cloud nodes where they deploy their intelligent systems, and the database systems they could use to communicate each of their layers. So, the simulation capabilities of SimulateIoT could be leveraged to test Digital Twins before putting them into production.

- *Inclusion of blockchain simulation capabilities*: It could be interesting to include blockchain technology within the SimulateIoT framework to explore its potential to enhance IoT security and data integrity [145]. Blockchain technology, with its decentralized and immutable ledger system, offers a robust solution for secure, transparent transactions and data sharing across IoT networks [145]. By integrating blockchain simulations, SimulateIoT could provide a valuable platform for researchers and developers to investigate the scalability, performance, and security implications of blockchain within IoT ecosystems. This integration could facilitate the testing of smart contracts, peer-to-peer transactions, and decentralized applications in simulated IoT environments, thereby offering insights into optimal blockchain configurations and protocols for IoT.

Therefore, the future development of SimulateIoT encompasses a broad spectrum of promising enhancements and explorations into new domains. Note that the outlined future directions not only seek to extend SimulateIoT's utility but also to contribute to the evolution of IoT technologies, ensuring a more efficient and sustainable future for IoT systems.

8.4 Reflections and Personal Insights

Toward the end of my Software Engineering degree, whispers of Ph.D. theses and esteemed doctors began to permeate my academic circles. Initially, it seemed like a pursuit reserved for the smartest ones, distant from my own aspirations.

Upon completing my undergraduate studies, the idea of pursuing a Master's in computer science surfaced. It felt premature to depart from the university environment after just four years, especially considering that the time passed very quickly. During my Master's, I had the privilege of working closely with my Thesis Director, Professor Pedro José Clemente, who introduced me to the world of research, expanding my knowledge about the University and the kinds of things that people who work there do.

Thus began my journey into research. Over time, and after finishing my Master's, almost imperceptibly, I found myself immersed in my university's

Ph.D. program in Information Technologies. It dawned on me that a Ph.D. wasn't solely reserved for the academic elite, it was also for those who have a great passion for what they do every day, and for those who do not mind struggling as much as necessary to achieve their goals.

So, my journey has been one of struggling for almost four years, the time it took to complete my thesis. But also one of passion, because I have loved what I have done. It's been a path where I have tried to play my part in society's progress. For me, that meant diving into IoT simulations, with the goal of aiding IoT researchers and practitioners in making their own contributions to the field. It may be a small effort in the bigger picture, but as Mother Teresa once said, ”We ourselves feel that what we are doing is just a drop in the ocean. But the ocean would be less because of that missing drop.” So, with this thesis, my hope is that people involved in the IoT field can benefit from my small contribution to the vast ocean of IoT possibilities.

Appendix A

SimulateIoT: A model-driven approach to simulate IoT systems*

Authors: José A. Barriga and Pedro J. Clemente
Title: SimulateIoT: A model-driven approach to simulate IoT systems*
Year: 2022
Conference: Doctoral Consortium in Computer Science (JIPII)
Nature: International

SimulateIoT: A model-driven approach to simulate IoT systems[⋆]

José A. Barriga[1][0000−0001−8377−1860] and Pedro J. Clemente[1][0000−0001−5795−6343]

Quercus Software Engineering Group. http://quercusseg.unex.es. Department of Computer Science. University of Extremadura, Av. Universidad s/n, 10003, Cáceres (Spain)
{jose,pjclemente}@unex.es

Abstract. Developing, deploying and testing IoT projects require high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software. However, in order to decrease the cost associated to develop and test the IoT system it can be previously simulated. In this regard, 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. Model-driven development 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, SimulateIoT, is proposed to define, generate code and deploy IoT systems simulations. The IoT simulation environment generated from each model includes the sensors, actuators, fog nodes, cloud nodes and analytical characteristics. Additionally, a case study, focused on an Industrial IoT environment is presented to show the simulation expressiveness.

Keywords: IoT systems · IoT simulation · model-driven development · model to text transformation.

1 Introduction

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 systems is one way to decrease this initial investment because the users can measure and resize the artefacts needed to deploy and interconnect

[⋆] This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

the systems. Some of the most relevant simulators are: Contiki-Cooja [10], OM-NeT++ [13], IoT-Lab [9], CupCarbon [5] or IoTSim-Edge [3]. 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.

Model-Driven Development (MDD) [11] is an emerging software engineering research area that aims to develop software guided by models based on Meta-modeling technique. In MDD, a MetaModel defines the domain concepts and relationships in a specific domain in order to model partial reality. A Model defines a concrete system conform to a Metamodel. Then, from these models it is possible to generate totally or partially the application code by model-to-text transformations [12]. 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, MDD is proposed to tackle this heterogeneous technology (devices, actuators, complex event processing engines, notification technology, publish-subscribe communication protocol, etc.) by increasing the *abstraction level* where the software is implemented, focusing on the domain concepts and their relationships.

The main contributions of this paper include:

– Evidence that 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 Model-Driven solution for researchers and practitioners that allows them to design and simulate IoT systems by defining a *SimulateIoT* metamodel, a graphical concrete syntax (graphical editor) to define models and a model-to-text transformation towards the code generation for specific IoT simulation environment. It includes the code generation to execute the IoT simulation. Furthermore, the IoT system generated can be deployed.
– The application of *SimulateIoT* to one case study focused on a generic Industrial IoT system (IIoT).

The rest of the paper is structured as follows. In Section 2, we present *SimulateIoT*, including the *SimulateIoT* metamodel, the graphical editor and the model-to-text transformation developed. In Section 3 an Industrial IoT case study is presented. Finally, Section 4 concludes the paper.

2 SimulateIoT Overview

SimulateIoT [1] is a model-driven approach to design, generate code and execute IoT simulations. The main components of SimulateIoT are: a) The Abstract Syntax or Metamodel; b) The Concrete Syntax or Graphical editor; and c) The Model-to-Text Transformations.

A) SimulateIoT Metamodel: In the context of Model-Driven Development, a MetaModel defines the concepts and relationships in a specific domain in order to model partially reality [11]. Later, Models conform to the MetaModel could be defined and they could be used to generate total or partially the application code. The software code could be generated for a specific technological platform, improving its technological independence and decreasing the error proneness.

The SimulateIoT metamodel (available in [2]) includes concepts related to sensors, actuators, databases, fog and cloud nodes, data generation, communication protocols, stream processing, and deploying strategies, among others.

B) SimulateIoT Graphical Concrete Syntax and Validator: In order to facilitate modelig IoT environments, a Graphical Concrete Syntax (Graphical editor) has been generated using the Eugenia tool [4]. 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) [8] based) can be validated against the defined metamodel (EMF and OCL based). Figure 1 B shows an example of model defined by using the Graphical Concrete Syntax generated.

C) SimulateIoT Model to Text Transformations: Once the models have been defined and validated conforming to the *SimulateIoT metamodel*, a model-to-text transformation defined using Acceleo [7] can generate several artefacts. Thus, the generated software includes, MQTT messaging broker (based on MQTT protocol [6]), device infrastructure, databases, a graphical analysis platform, a stream processor engine, docker container, etc.

Figure 1 A shows the deployment of the architecture of a generic IoT environment where the above mentioned artefacts and their interactions can be observed. Note that the deployment of the architecture is carried out by running the deployment script that is included in the model-to-text transformations (script that includes all the configurations defined in the previous system modelling).

3 Case study. Industrial IoT

In this section, an Industrial IoT environment is modeled and simulated by using SimulateIoT tools. It defines an IIoT where a supplier and its customers manage both product orders and shipments. This is a generic use case, i.e. the proposed environment could represent any type of industrial environment where a supplier delivers products to its customers or where a specific sensor triggers an alert to act.

3.1 Case study. Model definition

– Each industrial placement (customer) is provided with a set of devices (*Sensor* node, Figure 1 B, label 3.1) which is capable of measuring the stock of their products (i.e. the level of a concrete fluid).

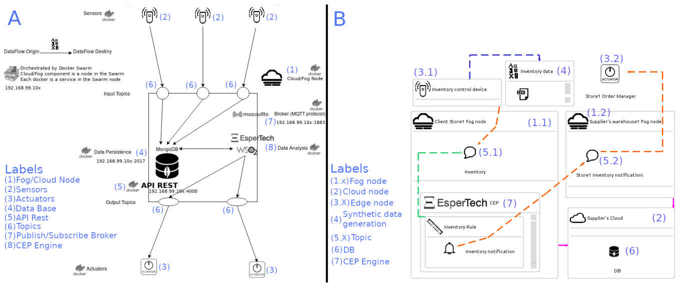


Fig. 1. A) Example of deploy diagram. B) Case study. A generic Industrial IoT system model.

- Each industrial placement (customer) is provided with a `Fog` node (Figure 1 B, label 1.1) capable of analysing (Figure 1 B, label 7) the data published by the stock devices (`Sensor) nodes`. In case of stock shortage (defined by a set of rules), the *Fog* node informs the supplier.
- The supplier has a series of warehouses, all of them equipped with `Fog` nodes (Figure 1 B, label 1.2) capable of receiving (Figure 1 B, label 5.2) and processing the information supplied by the industrial placements (stock status).
- Each warehouse also has an `Actuator` node (Figure 1 B, label 3.2) that receives the information related to the customer's stock and manages them (stock status to orders).
- Finally, the provider has a `Cloud` node (Figure 1 B, label 2) through which it is able to receive all the information from the environment (`Fog` nodes) and store them (Figure 1 B, label 6).

3.2 Case study. 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*.

Thus, executing the simulation modelled and later on deploying it, makes it possible to analyse the final IoT environment before it is implemented and deployed.

4 Conclusions

The MDD approach proposed in this paper, SimulateIoT, shows that Model-driven development techniques are a suitable way to tackle the complexity of domains where heterogeneous technologies are integrated. Besides, SimulateIoT helps users to design, generate, deploy, analyse, and optimise their IoT systems, streamlining the process of IoT systems development and saving costs.

