JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

SimulateIoT 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 IoT systems, one of the most widespread techniques is the well-known task scheduling technique. In recent years, several task scheduling proposals have been developed to optimise task processing from different perspectives, such as makespan, energy, money, etc. However, the technological heterogeneity and the complex infrastructure of IoT systems could imply that the development of such proposals is a hard, error-prone and tedious process. Furthermore, the tests required during the development of these proposals have to be isolated as otherwise, they could affect the system in production. This implies an investment of money, time and effort in the acquisition of devices, their configuration, deployment, etc. To avoid this scenario, the system can be simulated, and its underlying technical complexity reduced by increasing the abstraction level from which these systems are designed. In this regard, simulators based on Model-driven development can help both to test the IoT system 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 for the assessment of task-scheduling proposals. The generated simulations follow the Cloud-to-Thing continuum paradigm including cloud, fog, edge and mist nodes (allowing their federation), 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 proposals in the simulations. Finally, a case study focused on an 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

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. vv, No. nn, yyyy.
http://dx.doi.org/10.5381/jot.yyyy.vv.nn.aa

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 as isolated nodes, including nodes that belong to different computing layers conforming Cloud-Fog-Edge heterogeneous federations (Bittencourt et al. 2018; Kar et al. 2022; Mijuskovic et al. 2021). Consequently, the Cloud-to-Thing continuum paradigm emerges (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 a federation in the context of IoT, 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). In this context, the 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 (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 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.

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 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 (MDD) 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 suitably 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:

- 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 systems based on the Cloud-to-Thing continuum where task scheduling is a key factor.
- This work includes a metamodel to model IoT systems based on the Cloud-to-Thing continuum with task scheduling capabilities. Note that it includes a Graphical Concrete Syntax.
- A Model-to-text transformation to generate code for a specific IoT platform.
- A case study has been designed and assessed 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 the proposed simulator. In Section 4 the task schedul-

ing model is defined. Next, Section 5 presents the proposed simulator taking into account the design and implementation phases including the new metamodel and the graphical editor. In section 6 the model-to-text transformation from the proposed simulator models to code is explained. In Section 7 a case study to show the applicability of the proposed 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 are reviewed those that have been considered the most interesting. 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 the 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 allows the design of DAGs (Directed Acyclic Graph) and the simulation of their processing. Note that DAGs are similar to 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. Besides, this simulator is based on a mathematical model and therefore does not deploy a real component architecture on which to simulate the task scheduling processes, which could negatively impact the trustworthiness of the simulator (Chernyshev et al. 2018).

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. Besides, note that this simulator is based on a mathematical model, which could negatively impact the trustworthiness of the simulator. Additionally, it not support current elements related with IoT systems such as edge nodes, fog node, node federations, 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. Besides, note that this simulator is based on a mathematical model, which could negatively impact the trustworthiness of the simulator.

ScSF (Rodrigo et al. 2018) is a simulation tool that focuses only on task scheduling purposes. In this way, it is not only focused on IoT systems but focuses on any system with task scheduling needs. 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. Besides, note that this simulator is based on a mathematical model, which could negatively impact the trustworthiness of the simulator.

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, there are several differences between the proposal presented in this communication with WorkflowSim and with the other simulators: 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; 2) The proposed simulator is a simulator based on the MDD, addressing the design of the simulations from a high level of abstraction. Thus, 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 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).

3. Introduction to the extended 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.

In this regard, 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 model-to-text transformations (Sendall & Kozaczynski 2003). Thus, high-level definitions (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 technological independence and decreasing error proneness.

Therefore, in order to extend SimulateIoT towards the Cloud-to-Things 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 or the element that allows to graphically design models (M1) from the Metamodel (M2) and 3) Model-to-Text Transformations (M2T), the element that carries out the code generation (M0) from models (M1).

In this regard, as indicated in Section 1, SimulateIoT does not support the deployment of task scheduling approaches in its simulations as it does not have the required components for this purpose, such as components that generate tasks, process them, etc. Besides, although SimulateIoT allows the deployment of mist, edge, fog or cloud nodes, it does not allow their federation (Cloud-to-Thing continuum infrastructure), a key aspect in systems where task scheduling is applied. Thus, all the new features added to SimulateIoT (above mentioned metamodelling layers) are focused on allowing it to model, generate and simulate a suitable IoT system where users can integrate, test and analyse their task scheduling proposals.

For this purpose, it is necessary to define, develop and integrate a task scheduling model (Section 4) into SimulateIoT. In this respect, Figure 2 shows, from a high level of abstraction, the deployment of a generic simulation generated by using the extended version of SimulateIoT, i.e. with the aforementioned task scheduling model integrated. So, in Figure 2 it is possible to differentiate the main components included as extensions in this communication (coloured in red) from the components belonging to the previous version of the simulator (coloured in blue). The main contributions added as extensions to SimulateIoT shown in Figure 2 are outlined below.

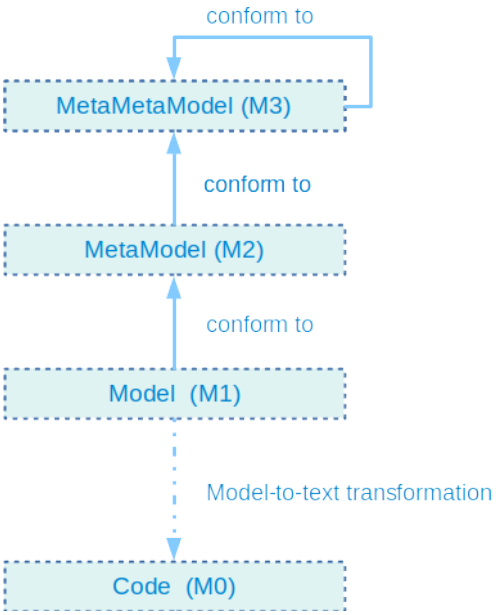

Figure 1 The four layers of metamodeling. In SimulateIoT (Barriga et al. 2021): a) M3 is Ecore, b) M2 is SimulateIoT Metamodel c) M1 is a model conforms to SimulateIoT Metamodel and d) Code is generated using the model-to-text transformations defined in SimulateIoT approach.

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. So, it is required to extend SimulateIoT to allow users to model `Tasks`, to assign each of these `Task` to the different nodes that will generate them during simulation, etc. In this regard, Figure 2 shows an exchange of these `Tasks` between the edge/mist layer and the fog layer (interaction ①), between the nodes of the fog layer, (interaction ⑥) and between the fog and the cloud layer (interaction ⑦). Note that this exchange of `Tasks` is also possible between nodes of the cloud and the mist/edge layer.

As the aforementioned `Tasks` have to be generated and offloaded to the system, SimulateIoT has been extended with two elements, the `Task Node` ② and the

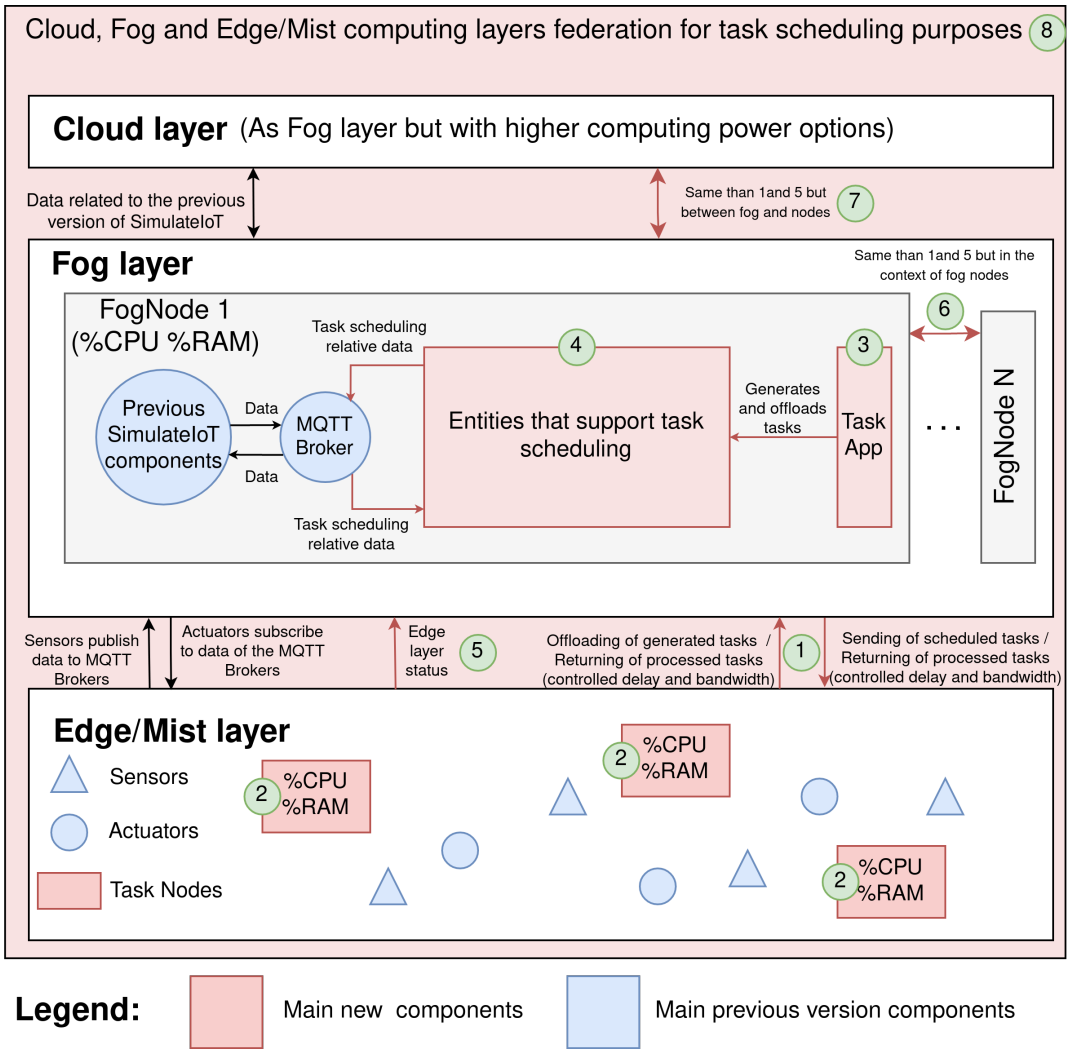

Figure 2 A generic simulation generated by using model-to-text transformation from a model defined with the proposed simulator.

