D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

A Model-Driven Engineering Approach for the Service Integration of IoT Systems

Darwin Alulema[1,2] · **Javier Criado**[2] · **Luis Iribarne**[2] · **Antonio Jesús Fernández-García**[2] · **Rosa Ayala**[2]

Received: date / Accepted: date

Abstract With the development of IoT devices and web services, the objects of the real world are more interconnected, which allows applications to extend their characteristics in different fields, including industrial or home environments, among other possible examples such as health, trade, transport, or agriculture. However, this development highlights the challenge of interoperability, because devices are heterogeneous and use different communication protocols and different data formats. For this reason, we propose a model for point-to-point integration in three-layer IoT applications: (a) hardware, which corresponds to the physical objects (controller, sensor and actuator), (b) communication, which is the bridge that allows the exchange of data between a MQTT [36] queue and REST web services, and (c) integration, which establishes a sequence of transactions to coordinate the components of the system. For this purpose, a metamodel, a graphic editor and a code generator have been developed that allow the developer to design IoT systems formed by heterogeneous components without having in-depth knowledge of every hardware and software platform. In order to validate our proposal, a smart home scenario has been developed, with a series of sensors and actuators that combined show a complex behavior.

Keywords Model-Driven Engineering (MDE) · Domain-Specific Language (DSL) · Web Services · Integration Pattern · Internet of Things (IoT) · Smart Home

Darwin Alulema
E-mail: doalulema@espe.edu.ec

Javier Criado
E-mail: javi.criado@ual.es

Luis Iribarne
E-mail: luis.iribarne@ual.es

Antonio Jesús Fernández-García
E-mail: ajfernandez@ual.es

Rosa Ayala
E-mail: rmayala@ual.es

[1] Universidad de las Fuerzas Armadas ESPE, Sangolquí, Ecuador
[2] Applied Computing Group, University of Almería, Spain

1 Introduction

Nowadays, objects have greater processing, storage and communication capabilities, among others [9,28], which has allowed them to be integrated with traditional enterprise level services [30,34]. This development has a very important role in the Internet of Things (IoT), since it allows us to deal with the complexity of systems, increase the visibility of information and improve production performance [13,35]. However, we face the problem of interoperability, because IoT devices can be heterogeneous, use different protocols (*e.g.*, HTTP, MQTT, DDS and CoA) and manage different data formats (*e.g.*, binary, XML, JSON and GIOP) [11,22].

From the wide diversity of devices and platforms, two main challenges have emerged. Firstly, the heterogeneous nature of IoT information makes the task of interpreting that information and detecting real-world events more complex. Secondly, the delivery of sensory information generates some problems, such as the limited resource consumption of IoT devices [2]. One scenario that illustrates these challenges is that of a smart home, such as Figure 1, in which a Passive Infrared Sensor (PIR), a Light Dependent Resistor (LDR) and a contact sensor for the alarm (Switch) have been deployed. There are also some actuators such as a bulb

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

controlled by a relay (KY-019), a blind controlled by a servo motor (SG-90) and a buzzer.

The proposed scenario presents several challenges. Firstly, the design and subsequent implementation of applications capable of using information from IoT devices. For example, if a developer wants to add a device that reports light intensity to their application, how can they integrate their functionality into traditional web services? Secondly, IoT applications are subject to asynchronous events, the system must be reactive in the presence of different events generated in the IoT environment. For example, the homeowner arrives at night and the lights must be turned on. Thirdly, there is the difficulty of coordinating the actions of all the elements. For example, if we want to turn on the light only if it detects movement and the blind is closed.

An additional challenge when designing IoT applications, even for experienced developers, is understanding the languages related to the hardware and software platform, the application domain and requirements of the business model and high-level rules [34]. In this respect, the model-based approach (MDE) applied to the IoT allows the automation of various tasks of the development and operation phases [7,35]. In addition, the principles of Resource Oriented Architecture (ROA) are best suited for IoT applications [9], because it uses the design philosophy of Representational State Transfer (REST). Finally, in order to integrate services into a business process, the orchestrator-based integration pattern coordinates the execution of operations [17].

This article extends our preliminary studies in [3,4] in several ways:

(a) We provide a description of the communication attributes of the IoT nodes, specifically, the attributes related to the publication and subscription in an MQTT Broker [20] and the RESTful web services [29]. These mechanisms are associated with each sensor and actuator of the system to manage the generated information, in a standardised way.
(b) We develop a communication mechanism that we call Bridge that allows linking the MQTT messaging of each sensor and actuator with the RESTful web service. This mechanism allows the IoT nodes to be considered as a web service which facilitates their integration into traditional business systems and within the scope of the WoT (Web of Things) [16].
(c) We implement an integration pattern for IoT nodes based on Orchestration [1], which coordinates each web service. The business logic of the Orchestrator is based on combinational functions that determine the status for each actuator based on the current states of the associated sensors.
(d) We standardise the format of the information for IoT nodes through JSON objects. 6 fields with ID, Date, Time, Location, Attribute, and Device information were established.
(e) We develop a tool for modelling IoT nodes with communication and integration capabilities. It consists of a graphic editor and a code generator, which generates software artefacts for Arduino (IoT Node) devices, Node-Red (Bridge) and Ballerina (Orchestrator, RESTful Web service).

The rest of this article is organized as follows. Section 2 briefly discusses some concepts of MDE, RESTful services and integration patterns. Section 3 presents the proposed service integration model for IoT. Section 4 then provides a test scenario for the proposal validation. A review of the related work is then presented in Section 5. Finally, the conclusions and future work are outlined in Section 6.

2 Fundamentals and background

The approach presented in this paper applies different technologies belonging to three paradigms: Model-Driven Engineering, Resource-Oriented Architectures and Integration Patterns. With this aim, this section describes the fundamentals and the main features required to understand our proposal.

2.1 Model-Driven Engineering paradigm (MDE)

Models provide a simplified or partial representation of a reality, an abstraction that allows us to easily work with the concepts, rules and the structure underlying it. The reflection of a reality through a relevant selection of its properties provides ideal conditions to address to that reality according to some purposes or needs we may have.

The Model-Driven Engineering (MDE) paradigm defines software approaches to the definition of models, transformation and development process [6]. This paradigm helps us to achieve the main challenges that we face in this work such us provide abstraction from technologies improving the portability of IoT systems and facilitating the interoperability between the different components integrated in them or the automatization of the generation of code from abstract models drawn in our user interface that deals with specific component itself, increasing the productivity and efficiency as well as reducing errors and making the IoT system flexible to foreseeable needs.

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

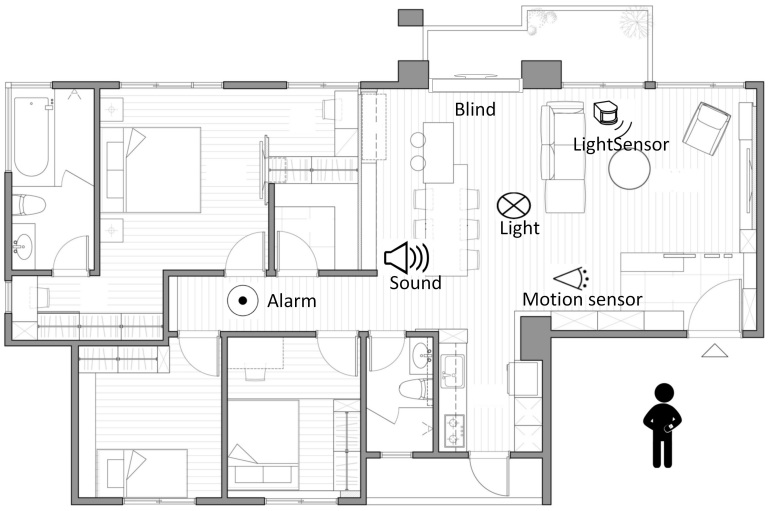


Fig. 1: Heterogeneous hardware systems in a Smart Home scenario

MDE impacts in the quality of the IoT systems by making easier the management of the increasing complexity of IoT systems. It helps to easy integrate different technologies (or migrating to new ones), provides adaptation to requirement changes as well as facilitate maintenance and documentation.

2.2 Resource-Oriented Architecture (ROA)

The main characteristics of Resource-Oriented Architecture (ROA) is that, by definition, is ready to work with interfaces that can be shared among all architectural components, contributing to more reliable systems where failures can be handled gracefully [24]. The ROA architecture is designed following the philosophy that services (resources) provides functionality to other applications through interfaces.

RESTful platforms enable the creation of ROAs. We decided to use REST architecture against SOAP [31] protocol, the most standardized, since IoT systems are widely used in the industry, where REST is more prevalent because of its flexibility and simplicity [8,23]. In general, REST has better performance and scalability, and SOAP requires more bandwidth and resources than REST. Besides, REST is widely used to connect cross-platform applications and even adopted by systems that allows end-user development using them [21].

2.3 Integration patterns: orchestration and choreography

The implementation of service-oriented architectures requires designing a workflow that enable the communication between services in real-time according to the applications needs. Modern IoT systems have to be capable of orchestrating the different technologies and resource types available in modern network infrastructures. Orchestration systems are responsible to manage the heterogeneity between technologies and enable autonomous and automated service deployment and adaptation [26].