References

1. Barriga, J.A., Clemente, P.J., Sosa-Sánchez, E., Prieto, E.: Simulateiot: Domain specific language to design, code generation and execute iot simulation environments. IEEE Access **9**, 92531–92552 (2021)
2. Barriga Corchero, J.A., Clemente, P.J.: "simulateiot metamodel", mendeley data, v1 (2022). https://doi.org/10.17632/4mmgv82k2c.1
3. Jha, D.N., Alwasel, K., Alshoshan, A., Huang, X., Naha, R.K., Battula, S.K., Garg, S., Puthal, D., James, P., Zomaya, A., et al.: Iotsim-edge: A simulation framework for modeling the behavior of internet of things and edge computing environments. Software: Practice and Experience **50**(6), 844–867 (2020)
4. Kolovos, D.S., García-Domínguez, A., Rose, L.M., Paige, R.F.: Eugenia: towards disciplined and automated development of GMF-based graphical model editors. Software & Systems Modeling pp. 1–27 (2015)
5. Mehdi, K., Lounis, M., Bounceur, A., Kechadi, T.: Cupcarbon: A multi-agent and discrete event wireless sensor network design and simulation tool. In: 7th International ICST Conference on Simulation Tools and Techniques, Lisbon, Portugal, 17-19 March 2014. pp. 126–131. Institute for Computer Science, Social Informatics and Telecommunications Engineering (ICST) (2014)
6. Oasis: Message queuing telemetry transport (mqtt) v5.0 oasis standard (2019), https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (Retrieved: 2022-06-13)
7. Obeo: Acceleo project (2012), http://www.acceleo.org (Retrieved: 2022-06-13)
8. OMG: OMG Object Constraint Language (OCL), Version 2.3.1 (January 2012), http://www.omg.org/spec/OCL/2.3.1/ (Retrieved: 2022-06-13)
9. Papadopoulos, G.Z., Beaudaux, J., Gallais, A., Noel, T., Schreiner, G.: Adding value to WSN simulation using the IoT-LAB experimental platform. International Conference on Wireless and Mobile Computing, Networking and Communications pp. 485–490 (2013)
10. Sehgal, A.: Using the Contiki Cooja simulator. Computer Science, Jacobs University Bremen Campus Ring, Technical Report (2013)
11. Selic, B.: The pragmatics of model-driven development. IEEE software **20**(5), 19–25 (2003)
12. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE software **20**(5), 42–45 (2003)
13. Varga, A., Hornig, R.: An overview of the omnet++ simulation environment. In: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops. p. 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2008)

Appendix B

Designing and simulating IoT environments by using a model-driven approach*

Authors: José A. Barriga and Pedro J. Clemente
Title: Designing and simulating IoT environments by using a model-driven approach*
Year: 2022
Conference: Iberian Conference on Information Systems and Technologies (CISTI)
Nature: International

Designing and simulating IoT environments by using a model-driven approach*

José A. Barriga and Pedro J. Clemente
Quercus Software Engineering Group. http://quercusseg.unex.es
Department of Computer and Telematic Systems Engineering. Universidad de Extremadura (ROR:https://ror.org/0174shg90)
Cáceres, Spain
e-mail: {jose, pjclemente}@unex.es

Abstract — **Developing, deploying and testing IoT projects require high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software. However, in order to decrease the cost associated to develop and test the IoT system it can be previously simulated. In this regard, 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. Model-driven development 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 is applied to define, generate code and deploy IoT systems simulations. The IoT simulation environment generated from each model includes the sensors, actuators, fog nodes, cloud nodes and analytical characteristics. Additionally, a case study, focused on an Industrial IoT environment is presented to show the simulation expressiveness.**

Keywords - IoT systems; IoT simulation; model-driven development; model to text transformation.

I. Introduction

The Internet of Things (IoT) is widely applied in several areas such as smart-cities, home environments, agriculture, industry, intelligent buildings, etc. [19]. 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 resize the artefacts needed to deploy and interconnect the systems. 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.

Model-Driven Development (MDD) 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 1). Thus, a Model (M1) is conform to a MetaModel (M2). Moreover, a Metamodel conforms to a MetaMetaModel (M3) which is reflexive [1]. The MetaMetaModel level is represented

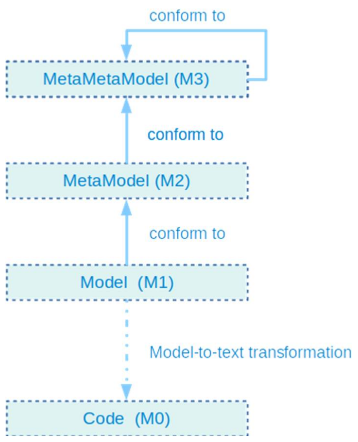


Figure 1. Model-Driven Development. Four layers of metamodeling.

by well-known standards and specifications such as Meta-Object Facilities (MOF) [10], ECore in EMF [21] 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 transformation [17]. 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, MDD is proposed to tackle this heterogeneous technology (devices, actuators, complex event processing engines, notification technology, publish-subscribe communication protocol, etc.). Model-Driven Development [5], [7], [15], [16] 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

* This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

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 Model-Driven solution that allows to design and simulate IoT environments 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.
- The application of *SimulateIoT* to one case study focused on a Smart building IoT System.

The rest of the paper is structured as follows. In Section II, we give an overview of existing IoT simulation approaches focused on high level IoT simulation environments. In Section III, we present *SimulateIoT*, including the *SimulateIoT* metamodel, the graphical editor and the model-to-text transformation developed. In Section IV an Industrial IoT case study is presented. Finally, Section V concludes the paper.

II. Related works

This section review several works related to high level modeling IoT environments such as COMFIT [4], IoTSuite [14], [20] and CupCarbon [9].

COMFIT [4] was a cloud environment to develop the Internet of Things system. It used model-driven techniques included in the Model-Driven Architecture (MDA) specification [6]. 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 [14], [20] 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.

Other approaches focus on simulating IoT systems proposing specific tools [3], [9], [18]. Thus, CupCarbon [9] 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.

III. SimulateIoT Overview

SimulateIoT [2] is a model-driven approach to design, generate code and execute IoT simulations. It has been applied on SmartBuildings and SmartAgro areas. In this section, how use SimulateIoT for modeling Industrial IoT environments is described.

In this regard, in order to offer a self-contained paper in this section SimulateIoT is briefly described. For this purpose, the main components of SimulateIoT are described: a) The Abstract Syntax or Metamodel; b) The Concrete Syntax or Graphical editor; and c) The Model-to-Text Transformations.

A. *SimulateIoT Metamodel*

In the context of Model-Driven Development, a MetaModel defines the concepts and relationships in a specific domain in order to model partially reality [16]. Later, Models conform to the MetaModel could be defined and they could be used to generate total or partially the application code. As aforementioned, the software code could be generated for a specific technological platform, improving its technological independence and decreasing the error proneness.

Figure 2 shows the SimulateIoT metamodel including concepts related to sensors, actuators, databases, fog and cloud nodes, data generation, communication protocols, stream processing, and deploying strategies, among others. Below, the main classes and relationships of the metamodel are described.

In this regard, the abstract class *Node* allows defining nodes to the IoT environment. These nodes can be of two types: a) nodes belonging to the Edge layer (*EdgeNode* abstract class) such as sensors and actuators (*Sensor* and *Actuator* classes); or b) processing nodes belonging to the Fog layer and the Cloud layer (*FogNode* and *CloudNode* classes).

The main objective of an *EdgeNode* is to publish data (*Sensor* class) or to subscribe to data (*Actuator* class). The publication and subscription of data is carried out through Topics (*Topic* class). Regarding the data publication by sensors, several aspects can be modelled (*Data* class), such as the source of the data (*DataSource* class), the type of synthetic data generation (*DataGeneration* class), etc.

On the other hand, the processing nodes, Fog and Cloud nodes, are responsible for supporting the IoT architecture for data processing. In this sense, both the Fog and Cloud nodes focus on offer several services: Topics to the Edge nodes, data persistence service (*DataBase* class), data analysis service through CEP (Complex Event Processing) engines (*Process Engine* abstract class), notification service (*Notification* class), etc.

B. *SimulateIoT Design and Implementation phase. Graphical Concrete Syntax and Validator*

In order to facilitate modelling IoT environments, a Graphical Concrete Syntax (Graphical editor) has been generated using the Eugenia tool [8]. The Graphical Concrete Syntax generated from SimulateIoT metamodel is based on Eclipse GMF (Graphical Modeling Framework) and EMF (Eclipse Modeling Tools).

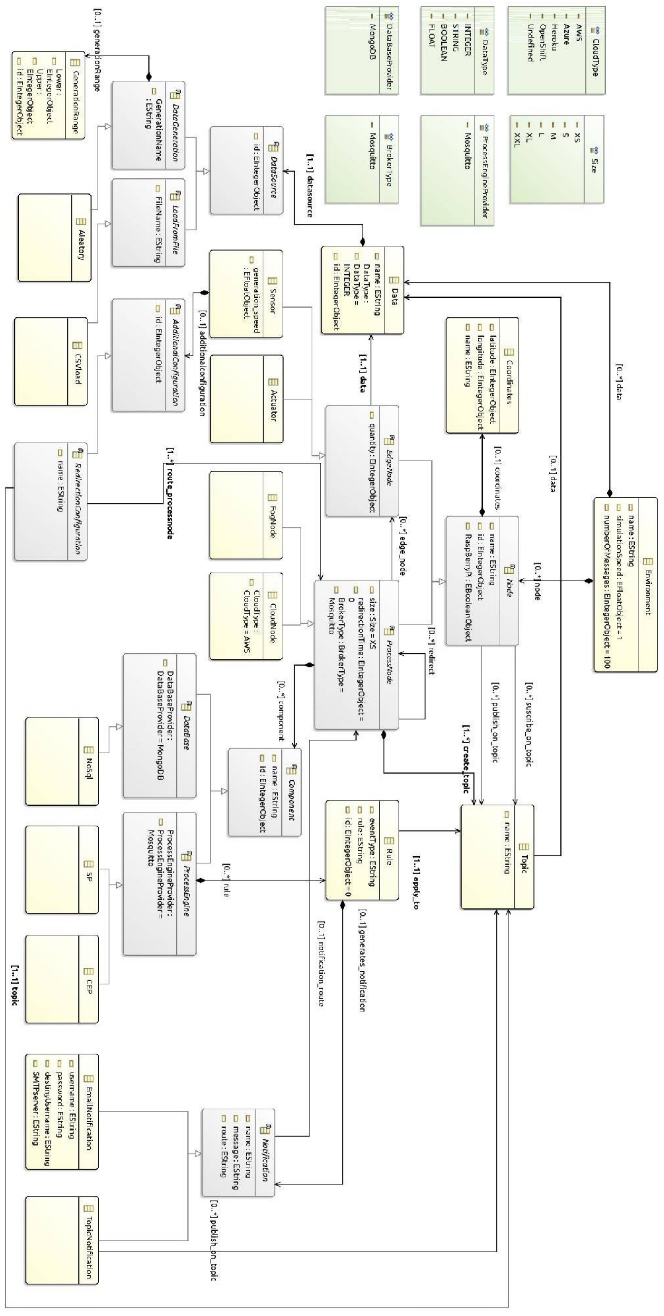


Figure 2. SimulateIoT Metamodel.