`Task App` ③. The `Task Node` ② is the node belonging to the mist/edge layer which can generate and offload `Tasks`, while the `Task App` is the component of the fog/-cloud layer that can generate and offload `Tasks`.

In this sense, in a task scheduling system, these generated `Tasks` need to be scheduled and processed. So, two components have been added to SimulateIoT for this purpose. On the one hand, the scheduling is carried out in the fog or cloud layer by the `Task Scheduler`, a component that belongs to the `Entities that support task scheduling` (Figure 2, ④). On the other hand, the processing is performed by the `Task Processor`. In the fog/cloud layer, the `Task Processor` is included in the `Entities that support task scheduling`. However, `Tasks` can also be processed by mist/edge nodes, so the `Task Nodes` can also integrate a `Task Processor`.

The CPU and RAM concepts have been included in SimulateIoT regarding the processing of `Tasks`. So, fog, cloud and `Task Nodes` have a CPU and a RAM (modelled by the user), that will be used when processing `Tasks`. Note that the nodes can share their hardware status (Figure 2, ⑤, ⑥, and ⑦). Thus, the `Task Scheduler` could use this hardware status as input for scheduling purposes.

Finally, task scheduling is often applied to environments that follow the Cloud-to-Things continuum paradigm. In this kind of system, nodes can be federated (Figure 2, ⑧), acting as a single entity rather than isolated nodes. For this purpose, SimulateIoT has been extended to this concept, allowing users to model federations. Thus, any node regardless of the computing layer to which it belongs can be federated. i.e. cloud-fog-edge-mist federations are allowed.

The component that allows federations in the extended version of SimulateIoT is the `Networking Node`. In this way, this node is integrated into each node that belongs to a federation, handling networking aspects such as the links between the nodes of a federation

(simulating the delay and bandwidth between nodes). Furthermore, note that the `Networking Node` can also share the status of each link (current delay and available bandwidth) with the `Task Scheduler`. In this way, the `Task Scheduler` can use this data for scheduling purposes.

In short, the key concepts related to the Cloud-to-Things continuum and to task scheduling have been identified and included as extensions to SimulateIoT. In this way, the resulting simulator allows users to simulate suitable IoT systems where users can perform several tests and analyse their task scheduling proposals without the need for high investment in the acquisition of devices, their configuration, deployment, etc.

4. The proposed Cloud-to-Thing continuum and task scheduling model

Before extending SimulateIoT towards the Cloud-to-Things continuum and towards task scheduling, it is first necessary to identify the main concepts of these systems. The previous Section 3, provides an introduction to these concepts: `Task`, `Task App`, `Task Node`, `Networking Node`, `Task Processor` and `Task Scheduler`.

In this regard, this section addresses these concepts in detail, thus describing the envisioned Cloud-to-Thing continuum and task scheduling model. Note that, in this communication, it is this model which has been added as extensions or contributions to each of the metamodelling layers (Section 3) of SimulateIoT.

4.1. 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. Note that hereafter the terms `Task` and workflow will be used as synonyms.

In this regard, Figure 3 shows a graphical representation of a workflow. This workflow represents a `Tasks` decomposed in 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 3, `Tasks` B and C cannot be processed until `Task` A has been processed. On the other hand, `Task` D cannot be processed until `Task` B and C have been processed.

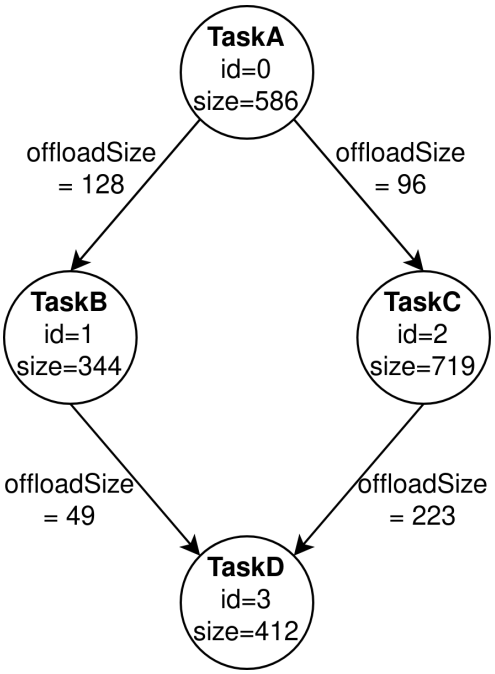

Figure 3 Graphical representation of a workflow.

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` 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`.

4.2. Task App

In task scheduling scenarios, it is common to deploy applications that provide services at the fog and cloud layers. Thus, these applications are the ones that generate and offload `Tasks` (the services they provide are decomposed in `Tasks`, i.e. workflows). In this model, these applications are included as `Task Apps`, which can be deployed in the different nodes of the fog and cloud layers.

4.3. 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 providing a better QoS than fog and cloud layers, since these devices are in the mist/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 their generated workflows.

On the other hand, as `Task Apps`, `Task Nodes` are also designed to generate and offload `Tasks` to the rest of the system. In this case, from the edge and mist layers, as could be edge or mist devices that could generate `Tasks` and require their processing.

4.4. Networking Node

Task scheduling is often applied to environments that have a Cloud-to-Things continuum infrastructure. In this kind of system, nodes can be federated, thus acting as a single entity rather than isolated nodes. In this regard, this model allows federations by means of `Links`, i.e. the connections between the different nodes that belong to a federation. In this context, it is the `Networking Node` which manages these `Links`.

In this model, these `Links` are unidirectional, consequently, two `Links` are needed to allow two components to interact with each other. In this respect, each `Networking Node` handles the `Links` whose source is the node where the `Networking Node` is integrated.

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 latency or bandwidth. However, as aspects such as delay and bandwidth among nodes are critical aspects in task scheduling systems (Jamil et al. 2022), users could require specific latency or bandwidth between nodes for simulation purposes. Note that, 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.

4.5. Task Processor node

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.

4.6. Task Scheduler node

In the context of this communication, this component is the most relevant, as 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 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 an API REST (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. The following sections (Sections 5 and 6) describe the extensions made to each SimulateIoT's metamodelling layer to extend SimulateIoT towards this Cloud-to-Things and task scheduling model.

5. 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 (M2T). This Section describes the proposed simulator 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 (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.

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 IoT systems with a Cloud-to-Thing continuum infrastructure and with task scheduling capabilities (entities and services described in Section 4).

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 12 (Appendix A) shows the complete metamodel, with the elements relating to the extension carried out also highlighted in blue.

For the sake of clarity, this section is divided into the domain-specific concepts identified in Section 4: `Task`, `Task App`, `Task Node`, `Networking Node`, `Task Processor` and `Task Scheduler`. In this way, each of these subsections includes the contributions that make it possible to model these concepts and their features (classes and relations shown in Figure 4). In this regard, note that this section does not aim to describe how these components work internally, which is addressed in Section 6, where M2T are presented.

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]` ⓧ, where x is the label associated with the `[class]` in Figure 4.

5.1.1. Task In order to model and include the `Task` concept in the simulations, several classes and relationships have been added to the metamodel of SimulateIoT. Thus, the following lists and describes the classes and relationships included in the metamodel to gather the `Task` concept.

- `Task class` ①
  Aim: The `Task class` is included to allow users to model `Tasks` in their simulations.
  Attributes: The `Task class` has three attributes, 1) `id`, to identify the `Task`, 2) `name`, to give a name to the `Task`, 3) `generation_period`, to specify the period of the `Task`, i.e. if the period is 10, the `Task` will be generated and offloaded to the system every 10 seconds.
  Relationships: The `Task class` have three composition relationships 1) with the class `Workflow` (1.1), 2) with the class `Task App` ②, and 3) with the class `Task Node` ③. For the sake of clarity, these relationships will be described later, when describing the above-mentioned classes in their sections.

- `Workflow class` (1.1)
  Aim: The aim of this class is to allow users to model `Tasks` ① by means of workflows.
  Attributes: The `Workflow class` has two attributes, `id` and `name`, which are included to identify and name respectively the instances of this class.
  Relationships: The `Workflow class` has three composition relationships, 1) with `Task`, as a `Task` has to be modelled by means of a `Workflow`, 2-3) with `Workflow_edge` (1.2) and `Workflow_Node` (1.3) classes, as a `Workflow` can be defined by a set of nodes interconnected by edges (Adhikari et al. 2019).

- `Workflow_edge class` (1.2)
  Aim: The `Workflow_edge class` is included since task scheduling workflows can be defined as a set of nodes and edges (Adhikari et al. 2019) (see Section 4.1). Thus, this class intends to allow users to model the edges of the `Workflow` class.