The concept of orchestration refers to selecting and controlling services and resources with the aim to meet the requirements defined by users and applications. All of this in-time and with a certain level security and consistency [32].

3 A model for the Integration of IoT systems

In order to solve the problem of interconnecting heterogeneous devices, we propose a model that includes:

- **Hardware**: Corresponds to hardware devices of IoT systems. For example, a Smart Home can contain several devices such as televisions, heating, air conditioning, ovens, washing machines or doors, among others, that need to be controlled locally or remotely.
- **Control**: Corresponds to communication, information storage and hardware control in IoT systems. For example, a traffic management system, where hardware includes cars, traffic lights and sensors, which are coordinated to organize the traffic, changing the lights according to the number of cars in each direction of an intersection.

Figure 2 shows the proposed architecture, where intelligent objects (hardware) send and receive information from a control component. However, these objects pose the problem of unpredictability when interacting

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

with people, such as a motion sensor that detects when a person gets out of bed at night to drink a glass of water. In this case we are dealing with an asynchronous event. For this reason, the control stage must admit an asynchronous interaction. This is why we developed a software device called *Bridge*, which allows communication between messaging-based mechanisms (publication/subscription) and distributed architectures (request/response). To illustrate how useful this device is, let's suppose that a software client wants to change the state of a light bulb. To do this, this software client must execute a POST method with the new state for the device, which is sent to the specific device queue. On the other hand, if the software client wants to obtain the information that describes the state of a motion sensor, it must execute a GET method, from the device-specific API REST, which extracts the last message from the device queue.

The format for the information exchanged in the system is a standardization proposal that arises as a modification of Zhou's proposal [37] for processing complex algebraic events for cyber-physical systems. The format of the proposed information is standardized by means of JSON objects, with the following structure:

- ID: Identifier of the event.
- Date: Date on which the event takes place.
- Time: Time on which the event takes place.
- Location: Physical location of the device.

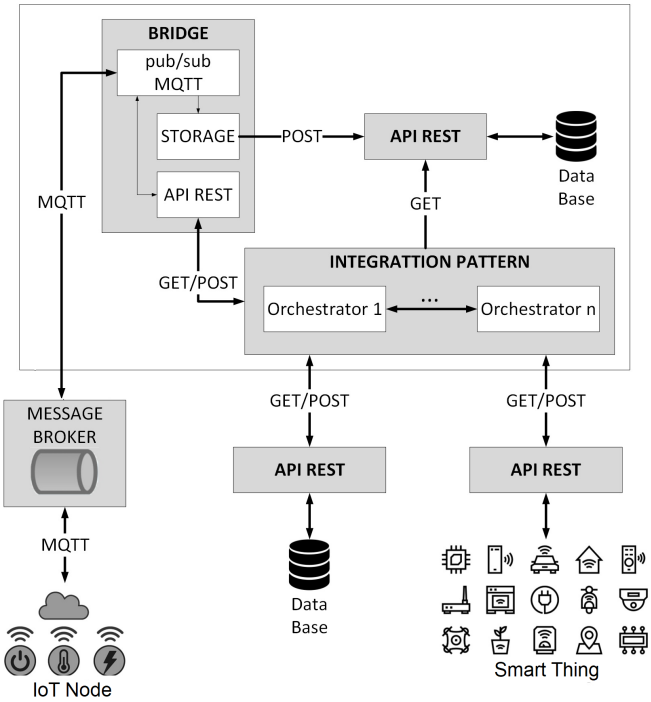


Fig. 2: Architecture of Integration

- Attribute: Value represented by the event.
- Device: Identifier of the software or hardware device.

For the integration of all the services associated to the components of the system, the Saga pattern [25] has been considered, which is a sequence of local operations, each operation updating the information within a single service. For this purpose, an orchestrator is established that tells participants which local operations they must execute. To illustrate how useful this device is, let's suppose that a person watches a video on their SmartTV in their living room and the blind automatically closes to avoid reflections on the screen and since there is still sunlight, the bulb turns off. As can be seen in the example, there are some simple events that have been coordinated to make the experience of watching a video more comfortable.

Within our example scenario, the *Bridge* receives in a queue the messages from the luminosity sensor or the motion sensor so that the Orchestrator coordinates operations with the bulb and the blind to implement the business logic. Messages from IoT devices or other services comply with the information format standard and can act as a *Trigger* for the Orchestrator. The Orchestrator uses the *Bridge* of a particular IoT device to send a message with the action to an actuator, if necessary. In addition, due to the changing environments of intelligent devices, our proposal deals not only with a centralized execution (orchestration) but also with a decentralized execution (choreography) for the business process. The first controls the execution of the participating services, while the second allows the expert in the domain to design and define simultaneously the communication between different processes.

3.1 Proposed metamodel

This subsection describes the metamodel for the integration of IoT systems following an approach based on MDE, by means of the Ecore meta-model language using Eclipse Modeling Framework (EMF) [33]. The proposal allows to define a set of attributes associated with hardware nodes, web services and databases. Each service has a set of operations (*e.g.*, GET, POST, PUT, DELETE) to access the resources created in databases for each hardware node. The implementation of these operations is created from properties of the devices (*e.g.*, sensor, actuator and controller), such as the ID of the event, date and time of occurrence, the location, the value of the attribute and the device name, and properties associated with the service such as the name of the selected object and the specific address of the service. In addition, it allows to define the business logic

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

that coordinates different operations to define the business logic.

As shown in Figure 3, the components of the IoT system are mainly grouped into three main categories:

- **Infrastructure**: Corresponds to the meta-classes that represent the characteristics of the necessary requirements for application deployment, and corresponds to `Web Server`,`Data Base Server`, `Acces Point` and `Message Broker.`
- **Hardware**: Corresponds to the meta-classes for the design of the IoT Nodes. This is the case of `Sensors`, `Actuators` and`Controller`, as well as the `Communication` meta-classes.
- **Control**: Corresponds to the meta-classes for the integration of each device's web services (sensor and actuator) to define the business logic. In this case, the `Bridge` and the I`Integration Pattern`, with the `Orchestrator` and `Función` and `Status`, that represent the operating logic of the actuators according to the states coming from the sensors in real time.

Figure 3 shows the metamodel that describes the Domain-Specific Language (DSL) [12] aimed to define the hardware nodes built on a development platform (*e.g.*, Arduino). These nodes also have connectivity (*e.g.*, serial, WiFi) to be linked or connected to other nearby devices and services. The platform also allows to control multiple types of analog or digital components (*e.g.*, sensors and actuators). However, to simplify the definition of the components input and output, it is only necessary to describe the ID of the Controller port to which devices are connected.

3.2 Proposed graphic editor

The proposed DSL allows the creation of software tools that can then be used by IoT application developers. In our case, we have developed two tools according to DSL semantics. The first consists of a graphical editor created with Sirius[1], which provides a palette of tools to model IoT scenarios (Figure 4). The second defined software tool that complies with DSL is a transformation engine, implemented in Acceleo[2], that allows model-to-text (M2T) transformations to generate code for both client and server.

Figure 4.a shows the editor's VSM (Viewpoint Specification Model) and Figure 4.b displays the generated tool palette. The graphic editor specification process considered the following activities: (a) creation of a Viewpoint Specification Project, (b) specification of a Viewpoint, (c) specification of the type of representation (Diagram), (d) mapping between the graphic elements of the diagram with the elements of the metamodel and (e) specification of the editor toolbar elements (Palette).

The VSM allows you to set up classes and their relationships, and to specify which objects are displayed and how they are displayed. The VSM includes: nodes (`WebServer`, `DBServer`, `AccesPoint`, `MessageBroker`, `OutputBridge`, `InputBridge`, `ExternlAPI`), containers (`WebService`, `IntegrationPattern`, `IoTNode`), subnodes (`Orchestrator`, `Function`, `Actuator`, `Controller`, `REST`, `Sensor`), edge nodes (`Status`, `Communication`, `InputOrchestrator`, `OutputOrchestrator`, `OutPort`, `InputPortcontroller`) and tool options to create system elements and the connections between them.

The graphic editor created has two sections: (a) Canvas, is the area destined to show and edit the diagram of the IoT application, and (b) Tool palette, is the area that displays the components to draw the visual representation. The palette groups the components of the IoT system into:

- **Infrastructure**, which shows the tools available to specify the infrastructure to be used in the IoT application (`WebServer`, `DBServer`, `Message`, `Broker`, `Bridge`, `AccesPoint`), for interconnecting all the components of the IoT system.
- **Control**, which shows the tools available for designing the REST services associated with each sensor and actuator to access its resources. In addition, it allows the specification of the business logic by means of orchestrators that coordinate a certain sequence of calls to the REST services, according to a logical function that describes the behavior of the system. It is composed of logical operators expressed in canonical form as a sum of *Minterms*, equal to that used in digital electrical systems.
- **Hardware**, which shows the tools available for the design of the IoT Nodes. It allows the developer to configure the controller (*e.g.*, ESP8266 Node MCU) by setting the input or output ports, whether they are digital or analog. It allows to select the sensors (*e.g.*, CO_2, Light, Button, HumidityG, Vibration, Temperature, Movement, Contact, TempHum) or actuators (*e.g.*, Buzzer, Led, Relay, Servo, LCD) that are connected to the controller. In addition, it allows to configure the connection to Internet by means of the AccesPoint and the Server of Messaging to establish the scheme of publication and subscription in a queue of messages.