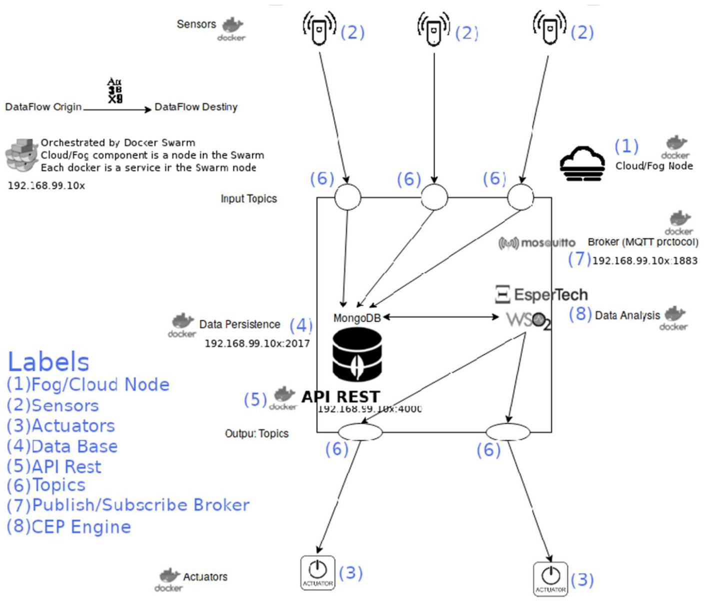


Figure 3. Example of deploy diagram. The main components are identified by numbers in the figure.

Consequently, models (EMF and OCL (Object Constraint Language) [13] based) can be validated against the defined metamodel (EMF and OCL based). Figure 4 shows an example of model defined by using the Graphical Concrete Syntax generated.

C. *SimulateIoT Design and Implementation phase. Model to Text Transformations*

Once the models have been defined and validated conforming to the *SimulateIoT metamodel*, a model-to-text transformation defined using Acceleo [12] can generate several artefacts. Thus, the generated software includes, MQTT messaging broker (based on MQTT protocol [11]), device infrastructure, databases, a graphical analysis platform, a stream processor engine, docker container, etc. Later, during deployment phase all the artefacts generated from the models are deployed. So, several software artefacts such as the MQTT messaging broker, device infrastructure, databases, graphical analysis platform, etc. are configured and deployed.

Figure 3 shows the deployment of the architecture of a generic IoT environment where the above mentioned artefacts and their interactions can be observed.

IV. CASE STUDY. INDUSTRIAL IOT

In this section, an Industrial IoT environment is modeled and simulated by using SimulateIoT tools.

It defines an IIoT where a supplier and its customers manage both product orders and shipments. This is a generic use case, i.e. the proposed environment could represent any type of industrial environment where a supplier delivers products to its customers or where a specific sensor triggers an alert to act.

In order to model this IoT system the following aspects are considered:

- Each industrial placement (customer) is provided with a set of devices (*Sensor* node) which is capable of measuring the stock of their products (i.e. the level of a concrete fluid).
- Each industrial placement (customer) is provided with a Fog node capable of analysing the data published by the stock devices (Sensor) nodes. In case of stock shortage (defined by a set of rules), the *Fog* node informs the supplier.
- The supplier has a series of warehouses, all of them equipped with Fog nodes capable of receiving and processing the information supplied by the industrial placements (stock status).
- Each warehouse also has an Actuator node that receives the information related to the customer's stock and manages them (stock status to orders).
- Finally, the provider has a Cloud node through which it is able to receive all the information from the environment (Fog nodes) and store them.

A. *Case study. Model definition*

Figure 4 shows an excerpt from the IoT model. It also includes numerical references for each node which are then used to describe the use case.

First, the architecture modelled for the customers, in particular their Edge layer, is analysed. As mentioned above, each industrial placement (customer) has a number of devices capable of measuring the stock of their products. This device is modelled by the Sensor node called *Inventory control device* (Figure 4, label 3.1). Note that for simplicity, only the modelling of one device is shown in the model, however, depending on the complexity of a real customer's product stock, there could be dozens or hundreds of such devices (measuring different kinds

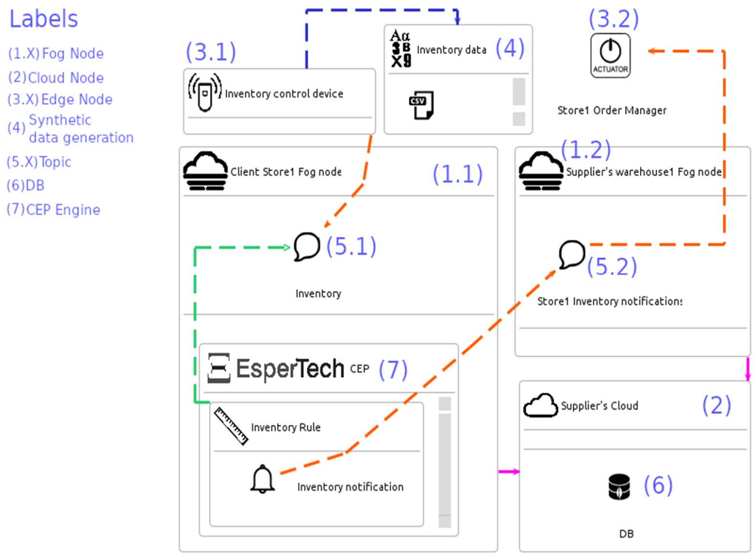


Figure 4. Case study. A generic Industrial IoT system. The main components are identified by numbers in the figure.

of stock). On the other hand, the *Inventory control device* is related to a synthetic data generation (Figure 4, label 4) which represents and simulates the stock of a product. This synthetic data generation uses a CSV file as its data source. It will be the data from this CSV that the sensor will publish during the execution of the simulation.

Each industrial placement (customer) are also provided with a Fog layer represented by a Fog node in the model (Figure 4, label 1.1). Note that, for simplicity's sake, only one Fog node (a customer) is shown. Also, note that depending on the complexity of the customer, more than one Fog node may be needed. Next, the Fog node presents a Topic node called *Inventory* (Figure 4, label 5.1). This Topic is offered to stock devices, so that they can publish their stock data here. As in the previous cases, there could be more than one Topic node of this kind, so that different stock devices could publish their data in the most appropriate one. For instance, in a hypothetical agricultural holding, the stock of grain could be published in one Topic, the stock of water in a different one, etc. Finally, each Fog node is provided with a CEP engine which is able to analyse the stock data (Figure 4, label 7). In case the stock falls below a certain threshold, the CEP engine sends a notification to the supplier. In the modelled use case, it sends the notification to the *Fog* node implemented in the supplier's warehouse assigned (for order management) to this customer.

As for the provider architecture, a Fog node called *Supplier's warehouse1 Fog node* is shown in the model (Figure 4, label 1.2). This Fog node is the one related to the warehouse assigned to the modelled customer. This warehouse Fog node receives the stock notifications from its assigned customers through the Topic *Store1 Inventory notification* (Figure 4, label 5.2). Again, in this warehouse Fog node several topics could coexist, for example, one for each customer assigned to this warehouse. Following the description of the supplier's architecture, the model shows the Actuator *Store1 Order Manager* (Figure 4, label 3.2), which receives the customer's stock notifications (by subscription to the Topic) and manages the orders or shipments of products

Finally, the Provider defines a Cloud node (Figure 4, label 2) that receives all the information from the Fog nodes of the architecture. In this use case, this information is processed and stored in this Cloud node for further use (the database where the information is stored is shown in Figure 4, label 6).

B. *Case study. 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*.

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 ProcessNode 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.

V. CONCLUSIONS

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 Industrial IoT system (IIoT), to propose several IoT alternatives and policies in order to achieve a suitable IIoT architecture. Finally, the IIoT systems modelled can be deployed and analysed.

Acknowledgment

This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

JUNTA DE EXTREMADURA
Consejería de Economía, Ciencia y Agenda Digital

Cofinanciado por la Unión Europea

References

[1] Atkinson, Colin, and Thomas Kuhne. 2003. "Model-Driven Development: A Metamodeling Foundation." *IEEE Software* 20 (5): 36–41.

[2] Barriga, José A., Pedro J. Clemente, Encarna Sosa-Sánchez, and Álvaro E. Prieto. 2021. "SimulateIoT: Domain Specific Language to Design, Code Generation and Execute Iot Simulation Environments." *IEEE Access* 9: 92531–52. https://doi.org/10.1109/ACCESS.2021.3092528**Error! Hyperlink reference not valid.**

[3] Bevywise. 2018. "Bevywise Iot Simulator. Https://Www.bevywise.com/Iot-Simulator/."

[4] Farias, Claudio M De, Italo C Brito, Luci Pirmez, Flávia C Delicato, Paulo F Pires, Taniro C Rodrigues, Igor L Santos, Luiz F R C Carmo, and Thais Batista. 2016. "COMFIT: A development environment for the Internet of Things." *Future Generation Computer Systems*, no. i. https://doi.org/10.1016/j.future.2016.06.031.X

[5] France, Robert, and Bernhard Rumpe. 2007. "Model-Driven Development of Complex Software: A Research Roadmap." In *2007 Future of Software Engineering*, 37–54. IEEE Computer Society.

[6] Group, Object Management. 2014. "MDA Guide Revision." https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.X

[7] Hailpern, Brent, and Peri Tarr. 2006. "Model-Driven Development: The Good, the Bad, and the Ugly." *IBM Systems Journal* 45 (3): 451–61.

[8] Kolovos, Dimitrios S, Antonio García-Domínguez, Louis M Rose, and Richard F Paige. 2015. "Eugenia: Towards Disciplined and Automated Development of GMF-Based Graphical Model Editors." *Software & Systems Modeling*, 1–27.

[9] Mehdi, Kamal, Massinissa Lounis, Ahcène Bounceur, and Tahar Kechadi. 2014. "Cupcarbon: A Multi-Agent and Discrete Event Wireless Sensor Network Design and Simulation Tool." In *7th International Icst Conference on Simulation Tools and Techniques, Lisbon, Portugal, 17-19 March 2014*, 126–31. Institute for Computer Science, Social Informatics; Telecommunications Engineering (ICST).