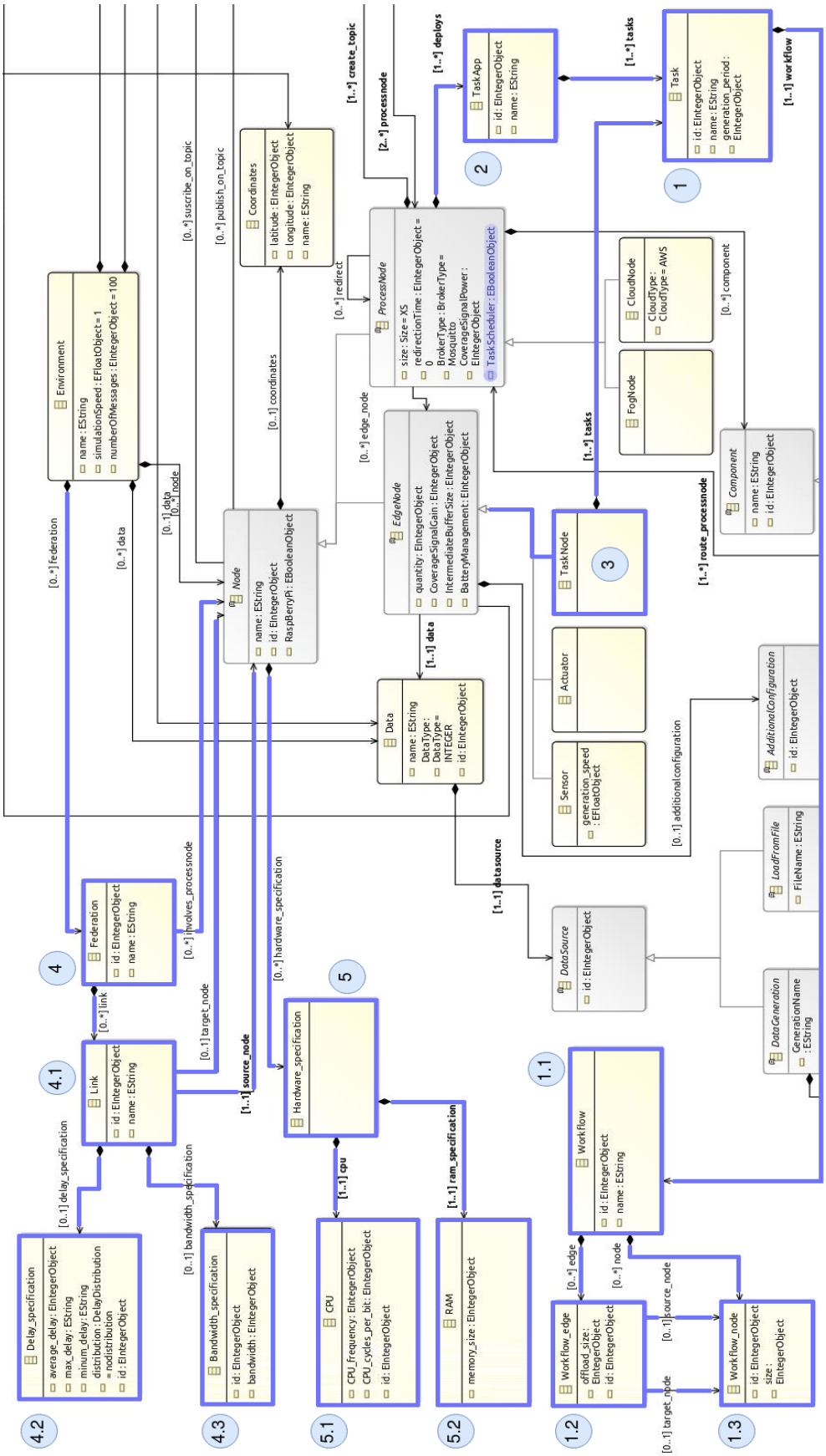

Figure 4 Excerpt of the proposed simulator Metamodel focusing on the task scheduling concepts. The complete metamodel can be found in Appendix A at Figure 12.

Attributes: The Workflow_edge class has two attributes, 1) id to identify each Workflow_edge instance and, 2) offload_size, to allow users to model the offload size of each Workflow_edge (data transmission between Workflow_nodes).
Relationships: The Workflow_edge class has three relationships, 1-2) two reference relationships with the Workflow_node class that aim to specify the source and the target Workflow_node (1.3) of the Workflow_edge, and 3) a composition relationship with the Workflow class (1.1), to allow users to assign Workflow_edges to a Workflow.

- Workflow_node class (1.3)
Aim: The Workflow_node class is included since task scheduling workflows can be defined as a set of nodes and edges (Adhikari et al. 2019). Thus, this class intends to allow users to model the nodes of a workflow.
Attributes: The Workflow_node class has two attributes, 1) id to identify each Workflow_node instance and, 2) size, to allow users to model the size of each Workflow_node (bytes).
Relationships: The Workflow_node class has three relationships, 1-2) two reference relationships with the Workflow_edge class (1.2) that have been described above, and 3) one composition relationship with the Workflow class (1.1), to allow users to assign Workflow_node to a Workflow.

5.1.2. Task App

Only one class has been considered necessary in order to include in the metamodel the concept of Task App, the TaskApp class ②. Thus, the *aim* of this class is to allow users to model the Task Apps that will be deployed on each fog or cloud node during the simulation. Regarding the *attributes* of this class, it has two attributes, id and name, to identify and name respectively each instance of Task App. As for the *relationships* of this class, it has two relationships, 1) with ProcessNode class (fog and cloud nodes) to allow users to specify on what fog or cloud node a Task App has to to be deployed, and 2) with the Task class ①, to allow users to model the Tasks that will be generated and offloaded to the system during a simulation by each Task App.

5.1.3. Task Node

Only one class has been considered necessary in order to include in the metamodel the concept of Task Node, the TaskNode class ③. Thus, the *aim* of this class is to allow users to model Task Nodes as part of the edge/mist layer of the system. This class has no *attributes*, as the attributes it needs are inherited from the EdgeNode class. In this regard, the Task Node class has two *relationships*, 1) a hierarchical relationship with the EdgeNode class (inheriting all its attributes), as the Task Node is one of the EdgeNodes that can be included in simulations, and 2) with Task class ①, to allow users to model the Tasks that will be generated and offloaded to the system during a simulation by each Task Node.

5.1.4. Networking Node

The Networking Node concept is not included in the metamodel by a specific class such as the Task, Task Node or the Task App concepts. Thus, the Networking Node is transparent to the user, who only has to be concerned about modelling the Links (4.1) to interconnect the nodes (edge, fog or cloud nodes) of a Federation ④ and their features (Delay (4.2) and Bandwidth (4.3) specifications).

Later, in M2T, all the features modelled in these classes are considered to generate the Networking Node code of each modelled node (those that belong to a Federation). Therefore, in order to model and include the Networking Node concept in the simulations, several classes and relationships have been added to the metamodel of SimulateIoT. Thus, the following lists and describes the classes and relationships included in the metamodel to gather the Networking Node concept.

- Federation class ④
Aim: The aim of this class is to allow users to model node federations. In this way, several edge, fog and cloud nodes could belong to a federation acting as a single entity rather than isolated nodes for task scheduling purposes.
Attributes: The Federation class has two attributes, id and name, to identify and name respectively each instance of Federation class.
Relationships: The Federation class has three relationships, 1) a composition relationship with Environment, the root class of the metamodel, 2) a reference relationship with Node, to allow users to model what nodes (edge, fog or cloud) belong to each federation modelled, 3) with the Link (4.1) class, which will be addressed below, in the Link class description.

- Link class (4.1)
Aim: Since in a federation nodes are interconnected, the aim of this class is to allow users to model the Links that connect each node belonging to the same federation.
Attributes: The Link class has two attributes, id and name, to identify and name respectively each instance of Link class.
Relationships: The Link class has five relationships, 1) a composition relationship with Federation ④. Thus, as Nodes, Links also belong to a Federation (connecting two of their nodes),

2-3) two reference relationships with Node, thus allowing users to model which Nodes connect each Link. Note that since Links are designed to be unidirectional, one reference allows users to specify which Node will be the source and the other reference allows users to specify which Node will be the target, 4-5) two composition relationships with the classes Delay (4.2) and Bandwidth specifications (4.3). These references will be addressed below, in the Delay and Bandwidth specification class description.

- Delay_specification class (4.2)
  Aim: This class aims to allow users to model the delay of each Link.
  Attributes: The Delay_specification class has four attributes, 1) id, to identify each instance of Delay_specification class, 2) average_delay, to specify the average delay of the Link (4.1) (milliseconds), 3-4) max_delay and minum_delay, to allow users to model the maximum and minimum delay that will experience the Link during the simulation (milliseconds), and 5) distribution, to model the probabilistic distribution (i.e. gauss bell) from which the delay will be generated.
  Relationships: The Delay_specification class has one composition relationship with Link. In this way, users can model the Delay_specification of each Link.

- Bandwidth_specification class (4.3)
  Aim: This class aims to allow users to model the bandwidth of each Link (4.1).
  Attributes: The Bandwidth_specification class has two attributes, 1) id, to identify each instance of the class and, 2) bandwidth, to allow users to model the bandwidth of each *Link* (bytes).
  Relationships: The Bandwidth_specification has one composition relationship with Link. In this way, users can model the Bandwidth_specification of each Link.

5.1.5. Task Processor The Task Processor concept is not included in the metamodel by a specific class such as the Task, Task Node or the Task App concepts. Thus, the Task Processor is transparent to the user, who only has to be concerned about modelling the *Hardware_specification* ⑤ (CPU (5.1) and RAM (5.2)) of each Node, i.e. edge, fog or cloud node where the Task Processor node will be deployed.

Later, in the M2T, all the features modelled in these classes are considered to generate the Task Processor code of each Node (Section 6.5). Therefore, in order to model and include Task Processors in the simulations, several classes and relationships have been added to the metamodel of SimulateIoT. Thus, the following lists and describes the classes and relationships included in the metamodel to gather the Task Processor concept.