[1]Sirius website – `https://www.eclipse.org/sirius/`

[2]Acceleo website – `https://www.eclipse.org/acceleo/`

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

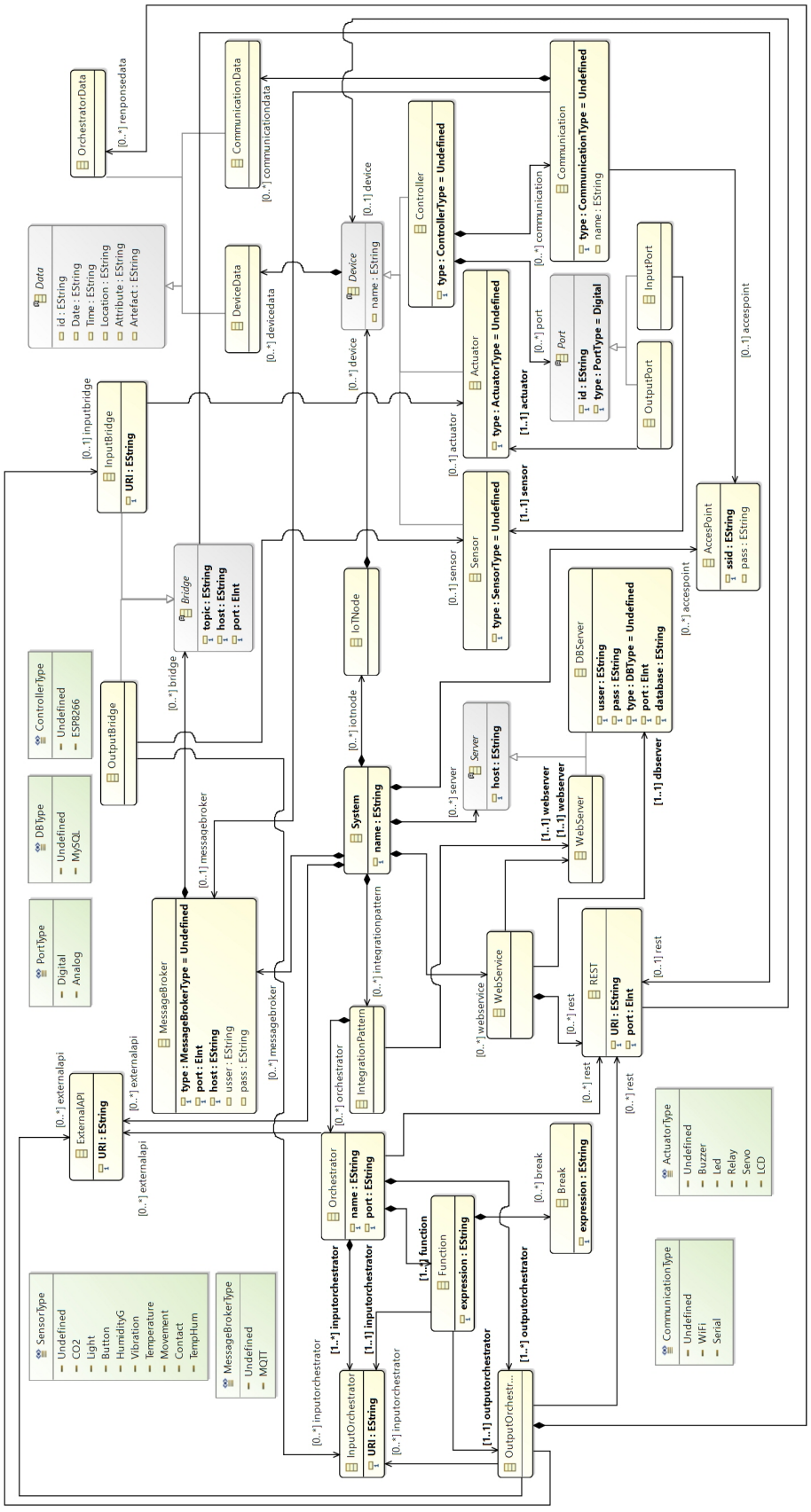


Fig. 3: Meta-model for integration

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

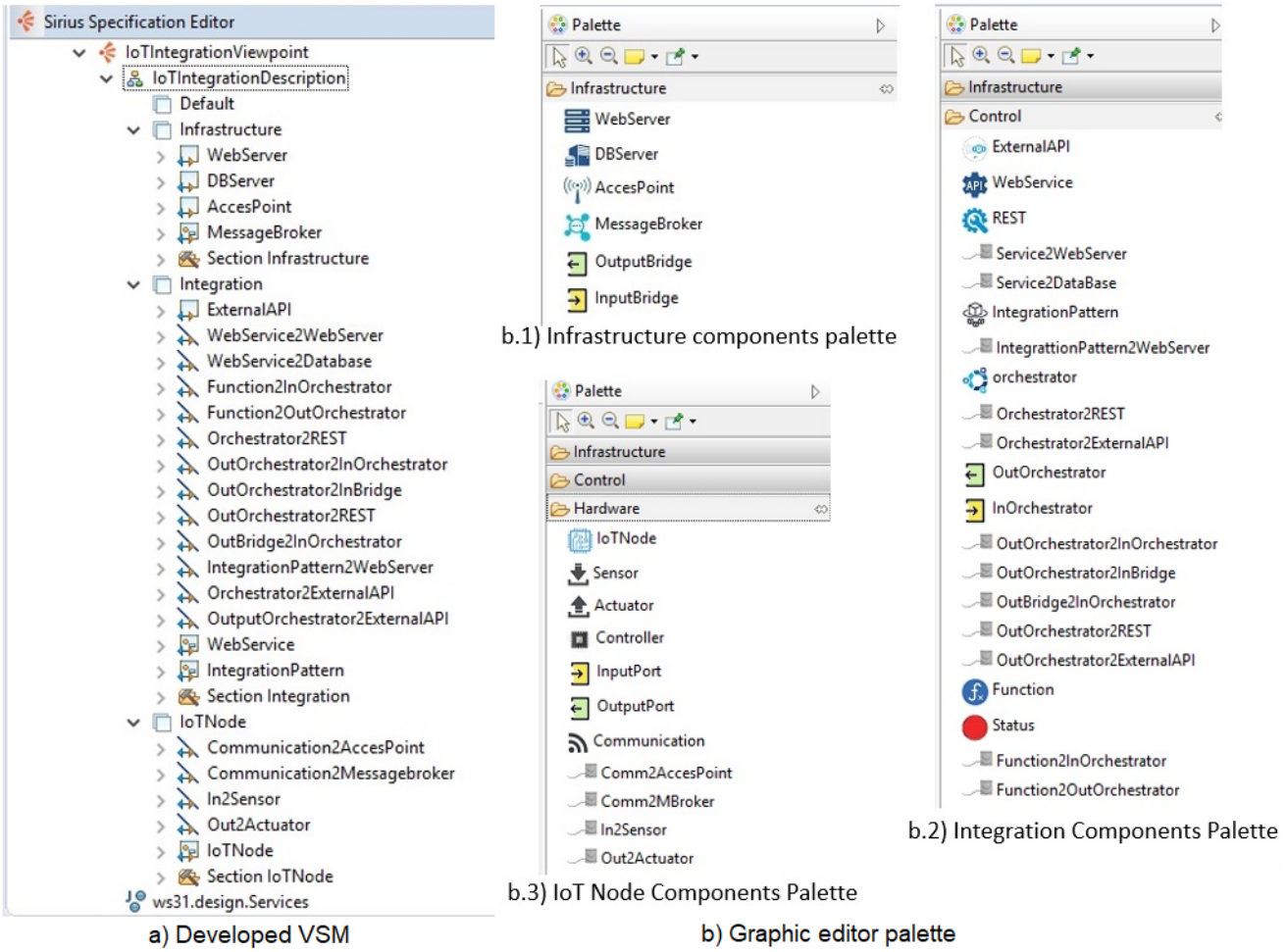


Fig. 4: Screen capture of the Viewpoint in Sirius and the tool palette created

3.3 Transformation Engine

The code generator is the second software tool defined. To build the generator, a model-to-text (M2T) transformation engine is implemented. In our case, the Acceleo tool has been used for code generation. In the following code fragments, we present a summarized code fragment for the M2T transformation that is done according to the DSL to generate instances of the Web Service, Bridge, Orchestrator and IoT Node.

The first rule to be defined is the Template, because it must create an instance of the system, `aSystem` in this case. Sequentially the following rules are: (a) defining a file for the Hardware, Infrastructure and Control, (b) defining configuration for the hardware connections and (c) defining the infrastructure characteristics, and (d) defining RESTful characteristics, Bridge characteristics and IntegrationPattern characteristics for the device control.

To define the transformation rules, the Arduino platform was considered due to its ease of use. WiFi was considered for connections and Node-Red was used for Bridge, due to its simplicity to create publication/subscription flows with MQTT Topics of the IoT Nodes, REST clients to consume POST methods to store information coming from the Nodes, and REST services to allow communication with the Orchestrators. Ballerina was chosen for the services and the orchestrator, because of its parallel service call capabilities. These platforms were chosen because they reduce the amount of code needed for configuration, which is best suited to Acceleo. The transformation process generates Arduino code files (`.ino`) for the Controller, code that represents the flows for Node-Red (`.json`) for the Bridge, and Ballerina code (`.bal`) for REST services and the Orchestrator.