[10] Meta Object Facility (MOF). 2016. *Meta Object Facility (MOF) Core Specification Version 2.5.1*. OMG Available Specification. Object Management Group. http://www.omg.org/spec/MOF/2.5.1/.X

[11] Oasis. 2019. "Message Queuing Telemetry Transport (Mqtt) V5.0 Oasis Standard." https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.X

[12] Obeo. 2012. "Acceleo Project Http://Www.acceleo.org."

[13] OMG. 2012. "OMG Object Constraint Language (OCL), Version 2.3.1." Object Management Group; Object Management Group. http://www.omg.org/spec/OCL/2.3.1/.X

[14] Patel, Pankesh, and Damien Cassou. 2015. "Enabling High-Level Application Development for the Internet of Things." *Journal of Systems and Software* 103: 62–84.

[15] Schmidt, Douglas C. 2006. "Model-Driven Engineering." *COMPUTER-IEEE COMPUTER SOCIETY-* 39 (2): 25.

[16] Selic, Bran. 2003. "The Pragmatics of Model-Driven Development." *IEEE Software* 20 (5): 19–25.

[17] Sendall, Shane, and Wojtek Kozaczynski. 2003. "Model Transformation: The Heart and Soul of Model-Driven Software Development." *IEEE Software* 20 (5): 42–45.

[18] Siafu. 2007. "Siafu. An Open source context simulator. http://siafusimulator.org/."

[19] Siow, Eugene, Thanassis Tiropanis, and Wendy Hall. 2018. "Analytics for the Internet of Things: A Survey." *ACM Computing Surveys (CSUR)* 51 (4): 74.

[20] Soukaras, Dimitris, Pankesh Patel, Hui Song, and Sanjay Chaudhary. 2015. "IoTSuite: A Toolsuite for Prototyping Internet of Things Applications." In *The 4th International Workshop on Computing and Networking for Internet of Things (Comnet-Iot), Co-Located with 16th International Conference on Distributed Computing and Networking (Icdcn)*, 6.

[21] Steinberg, David, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0*. 2nd ed. Addison-Wesley Professional.

Appendix C

SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments (Summary)*

Authors: José A. Barriga, Pedro J. Clemente, Encarna Sosa-Sánchez and Álvaro E. Prieto
Title: SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments (Summary)*
Year: 2022
Conference: Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS)
Nature: National

SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments (Summary)*

Jose A. Barriga[1][0000−0001−8377−1860], Pedro J. Clemente[1][0000−0001−5795−6343], Encarna Sosa-Sánchez[1][0000−0002−0267−5875], and Álvaro E. Prieto[1][0000−0002−2312−4589]

Quercus Software Engineering Group. http://quercusseg.unex.es. Department of Computer Science. University of Extremadura, Av. Universidad s/n, 10003, Cáceres (Spain)
`{jose, pjclemente, esosa, aeprieto}@unex.es`

Summary of the Contribution

Developing, deploying and testing IoT projects require high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software. However, in order to decrease the cost associated to develop and test the IoT system it can be previously simulated. 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. Model-driven development 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, SimulateIoT, is proposed to define, generate code and deploy IoT systems simulations. Additionally, two case studies, focused on smart building and agriculture IoT systems, are presented to show the simulation expressiveness.

Keywords: IoT systems · IoT simulation · fog computing · model-driven development · model to text transformation · data analysis

Acknowledgements

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

* This work has been published in IEEE Access, vol. 9, year. 2021 JCR 2020: 3.367, 58/223, Q2. https://doi.org/10.1109/ACCESS.2021.3092528

Appendix D

Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment

Authors: José A. Barriga
Title: Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment
Year: 2022
Conference: International Conference on Service-Oriented Computing (ICSOC)
Nature: International

Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment

José A. Barriga( )

Quercus Software Engineering Group, Department of Computer and Telematic Systems Engineering, University of Extremadura, Av. Universidad s/n, 10003 Cáceres, Spain
`jose@unex.es`
`http://quercusseg.unex.es`

Abstract. In the context of IoT systems, the use of services is a key element in managing system complexity. Concepts such as service-oriented computing/architecture or quality of service (QoS) are present in many IoT systems and are the aim of several studies. However, the analysis and assessment of the behaviour of these concepts requires the deployment of the IoT system, implying high investments in hardware and software. Thus, in order to decrease these costs, the system can be simulated. In this regard, IoT simulations have been tackled focusing on low level aspects such as networks, motes, etc. rather than on high-level concepts, such as services or computing layers. In this proposal, a model-driven development approach named SimulateIoT is proposed to model, generate code and deploy IoT systems simulations from a high abstraction level (from models). Besides of modeling the IoT environment call generation, the IoT system could be simulated. From these simulations it is possible to assess QoS-related aspects such as the delay or jitter between two nodes, the variation of delay or jitter over time, the use of bandwidth, the packet loss, the variation of these parameters as the system changes (e.g. increase of sensors), check whether Service level agreements (SLA) are met, etc. In order to show the proposal, a case study, focused on an Internet of Vehicles (IoV) system is presented.

Keywords: IoT systems · IoT simulation · Model-driven development · Quality of service · Service-oriented computing

This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).
Supervised by Pedro J. Clemente [0000-0001-5795-6343], Quercus Software Engineering Group, Department of Computer and Telematic Systems Engineering, University of Extremadura, Av. Universidad s/n, 10003, Cáceres, Spain, pjclemente@unex.es

1 Introduction

An IoT system involves different devices and services belonging to the Mist, Edge, Fog or Cloud layers. Handling the technological heterogeneity underlying IoT systems requires overcoming a learning curve and investing time and money in system development and hardware acquisition.

For this reason, in the literature, around 90% of studies that need to corroborate or test their proposals in an IoT system use simulators [4]. However, although there are several simulators in the literature (Contiki-Cooja [11], OMNeT++ [14], CupCarbon [7] or IoTSim-Edge [5]), they generally focus on modeling the system at a low level of abstraction rather than focusing on the high level IoT domain concepts and their relationships.

Model-Driven Development (MDD) [12] is an emerging software engineering research area that aims to develop software guided by models based on Metamodeling technique. In MDD, a MetaModel defines the domain concepts and relationships in a specific domain in order to model partial reality. A Model defines a concrete system conform to a Metamodel. Then, from these models it is possible to generate totally or partially the application code by model-to-text transformations [13]. 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, MDD is proposed to tackle the technological heterogeneity underlying IoT systems by increasing the *abstraction level* where the software is implemented, focusing on the domain concepts and their relationships.

The main contributions of this paper include:

- Evidence that 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 Model-Driven solution for researchers and practitioners that allows them to design and simulate IoT systems from a high abstraction level.
- A simulator from which gain knowledge about the IoT system and its services, such as QoS-related parameters (delay, packet loss, jitter variation, SLAs compliance, etc.).
- The application of *SimulateIoT* to one case study focused on the Internet of Vehicles (IoV).

The rest of the paper is structured as follows. In Sect. 2, we present *SimulateIoT*, including the *SimulateIoT* metamodel, the graphical editor and the model-to-text transformation developed. In Sect. 3 an IoV case study is presented. Section 4 outlines the future works. Finally, Sect. 5 concludes the paper.

2 SimulateIoT Overview

SimulateIoT [1,2] is a model-driven approach to design, generate code and execute IoT simulations. The main components of SimulateIoT are: a) The Abstract

Syntax or Metamodel; b) The Concrete Syntax or Graphical editor; and c) The Model-to-Text Transformations.

A) SimulateIoT Metamodel: In the context of Model-Driven Development, a MetaModel defines the concepts and relationships in a specific domain in order to model partially reality [12]. Later, Models conform to the MetaModel could be defined and they could be used to generate total or partially the application code. The software code could be generated for a specific technological platform, improving its technological independence and decreasing the error proneness.

   The SimulateIoT metamodel (available in [3]) includes concepts related to sensors, actuators, databases, fog and cloud nodes, synthetic data generation, communication protocols, stream processing, and deploying strategies (such as deployment on Fiware platform), among others.

B) SimulateIoT Graphical Concrete Syntax and Validator: In order to facilitate modeling IoT environments, a Graphical Concrete Syntax (Graphical editor) has been generated using the Eugenia tool [6]. 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) [10] based) can be validated against the defined metamodel (EMF and OCL based). Figure 1B shows an example of model defined by using the Graphical Concrete Syntax generated.

C) SimulateIoT Model to Text Transformations: Once the models have been defined and validated conforming to the *SimulateIoT metamodel*, a model-to-text transformation defined using Acceleo [9] can generate several artefacts. Thus, the generated software includes, MQTT messaging broker (based on MQTT protocol [8]), device infrastructure, databases, a graphical analysis platform, a stream processor engine, docker container specification, configuration files for each component, a deploy script, etc.

   Figure 1A shows the deployment of the architecture of a generic IoT environment where the above mentioned artefacts and their interactions can be observed. Note that the deployment of the architecture is carried out by running the deployment script that is generated by the model-to-text transformations (script that includes all the configurations defined in the previous system modeling).

3 Case Study. Internet of Vehicles

IoV is an emerging area where delay plays a key role [15]. This is because some critical services, such as those focused on passenger safety, are delay-sensitive services that require specific QoS and SLAs [15]. With the aim of verifying whether the services comply with the specified QoS and SLAs before being deployed in

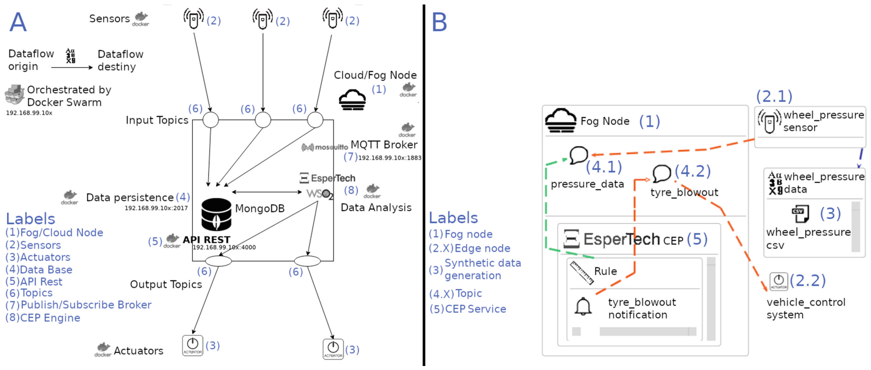


Fig. 1. A) Example of deploy diagram. B) Case study. A delay-sensitive IoV model. Note that Figure B has been developed with SimulateIoT's concrete syntax, however, the size of the name of each element has been increased to provide a better readability.

production, the system can be simulated. Thus, in this section, an Internet of Vehicles delay-sensitive system is modeled and simulated by using SimulateIoT tools.