- Hardware_specification class ⑤
  Aim: This class aims to allow users to model the hardware of each edge, fog and cloud node.
  Attributes: The Hardware_specification class has one attribute, an id to identify the class.
  Relationships: The Hardware_specification has three relationships, 1) a composition relationship with Node. In this way, users can model the Hardware_specification of each Node (edge, fog and cloud nodes), 2-3) a composition relationship with CPU (5.1) and RAM (5.2). These relationships will be addressed below, in the CPU and RAM class descriptions.

- CPU class (5.1)
  Aim: This class aims to allow users to model the CPU of each Node.
  Attributes: The CPU class has three attributes, 1) id, to identify each instance of the class, 2) CPU_frequency, to allow users to model the frequency (cycles per second) of the CPU and, 3) CPU_cycles_per_bit, to specify the cycles that the CPU needs to process a bit (in the context of this communication, a bit of a Task).
  Relationships: The CPU class has one composition relationship with Hardware_specification ⑤. Since each Hardware_specification is related to a Node, in this way, users can model the CPU of each Node.

- RAM class (5.2)
  Aim: This class aims to allow users to model the RAM of each Node.
  Attributes: The RAM class has two attributes, 1) id, to identify the class and 2) memory_size, to allow users to model the size of the RAM.
  Relationships: The RAM class has one composition relationship with Hardware_specification ⑤. Since each Hardware_specification is related to a Node, in this way, users can model the RAM of each Node.

5.1.6. Task Scheduler node The Task Scheduler node concept is not included in the metamodel by a specific class such as the Task, Task Node or the Task App concepts. Thus, the Task Scheduler node is transparent to the user, who only has to be concerned about modelling one attribute of the Node class regarding this component, the TaskScheduler attribute.

Since the task scheduling model envisages one `Task Scheduler node` for each federation, users can use this attribute to specify which `Node` of each federation has to deploy the `Task Scheduler node`. Note that further description of this node is carried out in Section 6, where M2T are described.

Extending the metamodel of SimulateIoT with these classes and relationships, the concepts related to the task scheduling model defined in 4 are gathered by the metamodel. Therefore, the required expressiveness to model IoT systems with task scheduling capabilities is achieved.

5.2. Graphical Concrete Syntax and validator extensions

Model-Driven Development allows designing models conforming to a metamodel. So, in order to do this, the Eugenia tool (Kolovos et al. 2015) makes it possible to generate a Graphical Concrete Syntax (Graphical editor). 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). Consequently, models (EMF and OCL (Object Constraint Language) (OMG 2012) based) can be validated against the defined metamodel (EMF and OCL based). Note that OCL is a standard to define model constraints.

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 using this metamodel and OCL constraints (OMG 2012).

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

6. Extensions of Model-to-text transformations

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.

In Section 5, the extensions carried out in 1) Metamodel or Abstract Syntax (Subsection 5.1) and 2) Graphical Concrete Syntax (Subsection 5.2) were described. Thus, in this section, the extensions for the proposed simulator carried out in 3) Model-to-Text Transformations are described.

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

Table 1 Acronyms used in figures of Section 6.

Once the models have been defined and validated conforming to the proposed simulator metamodel (examples of a model in Figure 11), an M2T defined using Acceleo (Obeo 2012) can generate the IoT system modelled.

Thus, this section describes the main extensions included in the M2T component 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 4 (as in Section 5.1). 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.

Note that this section focuses on describing the inner workings of each component generated. Thus, the concepts on which these components are based are only referenced when they are necessary to describe the inner workings of the components. The concepts on which the generated components are based are addressed in Section 5.1.

Finally, note that Table 1 summarises the acronyms used in several of the figures included in this section.

6.1. Task

Section 5.1.1 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 of the components that are related to `Tasks`, such as the aforementioned.

However, as `Task` are handled by these components by means of JSON, below are the different JSONs that could be shared between components during a simulation.

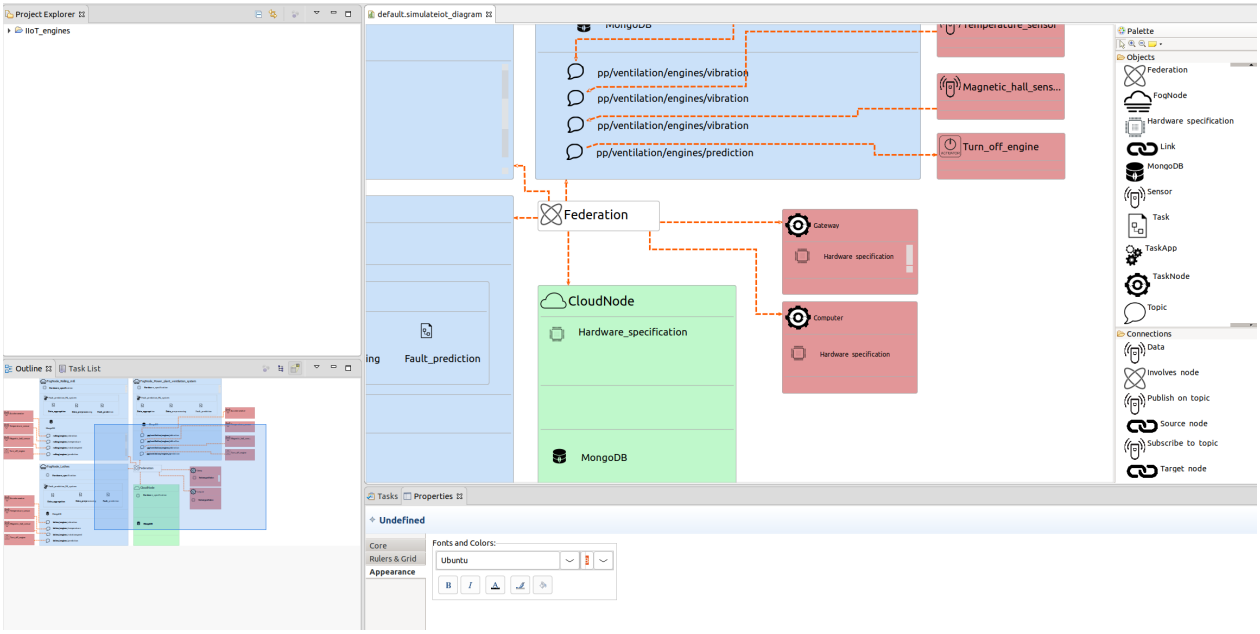

Figure 5 Graphical editor based on the Eclipse to model conforming to the proposed simulator metamodel.

To illustrate how Tasks are generated by the Task App or the Task Node (components that can generate Tasks), Listing 1 shows as an example the JSON related to the Task shown in Figure 3. This JSON includes all the necessary fields to represent a 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.

```
{
  "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 representation of a workflow. Specifically, the workflow illustrated in Figure 3.

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 related to the scheduling of `Task C`, which belongs to the workflow shown in Figure 3 and in Listing 1.

Among the fields of this JSON, 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 4) envisages dependency between `Tasks`, this JSON 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 related to the scheduling of TaskC of the workflow illustrated in Figure 3.

Once a node performs the processing of a `Task`, it includes in its JSON some fields related to the results of its processing. Listing 4 shows the JSON fields included 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 11:2
    9:54",
        "processingFinishedAt": "2023-1-25 11:
    29:57"
    }
        .
        .
        .
}
```

Listing 3 JSON excerpt that shows the fields related to the processing results of TaskC of the workflow illustrated in Figure 3.

As discussed above, since the task scheduling model (Section 4) 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 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 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 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 Task C of the workflow illustrated in Figure 3.

6.2. TaskApp

Figure 6 shows a generic `Task App` node Ⓑ (represented by a red box) and all its components (elements within the red box). In this regard, the main components of the `Task App` node are the `Task Generator` Ⓒ and a `MongoDBClient` Ⓔ. Besides, Figure 6 also shows how the `Task App` is deployed on a fog/cloud node and the interactions that its components could perform with other artefacts of the fog/cloud node, such as the interactions ① or ④ and with the rest of the IoT system, such as the interactions ② and ③. Note that all these software components are generated by the extended M2T as part of the contributions of this communication.

Below, the `Task App` node is illustrated by describing each of its components and their interactions with the rest of the system.

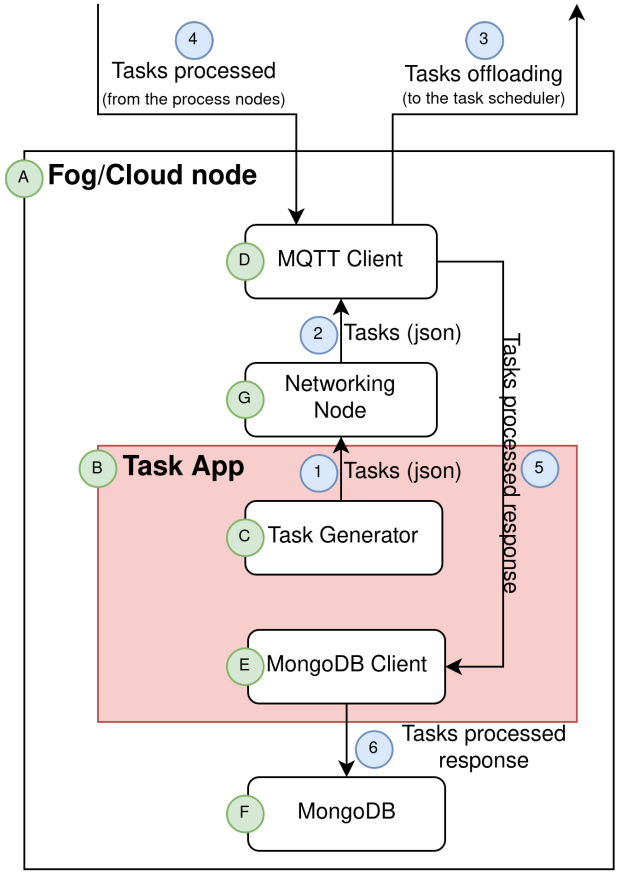

Figure 6 Task App component.