The platform chosen for IoT Node design is the *ESP 8266 Node MCU*, due to its ability to receive and send information over the Internet [5]. The structure of the

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

Arduino programs that will run on the platform are divided into three sections that can be seen in the code fragment shown in Listing 1: (a) Statement of constants associated with sensors and actuators, and the pin port of the controller board, which can be analog or digital and referenced to libraries `ESP8266WiFi.h`, `EP8266HTTPClient.h` and `PubSubClient.h` that allow WiFi connectivity and sending and receiving MQTT messages, (b) `wifi()`, `mqtt()`, `sensors()`, `actuators()` and `setup()` functions, where all initial connectivity parameters and sensor and actuator configuration associated with a variable with the name of the specific topic for the device are initialized, and (c) the `loop()` function, which will be executed continuously.

```
[for (node : IoTNode | aSystem.iotnode)]
 [for (controller : Controller | nodo.device)]
  [file (controller.name.toString().concat('.ino'
      ↪ ),false, 'UTF-8')]
   ...
   [for(acces: AccesPoint|controller.
       ↪ communication.accespoint)]
    void wifi() { ... }
   [/for]
   [for(mess: MessageBroker|controller.
       ↪ communication.messagebroker)]
    void mqtt() { ... }
   [/for]
   void actuators(String aux) { ... }
   void sensors() { ... }
   void setup() {
    ...
    [for(acces: AccesPoint|controller.
        ↪ communication.accespoint)]
     wifi();
    [/for]
    [for(mess: MessageBroker|controller.
        ↪ communication.messagebroker)]
     mqtt();
    [/for]
   }
   void loop() {
    Serial.flush();
    if(client.loop()){ sensors(); }
    else{ }
   }
  [/file]
 [/for]
[/for]
```

Listing 1: M2T transformation for the generation of Arduino code to implement the IoT nodes

Node-Red was used to implement the bridges, due to its simplicity to create publication/subscription flows with MQTT topics of the IoT nodes, REST clients to consume POST methods to store the information coming from the nodes, and REST services (GET and POST methods) to allow communication with the rest of orchestrators.

There are two types of Node-Red programs that will run on the platform: (a) `OutputBridge`, which allows to send information from a sensor MQTT message to an external web service via a REST client and (b) `InputBridge`, which allows to send information from an API REST to an actuator via an MQTT message coming from an external REST client. In addition, the two components consume the methods of the REST web services associated with each sensor or actuator. The structure of the Node-Red programs that will run in three sections can be seen in the code fragment of Listing 2: (a) Statement of the `OutputBridge`, and (b) Statement of the `InputBridge`.

```
[for (broker : MessageBroker | aSystem.
    ↪ messagebroker)]
 [for (out : OutputBridge | aSystem.messagebroker
     ↪ .bridge)]
  [file (out.sensor.name.toString().concat('.json
      ↪ '), false, 'UTF-8')]
['['/]
 ...
 {
  "id":"5395ff74.042a3",
  "type":"mqtt␣in",
  "z":"2fd02feb.4b81c",
  "name":"MQTT␣IN",
  ...
 }
 ...
 {
  "id":"3e91c700.00329a",
  "type":"http␣request",
  "z":"2fd02feb.4b81c",
  "name":"POST␣API␣RESTful",
  "method":"POST",
  ...
 }
  [for (uri:String|out.inputorchestrator.URI)]
 ,
 {
  "id":"268be48c.bb680[i/]",
  "type":"http␣request",
  "z":"2fd02feb.4b81c",
  "name":"POST␣ORQUESTATOR",
  "method":"POST",
  ...
   [/for]
  ...
  [']'/]
 [/file]
[/for]
[for (input : InputBridge | aSystem.
    messagebroker.bridge)]
[file (input.actuator.name.toString().
    concat('.json'),false, 'UTF-8')]
['['/]
 ...
 {
  "id":"d98a9e5d.a130c",
  "type":"http␣in",
  "z":"2fd02feb.4b81c",
  "name":"API␣REST␣(POST)",
  ...
 }
 ...
 {
  "id":"707fe980.de28c8",
  "type":"http␣request",
  "z":"2fd02feb.4b81c",
  "name":"POST␣API␣RESTful",
  "method":"POST",
  ...
 }
 ,
 {
  "id":"afd75f27.412b8",
  "type":"mqtt␣out",
  "z":"2fd02feb.4b81c",
  "name":"MQTT␣OUT",
  ...
 }
 ...
[']'/]
 [/file]
[/for]
```

Listing 2: M2T transformation for the generation of Node-Red code to implement the bridges

Ballerina was chosen for the creation of API RESTful services for each of the sensors or actuators. The structure of the services consists of three sections that can be seen in the code fragment of Listing 3: (a) Statement of the service URI, (b) Defining `POST` methods, recording information, `GET/{id}`, searching by ID, `GET/all`,

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

searching all records, `GET/last`, returning the last record, `PUT/{id}`, updating a record by ID, and `DELETE/{id}`, deleting a record by ID, and (c) Functions for Database management.

```
[for (webservice : WebService | aSystem.
    ↪ webservice)]
 [for (rest : REST | aSystem.webservice.rest)]
  [file (rest.device.name.toString().concat('.bal
      ↪ '),false, 'UTF-8')]
   ...
   @http:ServiceConfig {
    basePath: "/[rest.URI/]" }
   service LedData on httpListener {
    @http:ResourceConfig {
     methods: [ '[' /]"POST"[ ']' /],
     path: "/[rest.device.name/]/" }
    ...
    @http:ResourceConfig {
     methods: [ '[' /]"GET"[ ']' /],
     path: "/[rest.device.name/]/{id}" }
    ...
    @http:ResourceConfig {
     methods: [ '[' /]"GET"[ ']' /],
     path: "/[rest.device.name/]/all/" }
    ...
    @http:ResourceConfig {
     methods: [ '[' /]"GET"[ ']' /],
     path: "/[rest.device.name/]/last/" }
    ...
    @http:ResourceConfig {
     methods: [ '[' /]"PUT"[ ']' /],
     path: "/[rest.device.name/]/{id}" }
    ...
    @http:ResourceConfig {
     methods: [ '[' /]"DELETE"[ ']' /],
     path: "/[rest.device.name/]/{id}" }
    public function createTable() {
     string sqlString = "CREATE␣TABLE␣[rest.
         ↪ device.name/]␣(id␣int␣AUTO_INCREMENT,
         ↪ date␣varchar(20),␣time␣varchar(20),
         ↪ location␣varchar(20),␣attribute␣int,
         ↪ artefact␣varchar(20),␣PRIMARY␣KEY␣(id
         ↪ ));";
     var ret = deviceDB->update(sqlString);
    }
    ...
  [/file]
 [/for]
[/for]
```

Listing 3: M2T transformation for the generation of Ballerina code to implement the web services

The structure of the Orchestrators developed in Ballerina is observed in the code fragment of Listing 4, in which the program execution order is observed: (a) Declaring the URI of the Orchestrator; (b) Declaring the URI of the participating services; (c) Obtaining the information of the participating services; (d) Evaluating the logical function of the Orchestrator with the participants' information; and (e) Publishing the new state of the actuator or destination service.

```
[for (integration : IntegrationPattern | aSystem.
    ↪ integrationpattern)]
 [for (orchestrator : Orchestrator | integration.
     ↪ orchestrator)]
  [file (orchestrator.name.toString().concat('.
      ↪ bal'),false, 'UTF-8')]
   ...
   @http:ServiceConfig { basePath: "/[
       ↪ orchestrator.name/]" }
   service AsyncInvoker on asyncServiceEP {
    [for (func : Function | orchestrator.function
        ↪ )]
     @http:ResourceConfig {
      methods: ['['/]"POST"[']'/],
      consumes: ['['/]"application/json"[']'/],
      produces: ['['/]"application/json"[']'/]
     }
     resource function [func.inputorchestrator.
         ↪ URI/]
      (http:Caller caller, http:Request inRequest)
     returns error? {
      ...
      [for (rest : REST | orchestrator.rest)]
       [for (ws : WebService | aSystem.webservice
           ↪ )]
        [for (rest1 : REST | ws.rest)]
         [if (rest1.device.name = rest.device.
             ↪ name)]
          future<http:Response|error>
          fx[rest.device.name/]=start[rest.device
              ↪ .name/]
          EP->get("/last");
         [/if]
        [/for]
       [/for]
      [/for]
      ...
      boolean [orchestrator.name/]Function=[func.
          ↪ expression/];
      boolean [orchestrator.name/]Break=[func.
          ↪ break.expression/];
      ...
      if(![orchestrator.name/]Break) { ... }
      ...
     }
    [/for]
   }
  [/file]
 [/for]
[/for]
```

Listing 4: M2T transformation for the generation of Ballerina code to implement the orchestrators

4 A case study scenario

In order to test the functioning of our proposal, we will use the scenario presented for the introduction of the article motivation that is described in more detail in the smarthome Figure 5 that has been modeled with our tool. The scenario is implemented with two Controllers: a) One for the sensors of Light, Movement, Alarm, and b) One for the actuators, Light, Blind, and Sound. Six RESTful services have been created, one for each sensor and actuator. In addition, 3 InputBridge have been created for the sensors and 3 OutputBridge for the actuators. There are also 3 Orchestrators, one for each actuator, which establish the operating logic. Figure 6 shows the diagrams of the electrical circuit that is implemented for the IoT Nodes, in which the sensors in the 6.a and the actuators in the 6.b have been arranged.