3.1 Case Study. Model Definition

This model defines an IoV system in which several IoT devices are integrated into a vehicle and cooperate to assist the driver in the event of a tyre blowout. In this sense, a wheel pressure sensor (Fig. 1B, label 2.1) is integrated in each vehicle's wheel, which monitors the pressure of the wheels in real time. These sensors publish the wheel pressure data (Fig. 1B, label 3) in a Topic (Fig. 1B, label 4.1) deployed by a Fog node (Fig. 1B, label 1). A Complex event processing (CEP) (Fig. 1B, label 5) service deployed on this Fog node analyses the data and, in case of a tyre blowout detection, notifies the event (Fig. 1B, label 4.2) to the Actuator (Fig. 1B, label 2.2) in charge of assisting the driver.

3.2 Case Study. Code Generation and Deployment

Once the model has been defined, the model-to-text transformation is applied with the following goals: i) to generate code (Java, Python, Node, etc.) 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; Later on, the systems can be deployed using specific scripts generated ad-hoc to improve the user productivity. iv) to generate for each `ProcessNode` and `EdgeNode` a *Docker* container which can be deployed throughout a network of nodes using *Docker Swarm*;

Thus, executing the simulation modelled and later on deploying it, makes it possible to analyse the final IoT system before it is implemented and deployed.

4 Future Work

With the aim of increasing the scope and usability of SimulateIoT, some additional concepts have been identified in the literature and will be included to SimulateIoT in future works:

- Mobile nodes. Node mobility is one of the key concepts in many IoT systems. Therefore, giving SimulateIoT the ability to simulate node movement would allow many users to observe, analyse and optimise the behaviour of their mobile nodes by simulating them. This future work is particularly interesting for service-oriented computing as mobility directly affects the specified QoS and SLAs, since, for instance, a gateway switch could be a critical event where the specified QoS and SLAs may not be met.
- Task scheduling. Task scheduling is a concept that has gained relevance in the IoT area due to its potential to increase the QoS of IoT systems. Including this concept in SimulateIoT is interesting as users could test their task scheduling architectures or algorithms. Thus, being able to test if they are effective, in which situations they are more or less effective, how much they improve QoS with respect to other proposals, etc. Note that this future work is of special interest for service-oriented computing, as task scheduling techniques are aimed at optimising the QoS of the services deployed in an IoT system.

5 Conclusions

The Model-driven development (MDD) approach proposed in this paper, SimulateIoT, shows that MDD techniques are a suitable way to tackle the complexity of domains where heterogeneous technologies are integrated. Besides, SimulateIoT helps users to design, generate, deploy, analyse, and optimise their IoT systems, streamlining the process of IoT systems development and saving costs. Especially, in those IoT systems involving critical services that cannot be deployed in production until it has been verified that they comply with the specified QoS and SLAs.

Acknowledgment. This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF); and Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

Cofinanciado por la Unión Europea JUNTA DE EXTREMADURA Consejería de Economía, Ciencia y Agenda Digital

References

1. Barriga, J.A., Clemente, P.J., Hernández, J., Pérez-Toledano, M.A.: SimulateIoT-FIWARE: domain specific language to design, code generation and execute IoT simulation environments on FIWARE. IEEE Access **10**, 7800–7822 (2022). https://doi.org/10.1109/ACCESS.2022.3142894
2. Barriga, J.A., Clemente, P.J., Sosa-Sánchez, E., Prieto, E.: SimulateIoT: domain specific language to design, code generation and execute IoT simulation environments. IEEE Access **9**, 92531–92552 (2021)
3. Corchero, J.A.B., Clemente, P.J.: SimulateIoT metamodel. Mendeley Data, v1 (2022). https://doi.org/10.17632/4mmgv82k2c.1
4. Hosseinioun, P., Kheirabadi, M., Kamel Tabbakh, S.R., Ghaemi, R.: aTask scheduling approaches in fog computing: a survey. Trans. Emerg. Telecommun. Technol. **33**(3), e3792 (2022). https://doi.org/10.1002/ett.3792, https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3792. e3792 ETT-19-0285.R1
5. Jha, D.N., et al.: IoTSim-edge: a simulation framework for modeling the behavior of internet of things and edge computing environments. Softw. Pract. Experience **50**(6), 844–867 (2020)
6. Kolovos, D.S., García-Domínguez, A., Rose, L.M., Paige, R.F.: Eugenia: towards disciplined and automated development of GMF-based graphical model editors. Softw. Syst. Model. **16**(1), 229–255 (2015). https://doi.org/10.1007/s10270-015-0455-3
7. Mehdi, K., Lounis, M., Bounceur, A., Kechadi, T.: Cupcarbon: a multi-agent and discrete event wireless sensor network design and simulation tool. In: 7th International ICST Conference on Simulation Tools and Techniques. Lisbon, Portugal, 17–19 March 2014, Institute for Computer Science, Social Informatics and Telecommunications Engineering (ICST), pp. 126–131 (2014)
8. Oasis: Message queuing telemetry transport (MQTT) v5.0 Oasis Standard (2019). https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. Accessed 13 June 2022
9. Obeo: Acceleo project (2012). http://www.acceleo.org. Accessed 13 June 2022
10. OMG: OMG Object Constraint Language (OCL), Version 2.3.1 (2012). http://www.omg.org/spec/OCL/2.3.1/. Accessed 13 June 2022
11. Sehgal, A.: Using the Contiki Cooja simulator. Jacobs University Bremen Campus Ring, Technical report, Computer Science (2013)
12. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
13. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
14. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks And Systems & Workshops, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), p. 60. (2008)
15. Xu, W., et al.: Internet of vehicles in big data era. IEEE/CAA J. Autom. Sinica **5**(1), 19–35 (2018). https://doi.org/10.1109/JAS.2017.7510736

Appendix E

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

Authors: José A. Barriga, Pedro J. Clemente, Juan Hernández and Miguel A. Pérez-Toledano
Title: SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE
Year: 2023
Conference: Jornadas de Ingeniería del Software y Bases de Datos (JISBD)
Nature: National

SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE*

José A. Barriga[0000−0001−8377−1860], Pedro J. Clemente[0000−0001−5795−6343], Juan Hernández[0000−0002−6343−7395], and Miguel A. Pérez-Toledano[0000−0002−9417−9974]

Quercus Software Engineering Group. http://quercusseg.unex.es. Department of Computer Science. University of Extremadura, Av. Universidad s/n, 10003, Cáceres (Spain)
{jose, pjclemente, juanher, toledano}@unex.es

Keywords: Model-driven development, Internet of Things, IoT simulation, Services-oriented, FIWARE
Published in: IEEE Access, Vol. 10, pp. 7800 - 7822, 2022
Impact Factor: JCR 3.48 - Q2 - Position: 79/164 - Area: Computer Science / Information Systems
DOI: `https://doi.org/10.1109/ACCESS.2022.3142894`

Abstract. Given the complexity of IoT systems, several IoT platforms have emerged to optimise their development. FIWARE, an open-source platform fostered by the European Union, makes IoT simpler by means of driving key standards for breaking the information silos, transforming Big Data into knowledge, enabling data economy and ensuring data sovereignty. On the other hand, tools such as SimulateIoT, an IoT simulator, address the complexity of IoT systems by increasing the level of abstraction from which they are approached through the application of Model-Driven development. In this communication, with the aim of further optimising the development of IoT systems, SimulateIoT has been extended towards FIWARE. The resulting tool, SimulateIoT-FIWARE, tackles the complexity of IoT systems by using FIWARE together with Model-Driven development, thus simplifying and optimising several stages involved in the IoT system development life-cycle, such as their design, testing (simulations), code generation, deployment, etc. In addition, two case studies focused on a smart building and an agricultural IoT system are presented to show the applicability of the tool.

* This work was supported by the Ministerio de Ciencia e Innovación (MCI) through the Agencia Estatal de Investigación (AEI) under Project RTI2018-098652-B-I00; by the Government of Extremadura, Council for Economy, Science and Digital Agenda, under Grant GR21133 and Grant IB20058; by the European Regional Development Fund (ERDF); and by the Cátedra Telefónica de la Universidad de Extremadura (Red de Cátedras Telefónica).

Appendix F

SimulateIoT- Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations

Authors: José A. Barriga, Pedro J. Clemente
Title: SimulateIoT-Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations (Poster)
Year: 2022
Conference: International Workshop on MDE for Smart IoT Systems (MeSS)
Nature: International

SimulateIoT-Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations (Poster)

José A. Barriga[1], Pedro J. Clemente[1]

[1] *Quercus Software Engineering Group. http://quercusseg.unex.es. Department of Computer and Telematic Systems Engineering. University of Extremadura, Av. Universidad s/n, 10003, Cáceres (Spain)*

The Internet of Things (IoT) is being applied to areas such as smart-cities, home environment, agriculture, industry, etc. These application areas are very different from each other, thus requiring IoT systems with specific performance in terms of quality of service (QoS), delay, bandwidth or energy consumption. For instance, new IoT paradigms such as the Internet of Vehicles (IoV), or classic IoT systems such as healthcare, are latency sensitive application areas that need ultra-low latency infrastructure to make the application of IoT feasible. On the other hand, applications such as video analytics, or massively multiplayer online gaming involves high bandwidth requirements and an efficient management of the network [1]. In this context Cloud Computing is the common Computing paradigm applied, however it could be a bottleneck and a single point of failure. As part of the solution to these challenges and issues, fog computing has taken on a major role. Fog computing is defined by the OpenFog Consortium as "a horizontal system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum". As a layer located between the Cloud layer and the Edge layer, it is closer to the end-devices than the Cloud, thus reducing latency, increasing bandwidth, enabling greater energy savings, better management of network load balancing, in short, offering greater QoS at an affordable cost [2].