`Task Generator` Ⓒ As described in Section 4.2, the `Task App` can generate `Tasks` and offload them to the rest of the system. In this regard, the `Task Generator` is the component of the `Task App` whose aim is to generate and offload the `Tasks` modelled by the users in the modelling of the system stage.

Thus, the `Task Generator` sends to the `MQTT Client` Ⓓ of the `fog/cloud` node on which it is deployed, the generated `Tasks` in JSON format (interaction ① and ②). Note that the `Networking Node` behaviour is explained below (since this component is not part of the `TaskApp`). Later, the `MQTT Client` offloads these `Tasks` to the `TaskScheduler` of the federation (interaction ③).

`MongoDB Client` Ⓔ The `Tasks` offloaded by the `Task App` are processed by those nodes with processing capabilities that belong to the same federation that the fog/cloud node where the `Task App` is deployed. Later, these nodes return the `Tasks` processed to the `Task App`. Thus, in the context of task scheduling, the aim of the `MongoDB Client` is to receive these processed `Tasks` and store their results in the MongoDB database Ⓕ of

the fog/cloud node.

In this regard, the return of the processing results is represented by the interaction ④. These processing results are sent via MQTT protocol by the nodes that have processed them. So, the first to receive the results is the `MQTT Client` of the fog/cloud node, which forwards this data to the `MongoDB Client` (interaction ⑤). Finally, the `MongoDB Client` inserts the received results into the MongoDB database (interaction ⑥).

6.3. 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. In this regard, the main components of the `Task Node` are the `Task 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 with other artefacts of the system, such as the interactions ④ or ⑧. Note that all these software components are generated by the extended M2T as part of the contributions of this communication. Below, the `Task Node` is illustrated by describing each of its components and their interactions with the rest of the system.

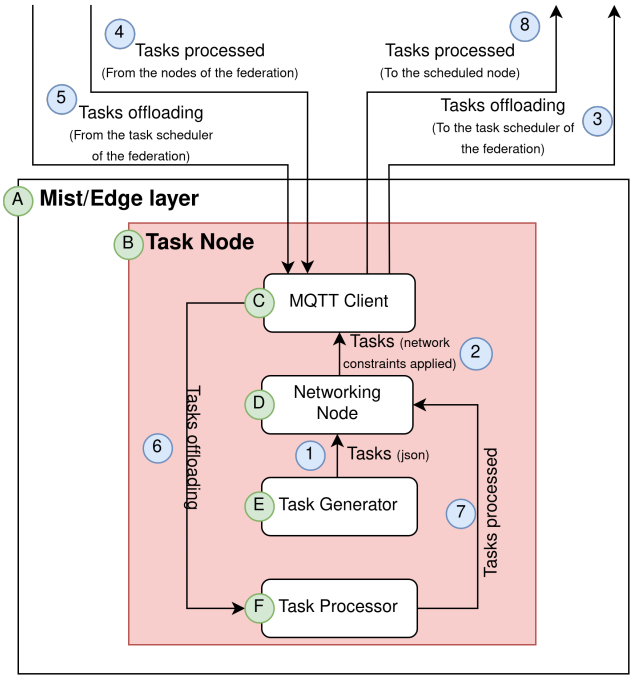

Figure 7 Task Node components.

`Task Generator` Ⓔ As the `Task App`, the `Task Node` also generates `Tasks`. Thus, the `Task Generator` is also present in the `Task Node`. This `Task Generator` has the same aim that the above described in the `Task App` section. However, in this case, the `Tasks` generated are not sent directly to the `MQTT Client` Ⓒ to be offloaded to the rest of the system. Instead of that, the generated `Task` are sent (interaction ①) to an element that acts as an intermediary between the `Task Generator` and the `MQTT Client`, the `Networking Node` Ⓓ.

The `Networking Node` 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 `Tasks` generated have to be offloaded to the system through a `Link`, the `Networking Node` applies to these `Tasks` 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 Section 6.4.

Thus, when the `Networking Node` has applied the bandwidth and delay constraints to the outgoing `Tasks`, it redirects them to the `MQTT Client` (interaction ②). Then, the `MQTT Client` offloads them to the federation's `Task Scheduler` (interaction ③). Finally, once processed, the `Tasks` (the processing results) return to the `Task Node` (interaction ④).

`Task Processor` Ⓕ `Task Nodes` Ⓑ belongs to the edge/mist layer, however, they can be part of the computing power of a federation, thus being able to process `Tasks`. In this regard, the `Task Processor` is the component of the `Task Node` that performs the processing of `Tasks`.

In this way, the `Task Scheduler` could schedule and assign the processing of a set of `Tasks` to a `Task Node`, thus offloading these `Tasks` to the `Task Node` (interaction ⑤).

As these `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 illustrated in Listing 2). However, as in the case of the `Task Generator` Ⓒ, the processed `Tasks` are not sent directly to the `MQTT Client` (to be sent to the target nodes). Instead, they are sent to the `Networking Node` Ⓓ (interaction ⑦). This is because the `Networking Node` also applies the delay and bandwidth constraints to these outgoing processed `Tasks`. So, once these network constraints have been applied, the `Networking Node` forwards the processed `Tasks` to the `MQTT Client` (interaction ②), which

sends them to their target nodes, i.e. the nodes that have been scheduled by the Task Scheduler to receive them (interaction ⑧).

Note that the behaviour of the Task Processor is described in more depth in the section reserved for it (Section 6.5).

Networking Node Ⓓ Users can model network aspects such as the delay and the bandwidth between each node belonging to a federation. The Networking Node aims to ensure compliance (during simulation) with these network aspects modelled.

In this regard, since the Links between nodes are unidirectional, the delay and bandwidth constraints are applied to the outgoing traffic of each node. In the context of the Task Node, the sources of the data are the *Task Pocessor* Ⓕ and the TaskGenerator Ⓔ, i.e. the Task Processor returns to the system the Tasks that it has processed, and the Task Generator sends the Tasks that it generates also to the system. Note that this offload to the system (of processed or generated Tasks) is performed by means of the MQTT Cliennt. In this context, the Networking Node acts as an intermediary between these Tasks and the MQTT Client. Thus, applying (simulating) the network constraints modelled by the user to them (interactions ① and ⑦). Once the constraints are applied, the Tasks are forwarded to the MQTT Client which sends it to their scheduled node (interaction 8).

Note that the Networking Node is described in more depth in the section reserved for it (Section 6.4).

MQTT Client Ⓒ This component performs the data sending/receiving on the Task Node Ⓑ. The MQTT Client was already included in the previous version of the simulator, however, it has been updated to support the performance of the rest of the components included as contributions in this communication.

In the context of the Task Node, the MQTT Client receives the processed and generated Tasks of the Task Node (interactions ① and ⑦), sending them to their target nodes (interactions ③ and ⑧). On the other hand, the MQTT client also receives all the Tasks sent from the IoT system to the Task Node (interactions ④ and ⑤). The received Task can be a) Processed Task, i.e. the return of the processing results of the Tasks offloaded by the Task Node that have been processed by the IoT system (interaction ④), b) Offloaded Tasks of other nodes of the federation that are scheduled by the TaskScheduler to be processed in the Task Node (interaction ⑤). Thus, a) for the processed tasks, the Task Node does not carry out any interaction but stores the results of their processing in its log, and b) for the offloaded Tasks the MQTT Clients forwards them to the Task Processor Ⓕ (interaction 6) for their processing.

6.4. Networking Node

Figure 8 shows a generic Networking Node Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. In this regard, 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 8 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. Note that all these software components are generated by the extended M2T as part of the contributions of this communication. 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. In this regard, 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 a workflow in JSON format (Listing 1). This JSON has among its fields the target node of the Task (Listing 1, field *generatedBy*), i.e. where the Task has to be sent. In this way, when the Delay Controller receives a Task, generated (interaction ①), or processed (interaction ②), it is able to identify the Link through which the Task has to be sent.