In order to determine all possible cases, a truth table has been represented which shows the behavior of actuators when activated or deactivated (on/off) and sensors when it detects changes in the environment. Table 1 illustrates the conditions for the control of Light and Blind. The logical function of Light depends on the current status of the actuator and the latest status of the Alarm and LightSensor sensors. MovementSensor events act as a trigger to initiate the Orchestrator's operation. In the case of the Blind logic function, it depends on the current status of the actuator, and the latest status of the LightSensor sensor and the Light actuator. The events of the MovementSensor and the

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

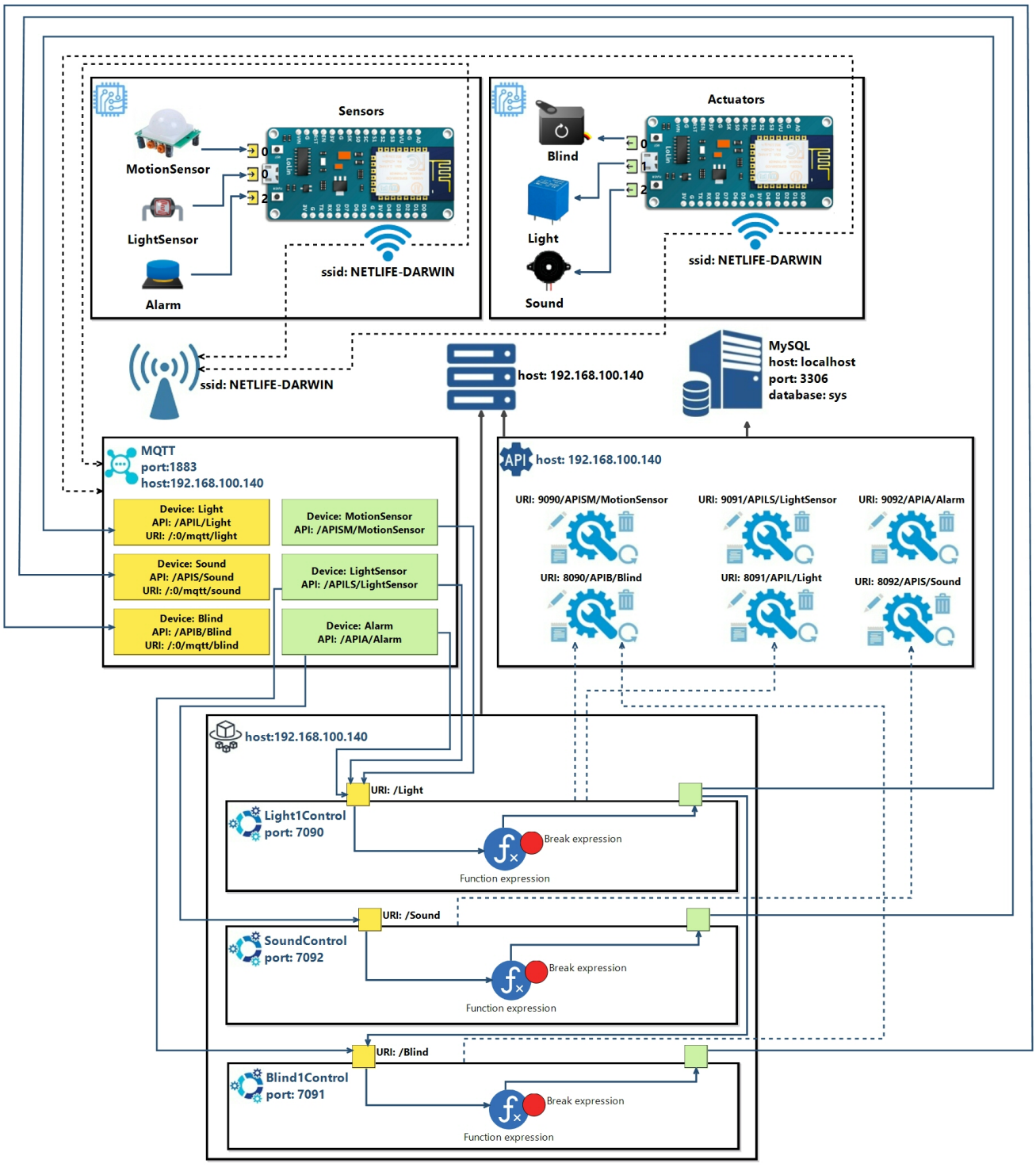
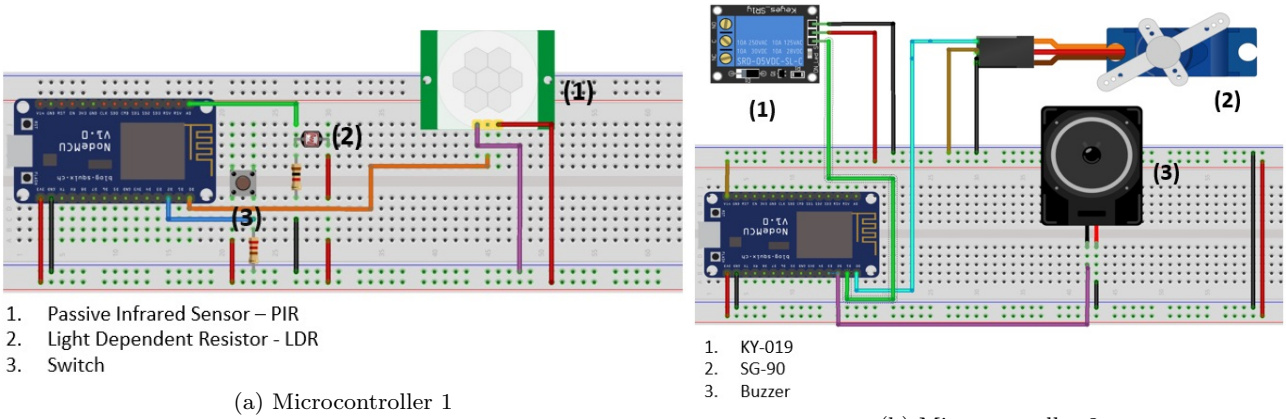


Fig. 5: System architecture for Smart Home

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.



(a) Microcontroller 1

(b) Microcontroller 2

Fig. 6: Electronic circuit (Sensors and Actuators)

B	L	A	LS	MS	FL	SL	FB	SB
1	1	1	1	1	0	1	0	1
1	1	1	1	0	0	1	0	1
1	1	1	0	1	0	1	0	1
1	1	1	0	0	0	1	0	1
1	1	0	1	1	0	0	0	1
1	1	0	1	0	0	0	0	1
1	1	0	0	1	0	1	0	1
1	1	0	0	0	0	0	0	1
0	0	1	1	1	1	0	1	0
0	0	1	1	0	1	0	0	1
0	0	1	0	1	1	0	0	1
0	0	1	0	0	1	0	0	1
0	0	0	1	1	0	1	1	0
0	0	0	1	0	0	1	0	1
0	0	0	0	1	1	0	1	0
0	0	0	0	0	0	1	0	1

Table 1: Blind and Light control function (1: True, 0: False, B: Blind, L: Light, A: Alarm, LS: Light Sensor, MS: Movement Sensor, FL: Function Light Control, SL: Status Light Control, FB: Function Blind Control, SB: Status Blind Control)

S	A	FS	SS
1	1	0	1
1	0	0	0
0	1	1	0
0	0	0	1

Table 2: Sound function (1: True, 0: False, S: Sound, A: Alarm, FS: Function Sound, SS: Status Sound)

Orchestrator for the Light control act as triggers to initiate the operation of the Orchestrator.

Table 2 illustrates the conditions for Sound control. The function depends on the current status of the actuator and the last status of the alarm. Alarm events act as a trigger to initiate the Orchestrator operation.

The expressions resulting from analyzing the truth tables are entered as parameters to the DSL meta-class *Function*, that algebraically represents the system behavior and *Status* meta-class, that represents when information should not be sent to the actuator. The Logic Function corresponds to the sum (OR) of products (AND) of the input functions according to the following expressions. According to the same logical function criteria, the statements for *Status* are determined according to the truth table. This statement allows you to determine whether the Orchestrator should send the information resulting from the evaluation of the logical function. Table 3 presents the three expressions for the business logic of the actuators.

```
BlindControlFunction = (!Blind * Light *
LightSensor * MovementSensor) + (!Blind * !Light
* LightSensor * MovementSensor) + (!Blind * !Light
* !LightSensor * MovementSensor)
```

```
LightControlFunction = (!Light * Alarm *
LightSensor * MovementSensor) + (!Light * Alarm *
LightSensor * !MovementSensor) + (!Light * Alarm *
!LightSensor * MovementSensor) + (!Light * Alarm *
!LightSensor * !MovementSensor) + (!Light * !Alarm
* !LightSensor * MovementSensor)
```

```
SoundControlFunction = (!Sound * Alarm)
```

Table 3: Business logic of the actuators

Figure 5 shows how Orchestrators control operation sequences. In this case the Bridge (InputBridge, OutputBridge) are the intermediaries with the MQTT message queues. In the case of the Light Control Orchestrator, the OutputBridge of the motion, light intensity and alarm sensors are the triggers that initiate the process and the new state resulting from evaluating the logic function is sent to the Light InputBridge. In the case of Blind Control, the trigger is the OutputBridge of

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

the light intensity sensor or the Light Orchestrator that starts the process and the new state resulting from evaluating the logic function is sent to Blind InputBridge. In the end the Sound control only depends on the Alarm InputBridge to determine the new state that is sent to the InputBridge Sound.