However, the IoT is constantly evolving. According to the International Data Corporation, by 2025 the number of devices connected to the Internet will be around 42 billion, and a total of 80 zettabytes of data will be generated in the same year. The rapid growth of internet-connected things, and thus the increase in data generated, brings new opportunities but also new challenges (e.g. IoV). Therefore, even though Fog computing has helped a number of organisations and corporations to meet their IoT goals, further progress is needed in the development of infrastructures capable of meeting these new challenges. In this sense, both

MeSS 2022: International Workshop on MDE for Smart IoT Systems, Nantes, France, July, 2022

[*] Corresponding author: José A. Barriga

[†] This work was funded by the Government of Extremadura, Council for Economy, Science and Digital Agenda under the grant GR21133 and the project IB20058 and by the European Regional Development Fund (ERDF). These authors contributed equally.

jose@unex.es (J. A. Barriga); pjclemente@unex.es (P. J. Clemente)
0000-0001-8377-1860 (J. A. Barriga); 0000-0001-5795-6343 (P. J. Clemente)

corporations and academia are focusing their efforts on the development of new computing paradigms, such as Edge-Cloud computing, Cloudlet computing, Mobile Cloud Computing or Mobile Ad-hoc Cloud computing [1]. These efforts are also being focused on the improvement of existing computing paradigms, such as Fog or Cloud computing (e.g. Fog Federations, Fog-Cloud federations, task scheduling or offloading algorithms and policies improvements). To do this, IoT systems need to be developed, deployed and tested, requiring high investments on devices, fog nodes, cloud nodes, analytic nodes, hardware and software. However, in order to decrease the cost associated with developing and testing 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 optimise 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 [3]. 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 engineering 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 respect, SimulateIoT [3] is a model-driven engineering approach to define, generate code and deploy IoT systems simulations. In this paper, SimulateIoT has been extended taking into account the requirements and new challenges of current IoT systems.

In this sense, the first contribution is based on the addition of the federated Fog concepts to the IoT domain and SimulateIoT metamodel. The federation of Fogs allows the different fog nodes to act as one entity rather than as isolated nodes. In this way, the user has the possibility to analyse the impact (usually on delay) of the application of new task scheduling or offloading algorithms and policies, using geographic distributions of Fog nodes, the addition or subtraction of certain nodes, etc.

The second contribution is based on the concept of Fog-Cloud federation. IoT systems are heterogeneous in infrastructure as a response to the heterogeneity (requirements) of their tasks and processes. In this respect, the cooperation between the different layers of an IoT system is essential to optimise the system, and a current research area. For instance, there are tasks that may be computationally complex and also have latency requirements in some parts of their processes, or applications that generate several kinds of tasks, such as delay sensitive tasks and complex computational tasks (e.g. a stream processing application). In order to achieve optimal execution of such tasks, federation between the Cloud layer and the Fog layer is a key element. In this way, the Fog layer should carry out the latency-sensitive processes, and the Cloud layer should carry out the computationally complex ones. In this sense, and as in the first contribution, end-users will be able to test the impact of algorithms and policies that manage the orchestration between Fog and Cloud in terms of performance in the execution of this kind of tasks.

The third contribution is carried out as a complement to the previous ones. The possibility of modelling IoT applications is added. IoT applications are the ones that generate different tasks and processes (with different requirements), thus making use of the new infrastructure included and allowing end-users to test their task scheduling algorithms, offloading policies, etc.

The last contribution focuses on the need to create a feasible latency model for the end-user of the simulator. SimulateIoT allows the Cloud and Fog nodes to be deployed on different machines, thus emulating a real system, otherwise the simulation results would not be realistic in terms of delay. To this end, we have included the possibility to model the latency that each Edge node or IoT application would hypothetically experience when interacting with the Fog/Cloud layers. In this way, the end-user can model the maximum and minimum latency, as well as the latency distribution (e.g. Gaussian) that each of the nodes might experience when interacting with each other.

In short, these extensions allow the modelling of a federated Fog and Cloud layer that can support critical applications with critical requirements for QoS, latency (e.g. ultra-low latency), bandwidth, energy consumption etc. Thus, end-users of the simulator can design, test, analyse and optimise IoT systems according to current and future IoT scenarios in terms of infrastructure and services requirements.

Keywords: IoT IoT systems simulation Model-driven Engineering Fog federation Fog-Cloud federation

References

[1] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, J. P. Jue, All one needs to know about fog computing and related edge computing paradigms: A complete survey, Journal of Systems Architecture 98 (2019) 289–330, ISSN 1383-7621, doi:\let\@tempa\bibinfo@X@doihttps://doi.org/10.1016/j.sysarc.2019.02.009, URL https://www.sciencedirect.com/science/article/pii/S1383762118306349.

[2] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, 13–16, 2012.

[3] J. A. Barriga, P. J. Clemente, E. Sosa-Sánchez, A. E. Prieto, SimulateIoT: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments, IEEE Access 9 (2021) 92531–92552, doi:\let\@tempa\bibinfo@X@doi10.1109/ACCESS.2021.3092528.

Bibliography

[1] A. Hevner, S. Chatterjee, A. Hevner, and S. Chatterjee, "Design science research in information systems," *Design research in information systems: theory and practice*, pp. 9–22, 2010.

[2] J. A. Barriga, P. J. Clemente, E. Sosa-Sánchez, and Á. E. Prieto, "SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments," *IEEE Access*, vol. 9, pp. 92531–92552, 2021.

[3] J. A. Barriga, P. J. Clemente, J. Hernández, and M. A. Pérez-Toledano, "SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE," *IEEE Access*, vol. 10, pp. 7800–7822, 2022.

[4] J. A. Barriga, P. J. Clemente, M. A. Pérez-Toledano, E. Jurado-Málaga, and J. Hernández, "Design, code generation and simulation of IoT environments with mobility devices by using model-driven development: SimulateIoT-Mobile," *Pervasive and Mobile Computing*, vol. 89, p. 101751, 2023.

[5] J. A. Barriga, J. M. Chaves-González, A. Barriga, P. Alonso, and P. J. Clemente, "Simulate IoT Towards the Cloud-to-Thing Continuum Paradigm for Task Scheduling Assessments," 2023.

[6] S. Madakam, V. Lake, V. Lake, V. Lake, *et al.*, "Internet of Things (IoT): A literature review," *Journal of Computer and Communications*, vol. 3, no. 05, p. 164, 2015.

[7] A. A. Laghari, K. Wu, R. A. Laghari, M. Ali, and A. A. Khan, "A review and state of art of Internet of Things (IoT)," *Archives of Computational Methods in Engineering*, pp. 1–19, 2021.

[8] W. Li, T. Yigitcanlar, I. Erol, and A. Liu, "Motivations, barriers and risks of smart home adoption: From systematic literature review to conceptual framework," *Energy Research & Social Science*, vol. 80, p. 102211, 2021.

[9] G. Halegoua, *Smart cities.* MIT press, 2020.

[10] S. Razdan and S. Sharma, "Internet of medical things (IoMT): Overview, emerging technologies, and case studies," *IETE technical review*, vol. 39, no. 4, pp. 775–788, 2022.

[11] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial internet of things (IIoT): An analysis framework," *Computers in industry*, vol. 101, pp. 1–12, 2018.

[12] M. S. Farooq, S. Riaz, A. Abid, T. Umer, and Y. B. Zikria, "Role of IoT technology in agriculture: A systematic literature review," *Electronics*, vol. 9, no. 2, p. 319, 2020.

[13] Y. Kabalci, E. Kabalci, S. Padmanaban, J. B. Holm-Nielsen, and F. Blaabjerg, "Internet of things applications as energy internet in smart grids and smart environments," *Electronics*, vol. 8, no. 9, p. 972, 2019.

[14] K. N. Qureshi, S. Din, G. Jeon, and F. Piccialli, "Internet of vehicles: Key technologies, network model, solutions and challenges with future aspects," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 3, pp. 1777–1786, 2020.

[15] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of things (iot): Research, simulators, and testbeds," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, 2017.

[16] G. D'Angelo, S. Ferretti, and V. Ghini, "Simulation of the Internet of Things," in *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 1–8, IEEE, 2016.

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

[18] S. N. Swamy and S. R. Kota, "An Empirical Study on System Level Aspects of Internet of Things (IoT)," *IEEE Access*, vol. 8, pp. 188082–188134, 2020.

[19] S. Ahdan, E. R. Susanto, and N. R. Syambas, "Proposed Design and Modeling of Smart Energy Dashboard System by Implementing IoT (Internet of Things) Based on Mobile Devices," in *2019 IEEE 13th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, pp. 194–199, IEEE, 2019.

[20] Z. A. Khan, I. A. Aziz, N. A. B. Osman, and I. Ullah, "A Review on Task Scheduling Techniques in Cloud and Fog Computing: Taxonomy, Tools, Open Issues, Challenges, and Future Directions," *IEEE Access*, vol. 11, pp. 143417–143445, 2023.

[21] FIWARE Foundation, "FIWARE Catalogue." `https://www.fiware.org/catalogue/`, 2024. Accessed: 2024-01-16.

[22] FIWARE Foundation, "FIWARE: Open Source Platform for Our Smart Digital Future." `https://www.fiware.org/`, 2024. Accessed: 2024-01-16.

[23] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, vol. 19, pp. 5–13, 2020.

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

[25] OMG, "Meta object facility (mof) core specification," 2014.

[26] D. Steinberg, F. Budinsky, and M. Paternostro, "Merks Ed (2008) EMF: eclipse modeling framework 2.0," 2008.

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

[28] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[29] R. Hassan, F. Qamar, M. K. Hasan, A. H. M. Aman, and A. S. Ahmed, "Internet of Things and its applications: A comprehensive survey," *Symmetry*, vol. 12, no. 10, p. 1674, 2020.

[30] S. Ketu and P. K. Mishra, "Cloud, fog and mist computing in IoT: an indication of emerging opportunities," *IETE Technical Review*, vol. 39, no. 3, pp. 713–724, 2022.

[31] J. Han, A. J. Chung, M. K. Sinha, M. Harishankar, S. Pan, H. Y. Noh, P. Zhang, and P. Tague, "Do You Feel What I Hear? Enabling Autonomous IoT Device Pairing Using Different Sensor Types," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 836–852, 2018.

[32] A. K. Sikder, G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac, "A survey on sensor-based threats to internet-of-things (iot) devices and applications," *arXiv preprint arXiv:1802.02041*, 2018.

[33] M. Nazari Jahantigh, A. Masoud Rahmani, N. Jafari Navimirour, and A. Rezaee, "Integration of internet of things and cloud computing: a systematic survey," *IET Communications*, vol. 14, no. 2, pp. 165–176, 2020.

[34] M. M. Sadeeq, N. M. Abdulkareem, S. R. Zeebaree, D. M. Ahmed, A. S. Sami, and R. R. Zebari, "IoT and Cloud computing issues, challenges and opportunities: A review," *Qubahan Academic Journal*, vol. 1, no. 2, pp. 1–7, 2021.

[35] A. Razzaq, "A systematic review on software architectures for iot systems and future direction to the adoption of microservices architecture," *SN Computer Science*, vol. 1, no. 6, p. 350, 2020.

[36] S. Dilek, K. Irgan, M. Guzel, S. Ozdemir, S. Baydere, and C. Charnsripinyo, "QoS-aware IoT networks and protocols: A comprehensive survey," *International Journal of Communication Systems*, vol. 35, no. 10, p. e5156, 2022.