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

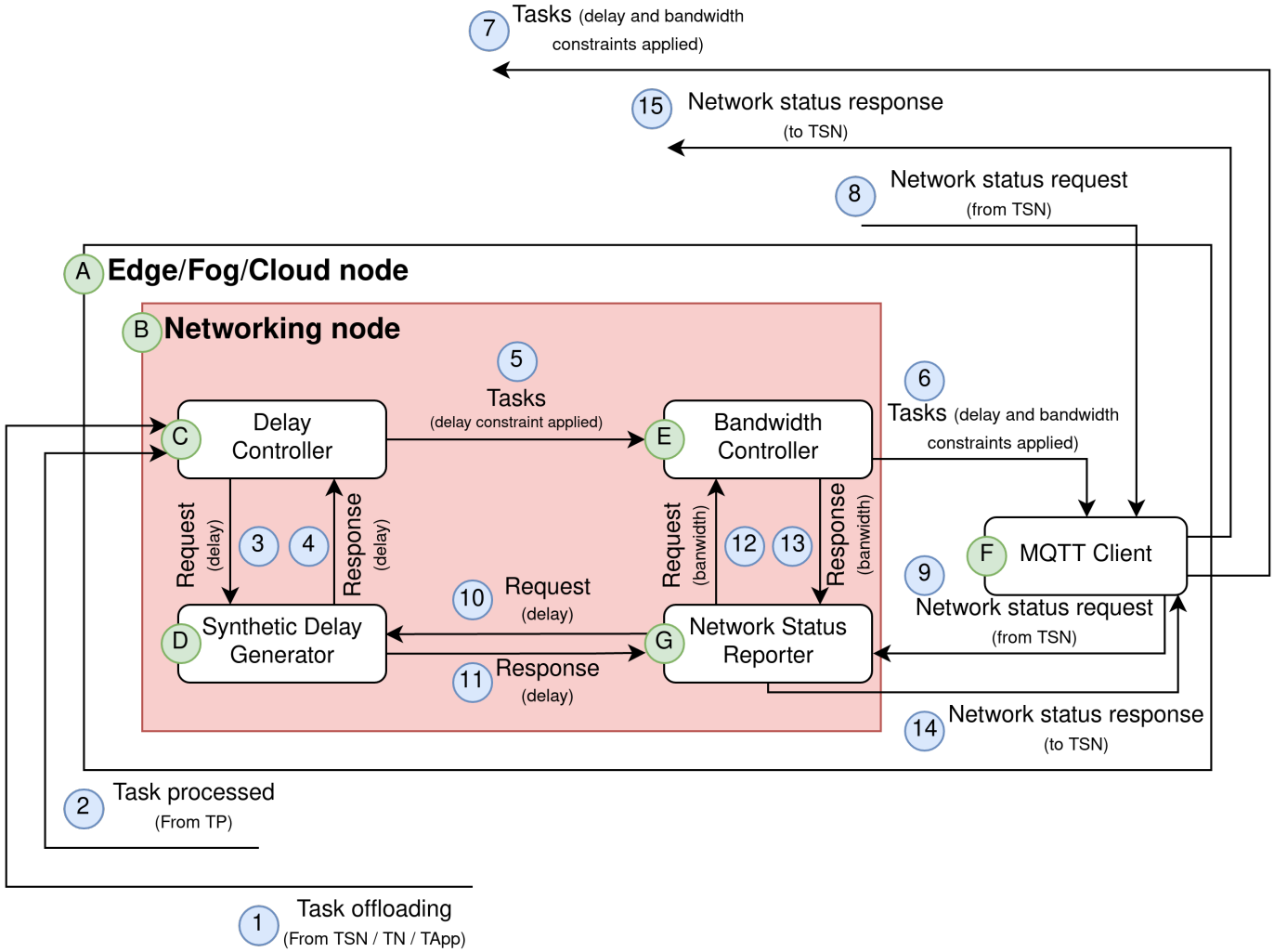

Figure 8 Networking Node component.

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 there is one `Networking Node` per node belonging to a federation, the `Bandwidth Controller` applies the bandwidth modelled to those `Links` whose source node is the edge/fog/cloud node where it is deployed.

Thus, the `Bandwidth Controller` receives `Tasks` from the `Delay Controller` (interaction ⑤). If a `Task` t_i is received and no other `Tasks` are being transmitted, the `Bandwidth Controller` holds the `Task` t_i for the time resulting from applying the following mathematical expression. Thus, simulating the time that the `Task` would have needed to be transmitted in a real environment.

$$TT_{t_i} = \frac{TS_{t_i}}{LB_{t_i}}$$

Where TT_{t_i} is the transmission time (seconds) required to send a `Task` 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 `Task` t_n arrives at the `Bandwidth Controller`, but there are $n-1$ `Tasks` being transmitted or pending to be transmitted through the same `Link` over which the `Task` t_n has to be transmitted, `Task` t_n is queued in a FIFO (First In First Out) `Task` queue. So, in this case, the transmission time of the `Task` t_n can be determined by the following expression.

Thus, simulating the time that the Task t_n would have needed to be transmitted in a real environment.

$$TT_{t_n} = \left(\sum_{n=1}^{n} \frac{TS_{t_n}}{LB_i} \right) + RT_{t_0}$$

Where a) TT_{t_n} is the transmission time required to send a Task t_n with a size of TS_{t_n} bytes through a Link with a bandwidth of LB_i bytes, b) over which a Task t_0 is being transmitted and RT_{t_0} bytes of this Task remain to be transmitted (when t_n arrives), and c) a set of $n-1$ Tasks are pending to be transmitted (queued before t_n).

Thus, once the delay and bandwidth constraints are applied, the Task 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 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 Links whose source node is the edge/fog/cloud node where it is deployed.

In this regard, 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. Then, 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 forwards the data to the Task Scheduler.

6.5. Task Processor

Figure 9 shows a generic Task Processor Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. In this regard, the main components of the Task Processor are the Task Manager Ⓓ, the Task Performer Ⓔ and the Task Processor Status Reporter Ⓕ. Besides, Figure 9 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. Note that all these software components are generated by the extended M2T as part of the contributions of this communication. 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.

In this regard, the Task Scheduler of a federation performs the scheduling of the Tasks offloaded to the federation. Then, following the schedule, the Task Scheduler sends these Tasks to those nodes of the federation with computing power, such as edge (Task Nodes), fog or cloud nodes.

Then, these Tasks reach the Task Processors of the computing nodes to be processed. As the Tasks 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.

Note that before sending the Tasks to the Task Performer, the Task Manager could include in these Tasks (fields in the JSON) data related to the processing results of their predecessor Tasks.

In this regard, since the task scheduling model (Section 4) envisages dependency between Tasks, when a Task is processed, the 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. In this way, when the Task Manager of a Task Processor receives a processed Task (interactions ⑭ and ⑮), it includes their processing results to the Task that needs them to be processed (the dependent Task). Note that the JSON that contains the results of the processed Task and the JSON that represents the dependent Task have the data needed by the Task Manager to perform the above-described process.

In this way, when the last Task of a workflow is processed, it will include the processing results of the rest.

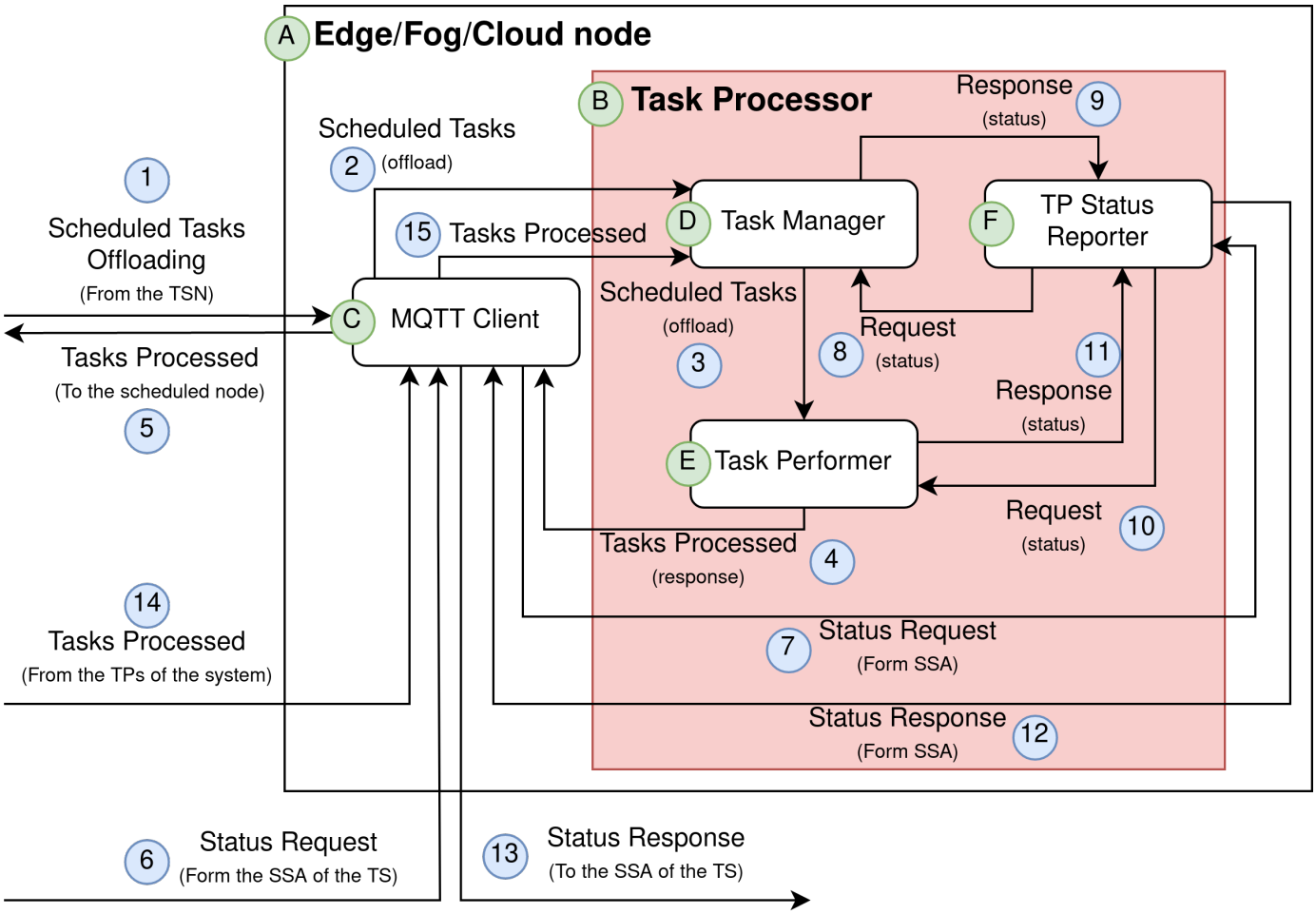

Figure 9 Task Proccessor components.

Note that this JSON is the processing result that is returned to the node that generated and offloaded the `Task` to the system for its processing.

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` that reach it.In this regard, the `Task Manager` sends `Tasks` (following the schedule carried out by the `Task Scheduler`) to the `Task Performer` Ⓔ for their processing (interaction ⑧). In this way, the `Task Processor` 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 following expression:

$$PT_{t_i} = \frac{TS_{t_i}}{CCB_{c_i} \times CF_{c_i}}$$

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 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 6.4), 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.