Let's imagine the motion sensor is activated (true). At that time, the status of the bulb is checked through the GET method of the API of the actuator (false). In addition, the state of the light sensor is checked to see if it is darkening (false). As a result, the user will observe that the blinds are opened (true) and that the bulb is turned on (true), because the new information is sent to change the actuator status. This scenario can be checked in the penultimate case of Table 1 or by evaluating the logical function of Light and Blind.

To make it easier to understand, the article has described the behavior of the devices using logical expressions and truth tables. However, all the behaviors of the case study included in this proposal have been defined using Business Process Model and Notation (BPMN) diagrams. Figure 7 shows briefly the behavior in charge of controlling the blind from the information obtained from: (1) the state of the light, (2) the light sensor and (3) the motion sensor. The rest of the BPMN material is available at the website of the research project that supports this proposal[3].

Our approach allows the design of scenarios in different fields, such as Smart Agro, Industry 4.0, Intelligent Transportation, among others. To exemplify the first mentioned field, we can consider an irrigation system that incorporates a series of sensors that monitor environmental conditions to determine when irrigation valves or ventilation shutters should be opened. In the second scenario, for example, sensors can be incorporated into a bottling plant in which the conveyor belts are controlled to monitor the stage in which the bottles are going and what process should be carried out, such as the entry of the order, the oven, the cooling and packaging. Finally, in the third area, modules can be designed that are incorporated into public transport units to monitor the opening of doors, number of passengers, location, whose information can feed an App that informs users.

4.1 Evaluation by function point estimation

Due to the nature of the proposed tool, what we need is to evaluate and validate the software devices generated by implementing the scenario proposed in the Introduction. In this case, a total of 3640 code lines have been generated in the following software devices: 2 Arduino programs with 369 code lines for the IoT Nodes, 6 Node-Red programs with 586 code lines for the Bridge, 6 Ballerina programs with 2100 code lines for the RESTful services and 3 Ballerina programs with 585 code lines for the Orchestrators.

To know the time that a single developer needs to implement the scenario, a first estimate has been made with a small group of experts (5 people) that on average have taken 2.5 hours for designing and building the scenario, without taking into account the deployment time. However, since at this stage no evaluation process has been done with a large team, a comparison has been made with Function Points (FPA) [27], which measures the functional size of the software from the customer's point of view.

For scenario analysis, the following is considered: (a) Sensor reading (EI, 3*6PF), (b) Actuator writing (EO, 3*5PF), (c) Database query with web services (EQ, 6*11PF), (d) Business logic file "Orchestrator" (ILF, 3*10PF). With these requirements the unadjusted function points is 129, with an adjustment factor of 17 (Data Communication: 4, Distributed Processing: 4, Online Data Entry: 5, Complex Processing: 1, Code Reusability: 1, Ease of Implementation: 1, Ease of Operation: 1). Finally, the value of the adjusted function points is 105 (129*(0.65+(0.01*17))). According to the values obtained, the number of man-hours required to implement the scenario with fourth-generation languages is 840 hours (105*8), and the estimated number of code lines is 2100 (105*20).

Therefore, the estimated time to implement the test scenario using our proposal is reduced to 0.29% of the value calculated with the Function Points. This means the scenario implementation can be done in 2.5 hours unlike the estimated 840 hours to perform the implementation manually. This is the main advantage of using MDE techniques that reduce the development time.

4.2 Performance and loading of the web services

To determine the performance of the RESTful web services created, Gatling[4] has been used, which is a load and performance test framework [18]. For the evaluation, 4 scenarios with 1,000, 2,000, 5,000 and 10,000 concurrent users have been proposed. In each scenario, each user makes 6 queries, one for each of the methods implemented (*i.e.*, `GET /uri/last`, `GET /uri/{id}`, `GET /uri/all`, `POST /uri`, `PUT /uri/{id}`, `DELETE /uri/{id}`) and each query separated by an interval of 5 seconds. Listing 5 shows the scenarios evaluated.

[3] `http://acg.ual.es/projects/cosmart/si4iot`

[4] Gatling official website – `https://gatling.io`

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

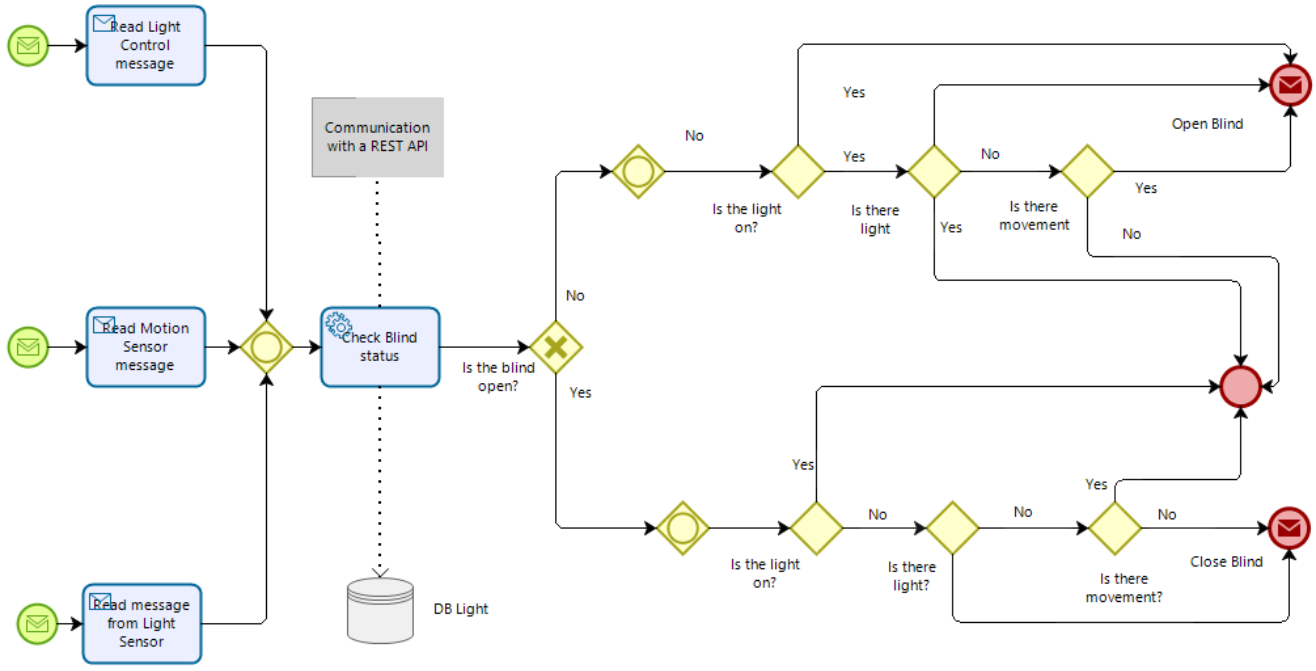


Fig. 7: Fragment of the BPMN diagram that describes the behavior of the blinds in the case study

```
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.jdbc.Predef._
class Test1 extends Simulation {
  val httpProtocol = http
    .baseUrl("...[host]...")
    .acceptHeader("*/*")
    .acceptEncodingHeader("gzip, deflate")
    .acceptLanguageHeader("es-ES,es;q=0.8,en-US;q
        ↪ =0.5,en;q=0.3")
    .userAgentHeader("Mozilla/5.0 (Windows NT
        ↪ 10.0; Win64; x64; rv:73.0) Gecko
        ↪ /20100101 Firefox/73.0")
 val headers_1 = Map("Content-Type" -> "text/
     ↪ plain;charset=UTF-8")
 val scn = scenario("Test1")
    .exec(http("request_0")
      .get("/...[uri].../last"))
    .pause(5)
    .exec(http("request_1")
      .post("/... [uri]...")
      .headers(headers_1)
      .body(RawFileBody("/test1/0001_request.txt"
          ↪ )))
    .pause(5)
    .exec(http("request_2")
      .get("/...[uri].../all"))
    .pause(5)
    .exec(http("request_3")
      .put("/...[uri].../66")
      .headers(headers_1)
      .body(RawFileBody("/test1/0003_request.txt"
          ↪ )))
    .pause(5)
    .exec(http("request_4")
      .get("/...[uri].../66"))
        .pause(5)
        .exec(http("request_5")
      .delete("/...[uri].../66")
      .headers(headers_1)
      .body(RawFileBody("/test1/0005_request.txt"
          ↪ )))
 setUp(scn.inject(atOnceUsers(...[#]...))).
     ↪ protocols(httpProtocol)
}
```

Listing 5: Gatling load test code

Figure 8 shows the result in each scenario, in terms of failed answers, and response times. The RESTful API created and implemented in a VPS (Virtual Private Server) with Windows Server 2012 is capable of handling more than 1,000 concurrent users without losing any requests. In the case of *request_2*, from Listing 5, it takes the longest to complete with a minimum delay of 1.8s and a maximum of 36.2s. In addition, 29% of queries are lost for the scenario with 2,000 users, while for the scenario with 5,000 users the losses increase to 71% and with 10,000 users this value rises to 85%. Even when query losses increase, communication is not completely lost. This is important in scenarios that have many users such as IoT, in which there are a large number of objects (sensors and actuators) connected.