[37] J. L. Herrera, P. Bellavista, L. Foschini, J. Galán-Jiménez, J. M. Murillo, and J. Berrocal, "Meeting stringent qos requirements in iiot-based scenarios," in *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pp. 1–6, IEEE, 2020.

[38] S. Abbasi, A. M. Rahmani, A. Balador, and A. Sahafi, "Internet of Vehicles: Architecture, services, and applications," *International Journal of Communication Systems*, vol. 34, no. 10, p. e4793, 2021.

[39] Y. Zhai, W. Sun, J. Wu, L. Zhu, J. Shen, X. Du, and M. Guizani, "An Energy Aware Offloading Scheme for Interdependent Applications in Software-Defined IoV With Fog Computing Architecture," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 6, pp. 3813–3823, 2021.

[40] F. E. F. Samann, S. R. Zeebaree, and S. Askar, "IoT provisioning QoS based on cloud and fog computing," *Journal of Applied Science and Technology Trends*, vol. 2, no. 01, pp. 29–40, 2021.

[41] Google Cloud, "Google Cloud Locations." `https://cloud.google.com/about/locations?hl=es`, 2024. Accessed: [2024].

[42] I. Stojmenovic, "Fog computing: A cloud to the ground support for smart things and machine-to-machine networks," in *2014 Australasian telecommunication networks and applications conference (ATNAC)*, pp. 117–122, IEEE, 2014.

[43] Cisco, "IoT: From Cloud to Fog Computing." `https://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing`, 2024. Accessed: 2024.

[44] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing," in *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 325–329, IEEE, 2014.

[45] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet of things journal*, vol. 3, no. 6, pp. 854–864, 2016.

[46] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[47] T. Qiu, J. Chi, X. Zhou, Z. Ning, M. Atiquzzaman, and D. O. Wu, "Edge computing in industrial internet of things: Architecture, advances and challenges," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2462–2488, 2020.

[48] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE access*, vol. 6, pp. 6900–6919, 2017.

[49] H. Elazhary, "Internet of Things (IoT), mobile cloud, cloudlet, mobile IoT, IoT cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions," *Journal of Network and Computer Applications*, vol. 128, pp. 105–140, 2019.

[50] J. E. Luzuriaga, J. C. Cano, C. Calafate, P. Manzoni, M. Perez, and P. Boronat, "Handling mobility in IoT applications using the MQTT protocol," in *2015 Internet Technologies and Applications (ITA)*, pp. 245–250, 2015.

[51] H. Hayashi, T. Sasatani, Y. Narusue, and Y. Kawahara in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall), title=Design of Wireless Power Transfer Systems for Personal Mobility Devices in City Spaces, year=2019, volume=, number=, pages=1-5, doi=10.1109/VTCFall.2019.8891268*.

[52] S. Ramnath, A. Javali, B. Narang, P. Mishra, and S. K. Routray, "IoT based localization and tracking," in *2017 International Conference on IoT and Application (ICIOT)*, pp. 1–4, IEEE, 2017.

[53] V. Sharma, I. You, K. Andersson, F. Palmieri, M. H. Rehmani, and J. Lim, "Security, Privacy and Trust for Smart Mobile- Internet of Things (M-IoT): A Survey," *IEEE Access*, vol. 8, pp. 167123–167163, 2020.

[54] J. Ding, M. Nemati, C. Ranaweera, and J. Choi, “IoT Connectivity Technologies and Applications: A Survey,” *IEEE Access*, vol. 8, pp. 67646–67673, 2020.

[55] S. M. Ghaleb, S. Subramaniam, Z. A. Zukarnain, and A. Muhammed, “Mobility management for IoT: a survey,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2016, pp. 1–25, 2016.

[56] I. Ali, S. Sabir, and Z. Ullah, “Internet of things security, device authentication and access control: a review,” *arXiv preprint arXiv:1901.07309*, 2019.

[57] K. Bierzynski, A. Escobar, and M. Eberl, “Cloud, fog and edge: Cooperation for the future?,” in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 62–67, IEEE, 2017.

[58] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, “The internet of things, fog and cloud continuum: Integration and challenges,” *Internet of Things*, vol. 3, pp. 134–155, 2018.

[59] X. Wei, C. Tang, J. Fan, and S. Subramaniam, “Joint Optimization of Energy Consumption and Delay in Cloud-to-Thing Continuum,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2325–2337, 2019.

[60] M. R. Alizadeh, V. Khajehvand, A. M. Rahmani, and E. Akbari, “Task scheduling approaches in fog computing: A systematic review,” *International Journal of Communication Systems*, vol. 33, no. 16, p. e4583, 2020.

[61] M. Z. Hasan and H. Al-Rizzo, “Task scheduling in Internet of Things cloud environment using a robust particle swarm optimization,” *Concurrency and Computation: Practice and Experience*, vol. 32, no. 2, p. e5442, 2020.

[62] M. M. Sandhu, S. Khalifa, R. Jurdak, and M. Portmann, “Task scheduling for energy-harvesting-based IoT: A survey and critical analysis,” *IEEE Internet of Things Journal*, vol. 8, no. 18, pp. 13825–13848, 2021.

[63] X.-Q. Pham, N. D. Man, N. D. T. Tri, N. Q. Thai, and E.-N. Huh, “A cost-and performance-effective approach for task scheduling based on collaboration between cloud and fog computing,” *International Journal of Distributed Sensor Networks*, vol. 13, no. 11, p. 1550147717742073, 2017.

[64] X. Cai, S. Geng, D. Wu, J. Cai, and J. Chen, “A Multicloud-Model-Based Many-Objective Intelligent Algorithm for Efficient Task Scheduling in Internet of Things,” *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 9645–9653, 2021.

[65] Azure, “Azure IoT Hub.” `https://azure.microsoft.com/es-es/products/iot-hub`, 2024. Accessed: 2024-01-16.

[66] ThingSpeak, “ThingSpeak - IoT Analytics.” `https://thingspeak.com/`, 2024. Accessed: 2024-01-16.

[67] ThingWorx, “ThingWorx Industrial IoT Platform.” `https://www.ptc.com/en/products/thingworx`, 2024. Accessed: 2024-01-16.

[68] The Things Network, “The Things Network,” 2024. Accessed: 2024-02-14.

[69] L. Babun, K. Denney, Z. B. Celik, P. McDaniel, and A. S. Uluagac, “A survey on IoT platforms: Communication, security, and privacy perspectives,” *Computer Networks*, vol. 192, p. 108040, 2021.

[70] J. Conde, A. Munoz-Arcentales, Alonso, S. López-Pernas, and J. Salvachúa, “Modeling Digital Twin Data and Architecture: A Building Guide With FIWARE as Enabling Technology,” *IEEE Internet Computing*, vol. 26, no. 3, pp. 7–14, 2022.

[71] J. A. Barriga and P. J. Clemente, “SimulateIoT: A model-driven approach to simulate IoT systems,” *Predoctoral en Ingenieria Informática*, p. 29, 2022.

[72] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 1–62, 2012.

[73] R. A. Light, “Mosquitto: server and client implementation of the MQTT protocol,” *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[74] Open Mobile Alliance, *NGSI Context Management.* Open Mobile Alliance, May 2012.

[75] F. Cirillo, G. Solmaz, E. L. Berz, M. Bauer, B. Cheng, and E. Kovacs, “A standard-based open source IoT platform: FIWARE,” *IEEE Internet of Things Magazine*, vol. 2, no. 3, pp. 12–18, 2019.

[76] J. Haxhibeqiri, E. De Poorter, I. Moerman, and J. Hoebeke, “A survey of LoRaWAN for IoT: From technology to application,” *Sensors*, vol. 18, no. 11, p. 3995, 2018.

[77] A. Lavric, A. I. Petrariu, and V. Popa, “Sigfox communication protocol: The new era of iot?,” in *2019 international conference on sensing and instrumentation in IoT Era (ISSI)*, pp. 1–4, IEEE, 2019.

[78] C. S. Lai, Y. Jia, Z. Dong, D. Wang, Y. Tao, Q. H. Lai, R. T. Wong, A. F. Zobaa, R. Wu, and L. L. Lai, “A review of technical standards for smart cities,” *Clean Technologies*, vol. 2, no. 3, pp. 290–310, 2020.

[79] C. Stolojescu-Crisan, C. Crisan, and B.-P. Butunoi, “An IoT-based smart home automation system,” *Sensors*, vol. 21, no. 11, p. 3784, 2021.

[80] J. Al Dakheel, C. Del Pero, N. Aste, and F. Leonforte, “Smart buildings features and key performance indicators: A review,” *Sustainable Cities and Society*, vol. 61, p. 102328, 2020.

[81] V. K. Quy, N. V. Hau, D. V. Anh, N. M. Quy, N. T. Ban, S. Lanza, G. Randazzo, and A. Muzirafuti, “IoT-enabled smart agriculture: architecture, applications, and challenges,” *Applied Sciences*, vol. 12, no. 7, p. 3396, 2022.

[82] H. Jaidka, N. Sharma, and R. Singh, “Evolution of iot to iiot: Applications & challenges,” in *Proceedings of the international conference on innovative computing & communications (ICICC)*, 2020.

[83] B. Rana, Y. Singh, and P. K. Singh, "A systematic survey on internet of things: Energy efficiency and interoperability perspective," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 8, p. e4166, 2021.

[84] J.-A. Jiang, J.-C. Wang, H.-S. Wu, C.-H. Lee, C.-Y. Chou, L.-C. Wu, and Y.-C. Yang, "A novel sensor placement strategy for an IoT-based power grid monitoring system," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7773–7782, 2020.

[85] G. Manogaran and B. S. Rawal, "An Efficient Resource Allocation Scheme With Optimal Node Placement in IoT-Fog-Cloud Architecture," *IEEE Sensors Journal*, vol. 21, no. 22, pp. 25106–25113, 2021.

[86] M. Ghobaei-Arani and A. Shahidinejad, "A cost-efficient IoT service placement approach using whale optimization algorithm in fog computing environment," *Expert Systems with Applications*, vol. 200, p. 117012, 2022.

[87] J. L. Herrera, J. Galán-Jiménez, J. Berrocal, and J. M. Murillo, "Optimizing the response time in sdn-fog environments for time-strict iot applications," *IEEE Internet of Things Journal*, vol. 8, no. 23, pp. 17172–17185, 2021.