In this regard, 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).

In this way, 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 gather 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).

6.6. Task Scheduler

Figure 10 shows a generic Task Scheduler Ⓑ (represented by a red box), all its components (elements within the red box) and the interaction between them. In this regard, the main components of the Task Scheduler are the System Status Agent Ⓖ, the Task Scheduler API Ⓔ and the Task Scheduling Proposal Ⓕ. Besides, Figure 10 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. Note that all these software components are generated by the extended M2T as part of the contributions of this communication. Below, the Task Scheduler is illustrated by describing each of its components and their interactions.

Task Buffer Ⓑ The Task Apps and the Task Nodes generate and offload Tasks to the system. In this regard, the Task Scheduler Ⓑ is the component that first receives these Tasks with the purpose of scheduling their processing. In this context, the Task Buffer acts as a buffer of Tasks, providing the Task Scheduler Proposal Ⓕ with them when requested for their scheduling.

In this regard, since the task offloading is carried out through the MQTT protocol, the MQTT Client Ⓒ is the first to receive them (interaction ①). Then, the MQTT Client sends these Tasks to the Task Buffer (interaction ②). Thus, the Task Buffer stores the Tasks acting as a Buffer for the Task Scheduling Proposal Ⓕ requests. Note that the Task Scheduling Proposal performs these requests by means of the Task Scheduler API Ⓔ (interactions ③ and ④).

System Status Agent Ⓓ The System Status Agent aims to gather the status of all the nodes that belong to the same federation that the Task Scheduler where it is deployed. Thus, this component requests the status of the Links that connect the nodes of the federation to the Network Status Reporter component of each Networking Node. Besides, the System Status Agent requests to the Task Processor Status Reporter component the status related to the hardware usage of each Task Processor that belong to the same federation that the Task Scheduler. Then, the System Status Agent provides this data to the Task Scheduling Proposal when requested by means of the Task Scheduler API Ⓔ (interactions ③ and ④). In this way, the Task Scheduling Proposal could use this data as input for task scheduling purposes.

For this purpose, when the System Status Agent receives a request from the Task Scheduling proposal, the System Status Agent forwards it to the MQTT Client (interaction ⑦), which sends this request to the Task Processors and to the Networking Nodes of the federation (interaction ⑧). Later, these components respond to the System Status Agent with the status of the system (interaction ⑨). As this response is sent through the system via MQTT protocol, the first component that receives it is the MQTT Client (interaction ⑨), which forwards it to the System Status Agent (interaction ⑩). Once the System Status Agent receive the response (status of the system), it sends this data to the Task Scheduling Proposal by means of the Task Scheduler API (interaction ⑪ and ⑫).

Task Scheduler API Ⓔ The Task Scheduling Proposal is the task scheduling approach that the user can deploy into the simulation for test or analysis purposes. In this context, 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 and 2) provide the Task Scheduling Proposal with the Tasks that have been offloaded to the Task Scheduler Ⓑ and 3) return the Tasks scheduled to the system.

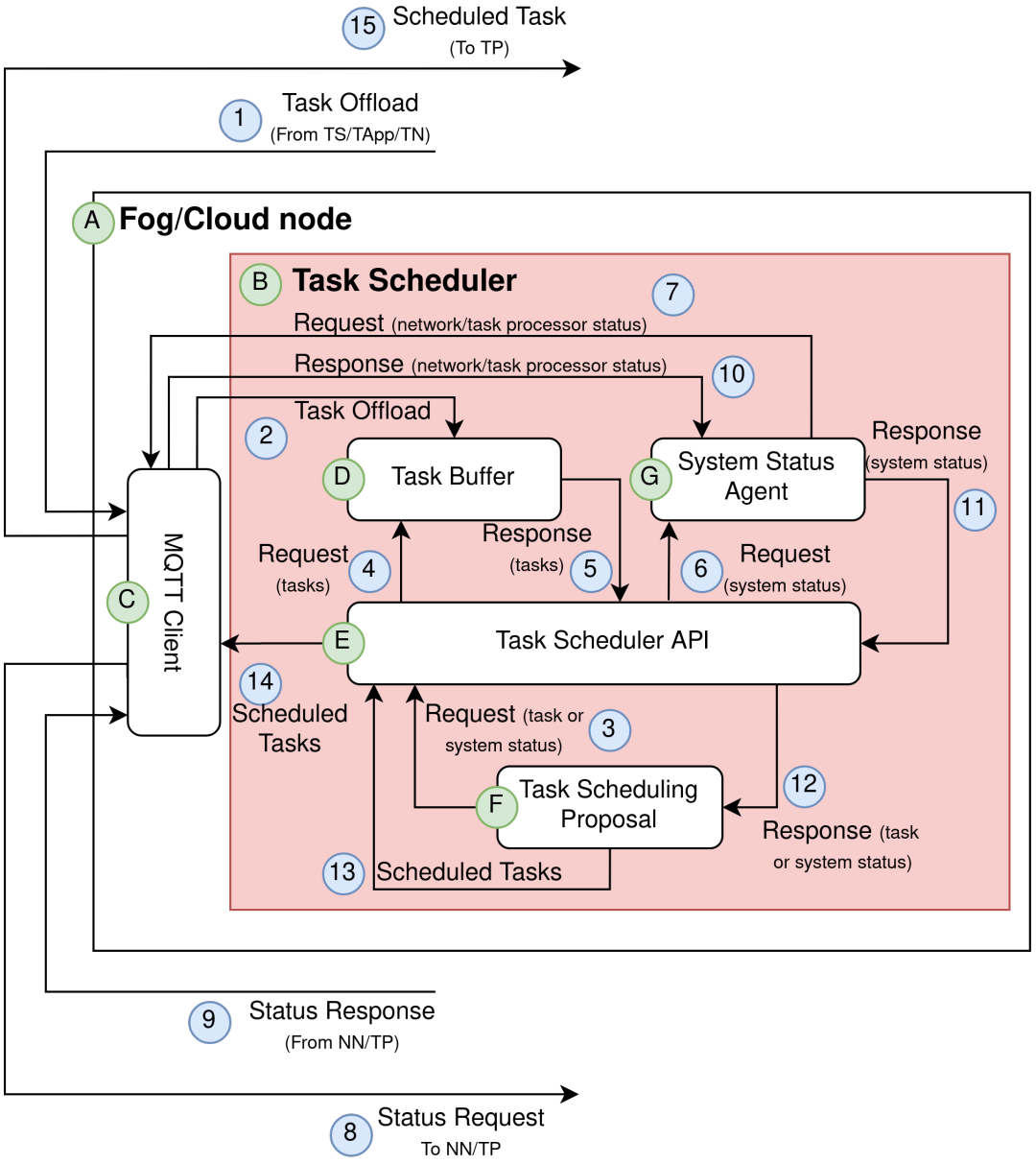

Figure 10 Task Scheduler components.

In this regard, the `Task Scheduling Proposal` can request to the `Task Scheduler API` the offloaded `Tasks` and the status of the nodes belonging to the same federation (interaction ③). If the `Task Scheduling Proposal` requests offloaded `Tasks`, the `Task Scheduler API` request for them to the `Task Buffer` Ⓓ (interaction ④). Then, the `Task Buffer` returns the `Tasks` that have been offloaded to the `Task Scheduler` ⑤. Note that the `Task Scheduler API` allows the `Task Scheduling Proposal` to specify the number of `Tasks` to be gathered from the `Task Buffer` Ⓓ. Once the `Task Scheduler API` receives the offloaded `Tasks`, it forwards them to the `Task Scheduling Proposal` (interaction (12)).

On the other hand, the `Task Scheduling Proposal` could request data related to the status of the system (interaction ③). So, when the `Task Scheduling Proposal` requests this data, the `Task Scheduler API` receives it, forwarding it to the `System Status Agent` (interaction ⑥). Once the `System Status Agent` has gathered the data related to the status of the system, responds to the `Task Scheduler API` (interaction (11)), which forwards this response to the `Task Scheduling Proposal` (interaction (12)). Thus, with the offloaded `Tasks` and with the status of each node and `Link` of the federation, the `Task Scheduling Proposal` can perform the schedule of each `Task`. Thus, when 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` is the task-scheduling approach that the user can deploy into the simulation for test or analysis purposes. The aim of the `Task Scheduling Proposal` is to schedule the offloaded `Tasks`. In this regard, the `Task Scheduling Proposal` only interacts with the `Task Scheduler API` Ⓔ. This is because the `Task Scheduling Proposals` are 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. In this way, by means of a series of requests, the `Task Scheduling Proposal` can interact with the simulated system receiving the offloaded `Tasks`, 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, in the `Task Scheduler API` description.

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 and therefore to improve the QoS of the services provided (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 QoS and SLAs 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 aim of the modelled system (using the metamodel presented in this communication) shown in 11 is to provide to a steel company with the capability to perform predictive maintenance on their electric engines.

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 (`Task`) of the publishing data for each sensor, 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).

Finally, a cloud node has been added to the system. This cloud node aims to provide additional hardware resources to the system if needed to meet user-defined QoS or SLAs.

7.3. Model definition

Figure 11 shows an excerpt of the IIoT system model. For the purpose of explaining this model, it includes numerical references for each node, which are refer-

enced 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 halls (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 halls 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 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 has 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. Besides, this `Task Node` carries out a `Task`, the data aggregation of the published data T.7. 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 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 this `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.