5 Related Work

The presence of IoT systems in our surrounding is not just a trend, they are going to stay more prominently with us. This fact entails the need for developing proper structures that deal with the level of interoperability, integration, and communication that they require.

Many researchers work in that sense. In [14], the authors have already implemented and evaluated four interoperable standards for the Internet of Things (OGC PUCK over Bluetooth, TinySOS, SOS over CoAP, and OGC SensorThings API) and, summarizing, each of the protocols discussed has its own negative and positive at-

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

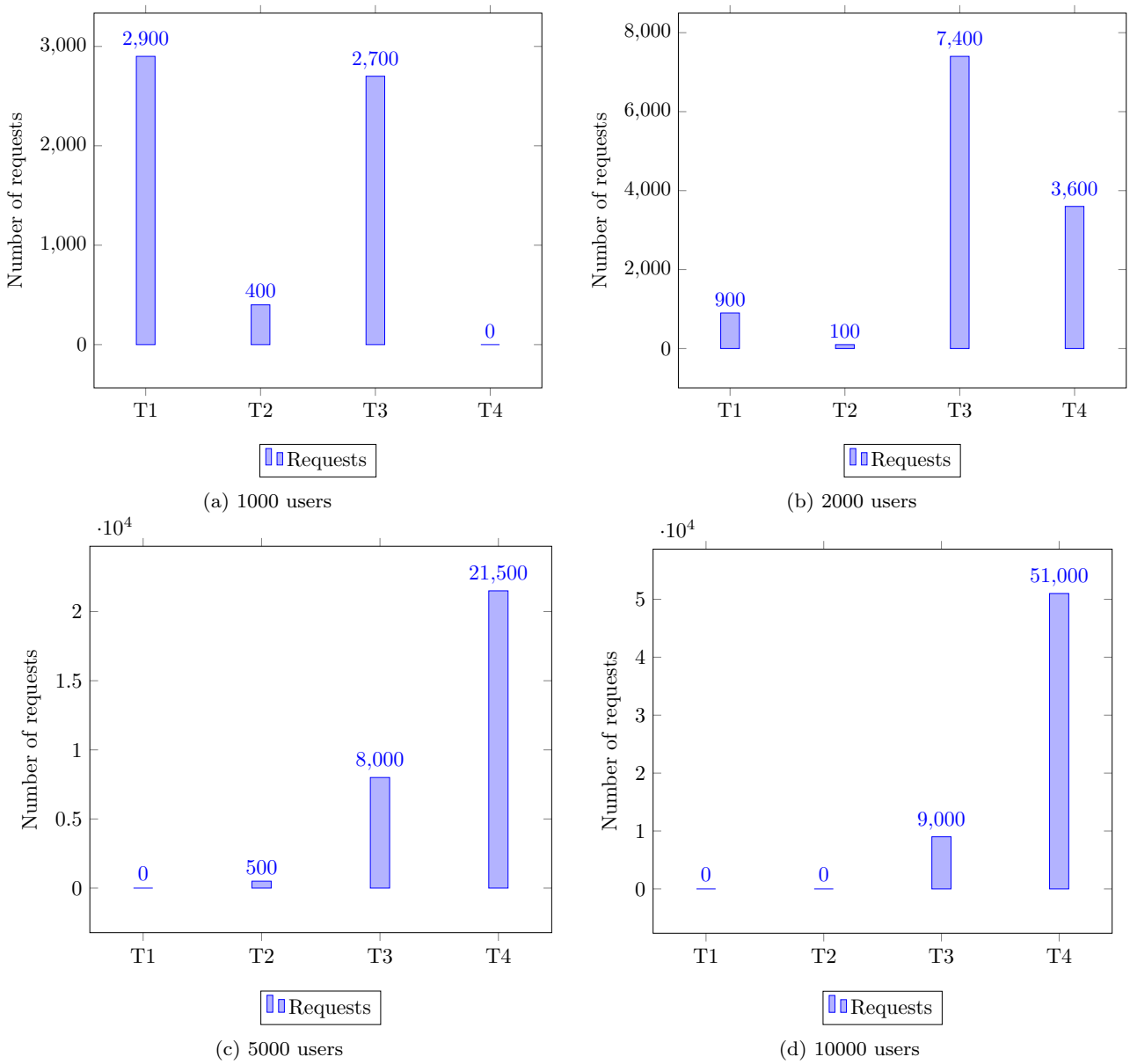


Fig. 8: Summary of results in all scenarios. (T1: t<800ms, T2: 800ms<t<1200ms, T3: t>1200ms, T4: Failed)

tributes. There is a huge scope of progress in this field that many researchers are addressing.

In works such as [10,15], the authors work with the goal of provide an (resource-oriented) architecture that allows the different components of IoT systems to work together to capitalize the computational units of smart agents providing orchestration to manage sensors, actuators, controllers in a coordinated manner in [10] and to execute operations of building, operating, and maintaining the IoT systems in [15].

In order to modelling the architecture some authors use BPMN (Business Process Model Notation) notation. In [19], the authors use BPMN as a general neutral platform to generate portable code, that allows user abstract from the hardware. In [9] the proposal relies on a Research Oriented Architecture (ROA) to provide uniform and structured communication with IoT devices using the BPMN notation to implement the processes without knowing much technical details.

The research oriented approach is widely used by IoT Systems, as previously seen in [15] and [9]. Other works, such as [2,34,35] make use as well of ROA architectures. In [35], the authors propose the use of microservices to improve traditional manufacturing environments to an IoT-based one (Industry 4.0), combining effectively MDE with IoT and the microservice ar-

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

Research Work	IoT-CPS	Services	Integration	MDE	CEP	BPMN
[10] [Darabseh et al., 2019]	✓				✓	
[15] [Kathiravelu et al., 2019]	✓	✓	✓		✓	
[19] [Martins et al., 2017]	✓			✓		✓
[35] [Thramboulidis et al., 2019]	✓		✓		✓	
[34] [Teixeira et al., 2017]	✓		✓		✓	
[2] [Al-Osta et al., 2018]	✓		✓		✓	
Our Proposal	✓	✓	✓	✓		✓

Table 4: Summary of the main features covered by the approaches of the related work

chitectural paradigm. In [34], the authors make a study of the works that pursue the simplification of the code-generation process, concluding that most of the proposals are based on MDE. In [2], the authors propose a decentralized architecture for IoT data through a two-layer data processing approach that employs Complex Event Processing (CEP) and Semantic Web techniques.

Model-Driven Engineering, present in our proposal, is also used in previously commented works such as [34, 35], always in connection with Service-Oriented Architectures but in [11], the authors make use of MDE to reduce the interoperability problem and reduce the development efforts towards ensuring that complex heterogeneous software systems interoperate with one another. It is important to highlight that in our proposal we use technologies, architectures, and tools widely accepted by the scientific community and fellow researchers. At the same time, none of the mentioned works propose a whole strategy including the definition of a Service-Oriented Architecture, Model-Driven Engineering, and Software-Defined Networking that deals with integration, complex events processing and interoperability in IoT and Cyber-Physical Systems.

In addition, a graphic editor has been designed and implemented that making use of Model to Text Transformations automatically generates code that provides compatibility with heterogeneous IoT devices such as sensors, actuators and controllers, providing an abstraction layer to developers that helps the maintenance of the systems, the integration of new technologies and increase the productivity and efficiency reducing errors.

Table 4 summarizes the main features investigated in our work in comparison with other similar approaches.

6 Conclusions and Future Work

This article proposes an approach to improve the design and implementation of services related to IoT devices. The main contributions of the approach are mainly related to the integration of these services and can be summarized as follows:

(a) We extend the capabilities of IoT nodes to allow communication based on publication and subscription in message queues.
(b) We propose a component that allows the connection between message queues and web services.
(c) We standardize the information format for IoT nodes by means of JSON objects.
(d) We associate the concept of RESTful web services to the IoT Nodes.
(e) We propose the integration of the IoT nodes by means of Orchestration patterns.

For that purpose, two software tools have been built, a graphic editor and an M2T transformation for the generation of code based on MDE in IoT systems. To achieve this goal, we have expanded the functionalities and capabilities of several of our tools introduced in previous work to make it easier to create applications, thus developers do not have to dig deeper into specific aspects of programming languages.

The approach allows developers to define the system using a graphical tool to standardize the structure of the IoT nodes, the links to the web services, the bridge to the MQTT protocol and the orchestrator to describe the business logic. As a result, we obtain a semi-automatic process for the generation of Ballerina code for RESTful web services and the orchestrator, and Arduino code for the deployment of IoT nodes.