[88] J. L. Herrera, J. Galán-Jiménez, L. Foschini, P. Bellavista, J. Berrocal, and J. M. Murillo, "QoS-aware fog node placement for intensive IoT applications in SDN-fog scenarios," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 13725–13739, 2022.

[89] A. K. M. Al-Qurabat and A. Kadhum Idrees, "Data gathering and aggregation with selective transmission technique to optimize the lifetime of Internet of Things networks," *International Journal of Communication Systems*, vol. 33, no. 11, p. e4408, 2020.

[90] Y. Liu, H.-N. Dai, Q. Wang, M. Imran, and N. Guizani, "Wireless powering Internet of Things with UAVs: Challenges and opportunities," *IEEE Network*, vol. 36, no. 2, pp. 146–152, 2022.

[91] I. Farris, L. Militano, M. Nitti, L. Atzori, and A. Iera, "MIFaaS: A mobile-IoT-federation-as-a-service model for dynamic cooperation of IoT cloud providers," *Future Generation Computer Systems*, vol. 70, pp. 126–137, 2017.

[92] N. Medhat, S. Moussa, N. Badr, and M. F. Tolba, "Testing techniques in IoT-based systems," in *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pp. 394–401, IEEE, 2019.

[93] J. Esquiagola, L. C. de Paula Costa, P. Calcina, G. Fedrecheski, and M. Zuffo, "Performance Testing of an Internet of Things Platform.," in *IoTBDS*, pp. 309–314, 2017.

[94] S. K. Datta, C. Bonnet, H. Baqa, M. Zhao, and F. Le-Gall, "Approach for Semantic Interoperability Testing in Internet of Things," in *2018 Global Internet of Things Summit (GIoTS)*, pp. 1–6, 2018.

[95] G. White, A. Palade, C. Cabrera, and S. Clarke, "IoTPredict: collaborative QoS prediction in IoT," in *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 1–10, IEEE, 2018.

[96] S. Noureddine and B. Meriem, "ML-SLA-IoT: an SLA Specification and Monitoring Framework for IoT applications," in *2021 International Conference on Information Systems and Advanced Technologies (ICISAT)*, pp. 1–12, 2021.

[97] X. Li and L. Da Xu, "A review of Internet of Things—Resource allocation," *IEEE Internet of Things Journal*, vol. 8, no. 11, pp. 8657–8666, 2020.

[98] X. Liu and N. Ansari, "Toward Green IoT: Energy Solutions and Key Challenges," *IEEE Communications Magazine*, vol. 57, no. 3, pp. 104–110, 2019.

[99] A. Sinha, D. Das, V. Udutalapally, M. K. Selvarajan, and S. P. Mohanty, "iThing: Designing Next-Generation Things with Battery Health Self-Monitoring Capabilities for Sustainable IoT in Smart Cities," *arXiv preprint arXiv:2106.06678*, 2021.

[100] Y. Ramzanpoor, M. Hosseini Shirvani, and M. Golsorkhtabaramiri, "Multi-objective fault-tolerant optimization algorithm for deployment of IoT applications on fog computing infrastructure," *Complex & Intelligent Systems*, vol. 8, no. 1, pp. 361–392, 2022.

[101] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, and C.-S. Shih, "Supporting Service Adaptation in Fault Tolerant Internet of Things," in *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 65–72, 2015.

[102] B. Pourghebleh and V. Hayyolalam, "A comprehensive and systematic review of the load balancing mechanisms in the Internet of Things," *Cluster Computing*, vol. 23, pp. 641–661, 2020.

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

[104] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture*, vol. 72, pp. 93–107, 2017.

[105] J. P. Dias, F. Couto, A. C. Paiva, and H. S. Ferreira, "A brief overview of existing tools for testing the internet-of-things," in *2018 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pp. 104–109, IEEE, 2018.

[106] M.-W. Tian, S.-R. Yan, W. Guo, A. Mohammadzadeh, and E. Ghaderpour, "A New Task Scheduling Approach for Energy Conservation in Internet of Things," *Energies*, vol. 16, no. 5, p. 2394, 2023.

[107] M. R. Raju and S. K. Mothku, "Delay and energy aware task scheduling mechanism for fog-enabled IoT applications: A reinforcement learning approach," *Computer Networks*, vol. 224, p. 109603, 2023.

[108] M. S. Kumar and G. R. Karri, “Eeoa: cost and energy efficient task scheduling in a cloud-fog framework,” *Sensors*, vol. 23, no. 5, p. 2445, 2023.

[109] J. F. Nunamaker Jr, M. Chen, and T. D. Purdin, “Systems development in information systems research,” *Journal of management information systems*, vol. 7, no. 3, pp. 89–106, 1990.

[110] V. K. Vaishnavi, *Design science research methods and patterns: innovating information and communication technology.* Auerbach Publications, 2007.

[111] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren, “Action design research,” *MIS quarterly*, pp. 37–56, 2011.

[112] R. Baskerville, J. Pries-Heje, and J. Venable, “Soft design science methodology,” in *Proceedings of the 4th international conference on design science research in information systems and technology*, pp. 1–11, 2009.

[113] M. Bilandzic and J. Venable, “Towards participatory action design research: adapting action research and design science research methods for urban informatics,” *Journal of Community Informatics*, vol. 7, no. 3, 2011.

[114] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of management information systems*, pp. 45–77, 2007.

[115] J. R. Venable, J. Pries-Heje, and R. L. Baskerville, “Choosing a design science research methodology,” 2017.

[116] J. A. Barriga and P. J. Clemente, “Designing and simulating IoT environments by using a model-driven approach,” in *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pp. 1–6, IEEE, 2022.

[117] J. A. Barriga Corchero, P. J. Clemente Martín, E. Sosa Sánchez, and A. E. Prieto Ramos, “SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments,”

[118] J. A. Barriga, “Simulating IoT Systems from High-Level Abstraction Models for Quality of Service Assessment,” in *International Conference on Service-Oriented Computing*, pp. 314–319, Springer, 2022.

[119] J. A. Barriga Corchero, P. J. Clemente Martín, J. M. Hernández Núñez, and M. A. Pérez Toledano, “SimulateIoT-FIWARE: Domain Specific Language to Design, Code Generation and Execute IoT Simulation Environments on FIWARE,”

[120] F. Wu, Q. Wu, and Y. Tan, “Workflow scheduling in cloud: a survey,” *The Journal of Supercomputing*, vol. 71, no. 9, pp. 3373–3418, 2015.

[121] A. Arunarani, D. Manjula, and V. Sugumaran, “Task scheduling techniques in cloud computing: A literature survey,” *Future Generation Computer Systems*, vol. 91, pp. 407–415, 2019.

[122] J. A. Barriga and P. J. Clemente, “SimulateIoT-Federations: Domain Specific Language for designing and executing IoT simulation environments with Fog and Fog-Cloud federations (Poster),” 2022.

[123] 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,” *Software: Practice and Experience*, vol. 51, no. 5, pp. 893–920, 2021.

[124] M. Salama, Y. Elkhatib, and G. Blair, “IoTNetSim: A modelling and simulation platform for end-to-end IoT services and networking,” in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 251–261, 2019.

[125] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, “Internet of things (iot): Research, simulators, and testbeds,” *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, 2018.

[126] Eclipse Foundation, “Eclipse Packages,” 2024. Accessed: 2024-01-19.

[127] Docker, Inc., “Docker: Empowering App Development for Developers,” 2024. Accessed: 2024-01-19.

[128] MongoDB, Inc., "MongoDB: La base de datos para aplicaciones modernas," 2024. Accessed: 2024-01-19.

[129] Eclipse Foundation, "Eclipse Mosquitto." `https://mosquitto.org/`, 2024. Accessed: 2024-01-17.

[130] EsperTech Inc., "EsperTech - Complex Event Processing," 2024. Accessed: 2024-01-17.

[131] FIWARE Foundation, "Orion Context Broker Documentation," 2024. Accessed: 2024-01-19.

[132] FIWARE Foundation, "Orion Context Broker Database Administration Documentation," 2024. Accessed: 2024-01-19.

[133] FIWARE Foundation, "Perseo Front-End Documentation," 2024. Accessed: 2024-01-19.

[134] FIWARE Foundation, "IoT Agent Node.js Library Documentation," 2024. Accessed: 2024-01-19.

[135] S. H. Shah and I. Yaqoob, "A survey: Internet of things (iot) technologies, applications and challenges," *2016 IEEE Smart Energy Grid Engineering (SEGE)*, pp. 381–385, 2016.

[136] A. Pal, A. Mukherjee, and B. P, "Model-driven development for internet of things: Towards easing the concerns of application developers," in *Internet of Things. User-Centric IoT: First International Summit, IoT360 2014, Rome, Italy, October 27-28, 2014, Revised Selected Papers, Part I*, pp. 339–346, Springer, 2015.

[137] R. Dautov and H. Song, "Towards iot diversity via automated fleet management.," in *MDE4IoT/ModComp@ MoDELS*, pp. 47–54, 2019.

[138] J. Saleem, M. Hammoudeh, U. Raza, B. Adebisi, and R. Ande, "Iot standardisation: Challenges, perspectives and solution," in *Proceedings of the 2nd international conference on future networks and distributed systems*, pp. 1–9, 2018.

[139] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry, “Iot architecture challenges and issues: Lack of standardization,” in *2016 Future technologies conference (FTC)*, pp. 731–738, IEEE, 2016.

[140] G. Kecskemeti, G. Casale, D. N. Jha, J. Lyon, and R. Ranjan, “Modelling and simulation challenges in internet of things,” *IEEE cloud computing*, vol. 4, no. 1, pp. 62–69, 2017.

[141] V. Agarwal, S. Tapaswi, and P. Chanak, “A survey on path planning techniques for mobile sink in iot-enabled wireless sensor networks,” *Wireless Personal Communications*, vol. 119, no. 1, pp. 211–238, 2021.

[142] T. Bu, Z. Huang, K. Zhang, Y. Wang, H. Song, J. Zhou, Z. Ren, and S. Liu, “Task scheduling in the internet of things: challenges, solutions, and future trends,” *Cluster Computing*, vol. 27, no. 1, pp. 1017–1046, 2024.

[143] J. C. Kirchhof, L. Malcher, and B. Rumpe, “Understanding and improving model-driven iot systems through accompanying digital twins,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 197–209, 2021.

[144] M. Segovia and J. Garcia-Alfaro, “Design, modeling and implementation of digital twins,” *Sensors*, vol. 22, no. 14, p. 5396, 2022.

[145] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, “On blockchain and its integration with iot. challenges and opportunities,” *Future generation computer systems*, vol. 88, pp. 173–190, 2018.