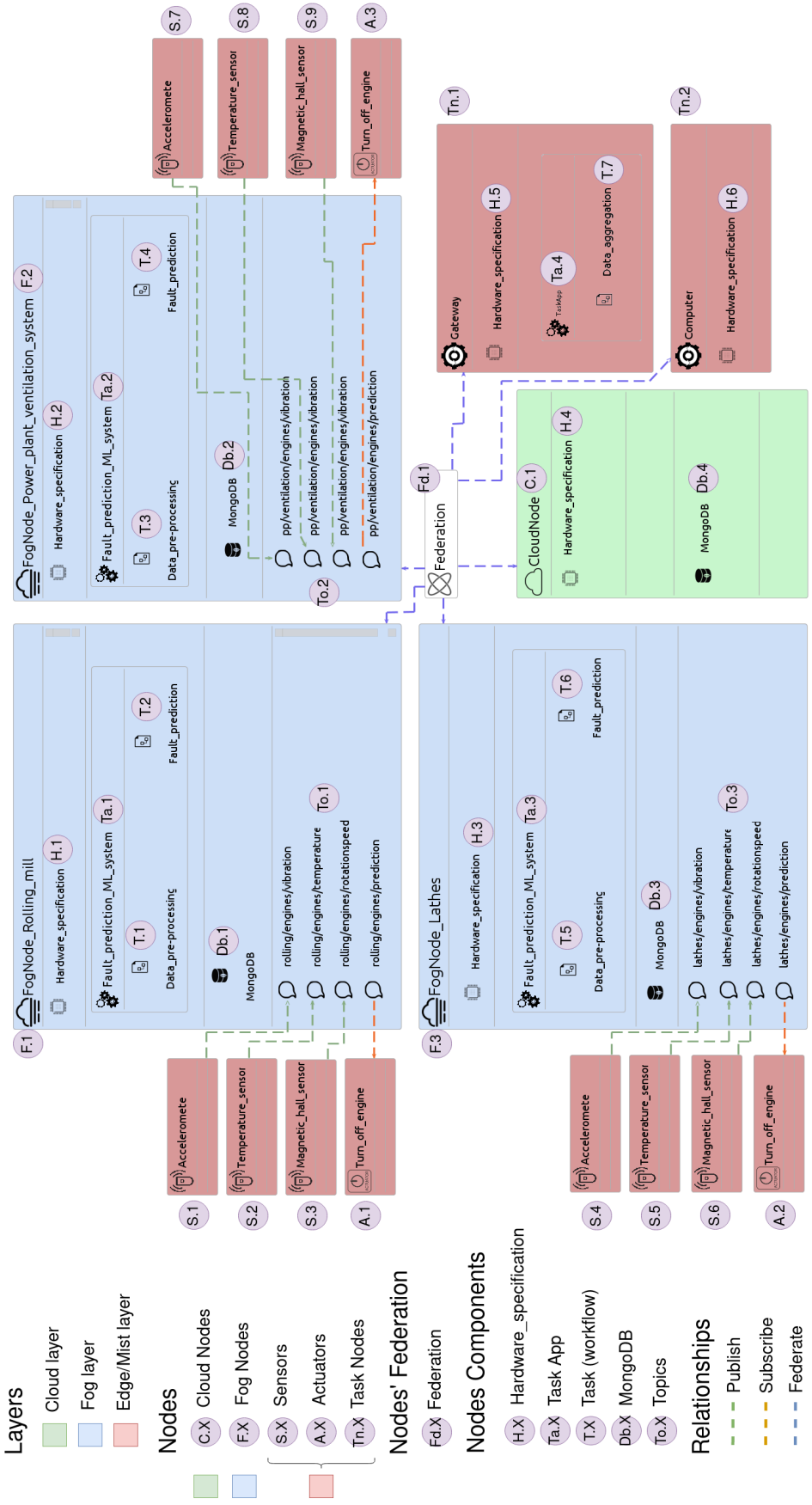

Figure 11 Model conforms to the proposed simulator metamodel. An IIoT system applied to the steel industry for predictive maintenance of electric engines.

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 inter-connects 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. Model-to-text transformations

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 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 6 where model-to-text transformation has been defined.

7.5. Simulation analysis

The benefits that can be obtained from a simulator come from the outputs, data, etc. that can be obtained from the simulations carried out. In this section, a set of tests based on the model described above are carried out, illustrating the possibilities and benefits provided by the proposed simulator. To carry out these 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 M2T only need the task scheduling proposal (in this case the HEFT algorithm) to be on the same path as the generated components (by M2T) 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).

Once the simulation has finished (120 seconds), the average time reported to predict the failure of an engine (any engine) is 4.391 seconds, and the maximum time 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 the required QoS and SLAs. 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.

Once the simulation has finished (120 seconds), 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 redesign them until reaching an optimal configuration that satisfies their requirements (QoS. SLAs, etc.). 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 is suitable to handle the technological complexity of domains where heterogeneous technologies are used. These techniques focus on modelling the application domain domain by using the well-known four-layer metamodel architecture. Then, by using model-to-text transformations the code for specific technology can be generated.

The domain-specific language proposed in this communication for IoT simulation help users to think about the task scheduling IoT system design. It allow users to propose several IoT designs and task scheduling solutions in order to achieve a suitable system, i.e. a system that satisfies users' QoS requirements. In this regard, several components can constitute the IoT system, such as cloud, fog, edge and mist nodes, allowing their federation, devices and applications capable of generating and offloading workflow-based tasks, the architecture required for processing these tasks, etc. providing users with a simulation tool to model realistic IoT systems where task scheduling is a key factor. Finally, the IoT systems modelled can be generated (their code) and deployed. Then, the simulation outputs can be gathered and analysed for system's improvement purposes.

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 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 systems (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). Introduce the concept of devices' batteries or system energy consumption to the proposed simulator, could help those users who requires 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). Integrate the concept of resources cost or the model of *pay-as-you-use* could be interesting to allow these users to test their proposals.

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:

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

Adhikari, M., Amgoth, T., & Srirama, S. N. (2019, aug). A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Comput. Surv.*, *52*(4). Retrieved from https://doi.org/10.1145/3325097 doi: 10.1145/3325097

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

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

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.

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.

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*.

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

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.

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.

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 12 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.

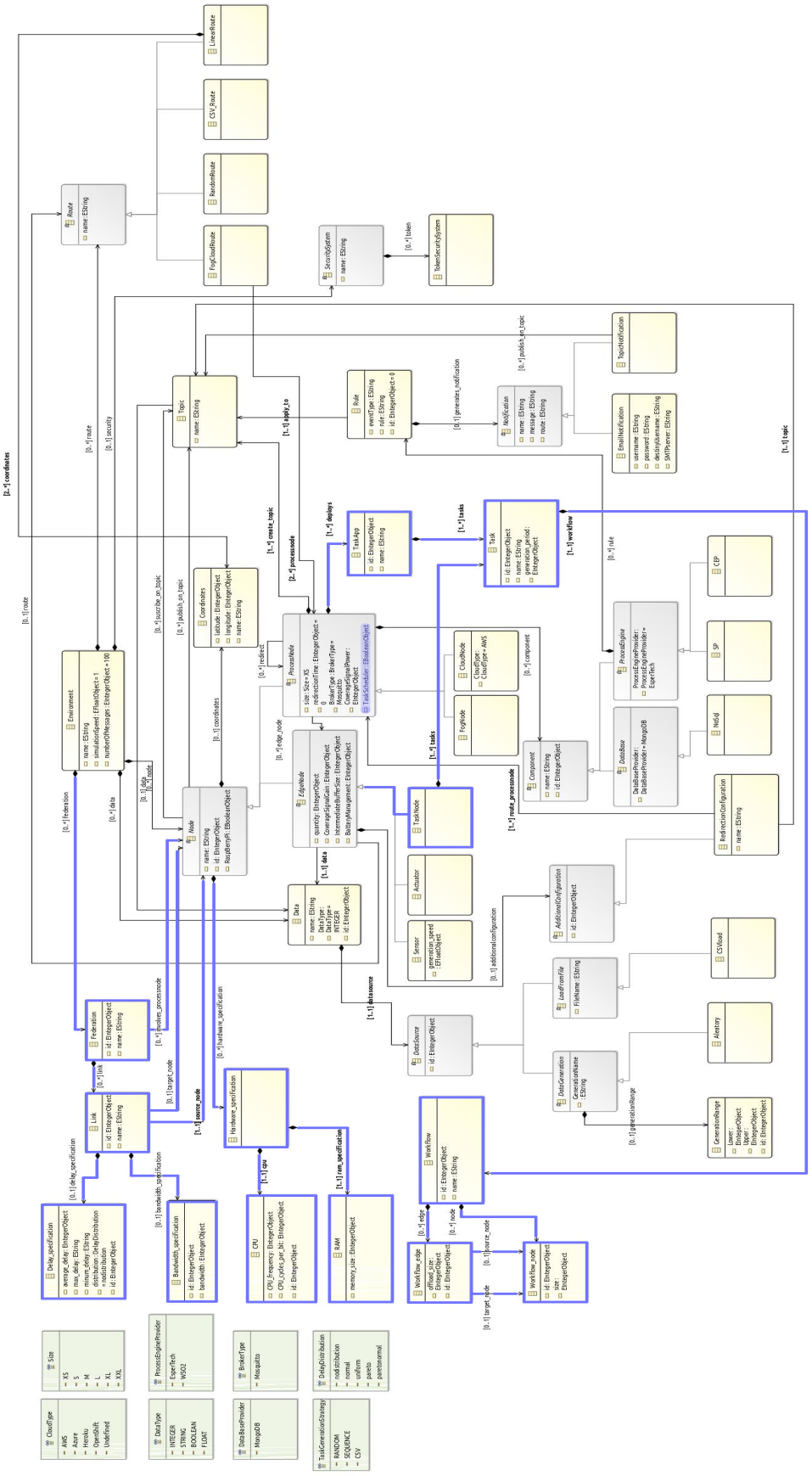

Figure 12 Complete SimulateIoT metamodel with the extension concepts and relationships included.