The methodology includes a manual step before executing the application that involves the developer having to dispose of the infrastructure with the parameters used in the design, compilation and execution of the software devices created in each of the platforms. In addition, it must make the electrical connections of the components (sensors and actuators) in the controller.

For future work, the following lines of research have been identified: (a) extending the DSL to include possible errors in the operation flows in the web services, (b) integrating the digital TV and SmartPhone platforms in the IoT system, for the Front End, (c) apply load balancing strategies to handle large numbers of concurrent users, and (d) testing the operation of the

http://doi.org/10.1007/s10586-020-03150-x

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

tool with users from different degrees and levels related to computer science studies.

Acknowledgments

This work has been funded by the EU ERDF and the Spanish Ministry MINECO under the research project CoSmart TIN2017-83964-R and by the regional project (CEIJ-C01.2) coordinated from UAL-UCA and funded by CEIMAR consortium.

References

1. Ahmad, S., DoHyeun, K.: A Multi-Device Multi-Tasks Management and Orchestration Architecture for the Design of Enterprise IoT Applications. *Future Generation Computer Systems* 106:482–500. Elsevier (2020).
2. Al-Osta, M., Bali, A., Gherbi, A.: Event Driven and Semantic based Approach for Data Processing on IoT Gateway Devices. *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–16. Springer (2018).
3. Alulema, D., Criado, J., Iribarne, L.: A Model-Driven Approach for the Integration of Hardware Nodes in the IoT. *7th World Conference on Information Systems and Technologies* (CIST'2019), pp. 801-811 (2019).
4. Alulema, D., Criado, J., Iribarne, L.: RESTIoT. A Model-based Approach for Building RESTful Web Services in IoT Systems. *XXIV Jornadas de Ingeniería del Software y Bases de Datos* (JISBD), Pages 4. SISTEDES (2019).
5. Badamasi, Y.: The Working Principle of an Arduino. *11th International Conference on Electronics, Computer and Computation* (ICECCO), pp. 1–4, IEEE (2014).
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice.*Morgan & Claypool Publishers*, pp. 2019-2031 (2017).
7. Ciccozzi, F., Spalazzese, R.: MDE4IoT. Supporting the Internet of Things with Model-driven Engineering. *International Symposium on Intelligent and Distributed Computing*, pp. 67-76. Springer (2016).
8. Costa, B., Pires, P. F., Delicato, F. C., Merson, P.: Evaluating REST Architectures. Approach, Tooling and Guidelines. *Journal of Systems and Software*, 112(February):156-180. Elsevier (2016).
9. Dar, K., Taherkordi, A., Baraki, H., Eliassen, F., Geihs, K.: A Resource Oriented Integration Architecture for the Internet of Things. A Business Process Perspective. *Pervasive and Mobile Computing*, 20:145–159. Elsevier (2015).
10. Darabseh, A., Freris, N.: A Software-defined Architecture for Control of IoT Cyberphysical Systems. *Cluster Computing*, pp. 1-16, on-line. Springer (2019).
11. Grace, P., Pickering, B., Surridge, M.: Model-driven Interoperability. Engineering Heterogeneous IoT Systems. *Annals of Telecommunications*, 71(3–4):141–150. Springer (2016).
12. Gronback, R. C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. *Pearson Education*, pages 736 (2009).
13. Hwang, G., Lee, J., Park, J., Chang, T. W.: Developing Performance Measurement System for Internet of Things and Smart Factory Environment. *International Journal of Production Research*, 55(9):2590-2602. Taylor & Francis (2017).
14. Jazayeri, M., Liang, S., Huang, C.: Implementation and Evaluation of Four Interoperable Open Standards for the Internet of Things. *Sensors*, 15-9:1424-8220. MDPI (2015).
15. Kathiravelu, P., Van Roy, P., Veiga, L.: SD-CPS. Software-defined Cyber-physical Systems. Taming the challenges of CPS with workflows at the edge. *Cluster Computing*, 22(3):661–677. Spriger (2018).
16. Kovatsch, M., Matsukura, R., Lagally, M., Kawaguchi, T., Toumura, K., Kajimoto, K.: Web of Things (WoT) Architecture, W3C Recommendation (2020).
17. Limon, X., Guerra-Hernandez, A., Sanchez-Garcia, A. J., Perez Arriaga, J.: SagaMAS. A Software Framework for Distributed Transactions in the Microservice Architecture. *6th International Conference in Software Engineering Research and Innovation* (CONISOFT'2018), pp. 50–58. IEEE (2019).
18. Maila-Maila, F., Intriago-Pazmiño, M., Ibarra-Fiallo, J.: Evaluation of Open Source Software for Testing Performance of Web Applications. Advances in Intelligent Systems and Computing, 931:75–82. Springer (2019).
19. Martins, F., Domingos, D.: Modelling IoT Behaviour within BPMN Business Processes. *Procedia Computer Science*, 121:1014–1022. Elsevier (2017).
20. Martin-Lopo, M., Boal, J. Sánchez-Miralles, A.:A literature review of IoT energy platforms aimed at end users. *Computer Networks*, 17:1-19. Elsevier (2020).
21. Mesfin, G., Gronli, T.-M., Midekso, D., Ghinea, G.: Towards end-user Development of REST Client Applications on Smartphones. *Computer Standards & Interfaces*, 44:205-219. Elsevier (2016).
22. Mineraud, J., Mazhelis, O., Su, X., Tarkoma, S.: A Gap Analysis of Internet-of-Things Platforms. *Computer Communications*, 89:5-16. Elsevier (2016.)
23. Muehlen, M., Nickerson, J., Midekso, D., Ghinea, G.: Developing Web Services Choreography Standards. The Case of REST vs. SOAP. *Decision Support Systems*, 40(1):9-29. Elsevier (2005).
24. Pautasso, C., Wilde, E., Alarcon, R.: REST: Advanced Research Topics and Practical Applications. Pages 222. *Springer* New York (2014).
25. Richardson, C.: Microservices Patterns. *Manning Publications Co.*. Pages 520 (2018).
26. Rotsos, C., King, D., Farshad, A., Bird, J., Fawcett, L., Georgalas, N., Gunkel, M., Shiomoto, K., Wang, A., Mauthe, A., Race, N., Hutchison, D.: Network Service Orchestration Standardization: A Technology Survey. *Computer Standards & Interfaces*, 54:203-215. Elsevier (2017).
27. Shah, J., Kama, N.: Extending Function Point Analysis Effort Estimation Method for Software Development Phase. 7th International Conference on Software and Computer Applications ICSCA, pp. 77–81. ACM (2018).
28. Sharma, S., Chang, V., Tim, U. S., Wong, J., Gadia, S.: Cloud and IoT-based Emerging Services Systems. *Cluster Computing*, 22(1):71-91. Springer (2019)
29. Silva, B., Murad, K., Kyuchang, L., Yongtak, Y., Diyan, M., Jihun, H., Kijun, H.: RESTful Web of Things for Ubiquitous Smart Home Energy Management. International Conference on Computing, *Networking and Communications*, ICNC 2020, pp. 176–180. IEEE (2020).
30. Slama, D., Puhlmann, F., Morrish, J., Bhatnagar, R. M.: Enterprise IoT: Strategies and Best Practices for Connected Products and Services. *O'Reilly Media, Inc.*, Pages 474 (2015)
31. SOAP (Simple Object Access Protocol) W3C Standard. 2007. `https://www.w3.org/TR/soap12/`. Online: last accessed September 2019.

D. Alulema, J. Criado, L. Iribarne, A.J. Fernández-García, R. Ayala.(2020)A model-driven engineering approach for the service integration of IoT systems. Cluster Computing 23:1937–1954, Springer.ISSN: 1386-7857.

32. de Sousa, N. F. S., Perez, D. A. L., Rosa, R. V., Santos, M. A., Rothenberg, C. E.: Network Service Orchestration: A Survey. *Computer Communications*, 142/143:69-94. Elsevier (2019).
33. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF. Eclipse Modeling Framework. *Pearson Education*, Pages 749 (2009).
34. Teixeira, S., Agrizzi, B. A., Filho, J. G. P., Rossetto, S., Baldam, R. de L.: Modeling and Automatic Code Generation for Wireless Sensor Network Applications using Model-driven or Business Process Approaches. A Systematic Mapping Study. *Journal of Systems and Software*, 132:50–71. Elsevier (2017).
35. Thramboulidis, K., Vachtsevanou, D. C., Kontou, I.: CPuS-IoT. A Cyber-Physical Microservice and IoT-based Framework for Manufacturing Assembly Systems. *Annual Reviews in Control*, 47:237-248. Elsevier (2019).
36. Yassein, M. B., Shatnawi, M. Q., Aljwarneh, S., Al-Hatmi, R.: Internet of Things. Survey and Open Issues of MQTT Protocol. International Conference on Engineering & MIS (ICEMIS), pp. 1-6. IEEE (2017).
37. Zhou, C., Feng, Y., Yin, Z.: An Algebraic Complex Event Processing Method for Cyber-physical System. *Cluster Computing*, 3:1–9. Springer (2018).

http://doi.org/10.1007/s10586-020-03150-x