A DEVS-Based Methodology for Simulation and Model-Driven Development of IoT

Iman Alavi Fazel(✉) and Gabriel Wainer

Carleton University, 1125 Colonel By Dr, Ottawa, ON K1S 5B6, Canada
{imanalavifazel,gwainer}@cmail.carleton.ca

Abstract. The Internet of Things (IoT) has emerged as a promising technology with diverse applications across industries, including smart homes, healthcare services, and manufacturing. However, despite its potential, IoT presents unique challenges, such as interoperability, system complexity, and the need for efficient development and maintenance. This paper explores a model-driven development (MDD) approach to design IoT applications by employing high-level models to facilitate abstraction and reusability. Specifically, we adopt a methodology based on Discrete Event System Specification (DEVS), a modular and hierarchical formalism for MDD of IoT. In our work, different DEVS models are developed to address distinct functional aspects of the devices, encompassing data retrieval, data serialization/deserialization, and network connectivity. The developed models, along with a DEVS simulator, are then used for both simulation and deployment. To create a comprehensive simulation environment, the paper introduces two additional models for simulating the MQTT protocol, including its Quality of Service (QoS) mechanism.

Keywords: IoT · DEVS · Model-driven development · MDD

1 Introduction

The Internet of Things (IoT), characterized as an internet-accessible network of sensors and actuators, has emerged as a promising solution in numerous areas. In addition to its established application within home automation, IoT has been used in sectors such as healthcare, supply chains, and manufacturing [1]. What is referred to as the fourth industrial revolution, or Industry 4.0, is backed by IoT technologies which results in more autonomy and enhanced efficiency of industrial plants and processes.

Due to the nature of these systems, it is challenging to design, implement, and verify their components and their interconnections, in particular interoperability, the ability of these systems to properly work and communicate together [2]. In addition, there are various difficulties related mainly to the distributed nature, heterogeneity, and the presence of "human-in-the-loop" in these systems [3]. To help with the design, different life cycle phases should be automated, as manual efforts for development, deployment, and maintenance of the plethora of devices are prone to errors [4].

J.-L. Guisado-Lizar et al. (Eds.): SIMUtools 2023, LNICST 519, pp. 3–17, 2024.
https://doi.org/10.1007/978-3-031-57523-5_1

Model-driven development (MDD) approaches can overcome these challenges. Using these methods, the code for devices is designed using high-level models instead of platform-specific programs. These models can be expressed in various formats such as graphical and textual or with formal or informal semantics. However, the goal of all of them is to create an abstraction over specific code implementation. After their design and analysis, these models will eventually be transformed into executable code that can be run on the hardware. In MDD, the use of high-level models provides *abstraction*, *separation of concern,* and *reusability* to the development cycle [5, 6].

This paper presents a methodology based on Discrete Event System Specification (DEVS) for model-driven development of IoT devices. To achieve this, a series of DEVS models were developed, each performing a specific functional aspect of the devices. These tasks encompassed capabilities such as retrieving data from ADC channels, serializing/deserializing the data (such as to and from JSON or XML), and providing network connectivity. These models, alongside a DEVS simulator, would then be flashed onto the devices for deployment. We developed prototype models for the widely used ESP32 microcontrollers, but they can be extended to other platforms. For simulation purposes, we include two models for MQTT brokers and clients, replicating MQTT at a high level (including its acknowledgment mechanism for Quality of Service - QoS). The models for MQTT communication facilitate the collection of information about network traffic and the feasibility of an event within a specified time frame.

An advantage of using DEVS lies in the ability to reuse models with similar interfaces and behavior, for both simulation and deployment on the device. Furthermore, if the DEVS models perform hardware-independent tasks, the same implementation code for the models can be used for both scenarios. Utilizing DEVS for model-driven development additionally enables us to leverage a set of techniques and methods that have been developed for the verification and validation of their behavior [7–9].

The rest of the paper is organized as follows. In Sect. 2, some previous work on model-driven development of IoT alongside a short description of DEVS is presented. Section 3, provides the DEVS specification of the developed models. A simple case study comprising of moisture sensor and an irrigation system is presented in Sect. 4. In Sect. 5, the implementation of the DEVS models is discussed. Lastly, Sect. 6, summarizes the paper and discusses future directions.

2 Related Work

High-level models have found diverse applications in the design and development of IoT technologies. These models have been employed to capture the characteristics of specific components of the infrastructure, such as network connectivity and resource allocation schemes, as well as to model the complete behavior of a node. The models were then used for purposes such as simulation, verification, and code generation. In this section, we present some of the previous research that used high-level models in the context of IoT [10].

Authors in [11] developed a framework called IFogSim, which used models of sensors and actuators to simulate different resource management strategies within the fog computing paradigm. This framework is an extension of CloudSim and is based on

discrete-event simulation (DES). Another CloudSim-based simulator [12] considers the big data aspect of these devices using MapReduce. SimTalk [13], a simulation software, facilitates the emulation of environments that can be a combination of virtual and interconnected IoT devices using graphical representations of the actuating systems in the network. A survey by [14] presents 31 simulators for Wireless Sensor Networks (WSNs) which employ a form of model for the sensor nodes, propagation medium, or communication technologies.

The use of models in the form of model-driven development (MDD) of IoT applications was the focus of other works in the literature. The authors used various kinds of models such as the one based on visual notations, to textual formats with domain-specific languages (DSLs) to design IoT applications. These models also differed in their semantics being formal, expressed as mathematical techniques, or informal, such as the one that uses plain language [15]. Some research in this category further used models for verification using methods such as simulation and model checking to ensure the correct behavior of the system. In what follows, some of these works are presented.

In [16], the authors presented a DSL that allows developers to define ports, properties, and Statecharts in text-based format, and later use a set of tools to automatically transform models and generate code. In [17], a UML-based approach was used to generate wrappers, enabling the integration of diverse IoT elements. Research in [5] introduced a framework to enable node-centric and rule-based programming through a DSL, providing reusability, flexibility, and maintainability. Authors in [18] presented a method for designing and analyzing IoT applications to verify their correctness and QoS using SysML4IoT, a framework consisting of a SysML profile and a model-to-text translator that converts the models for a model checker.

In our work, we applied the DEVS formalism for model-driven development and simulation of IoT devices. DEVS can be viewed as finite-state machines where transitions between states occur based on new input to the system or expiration of their lifespan [19]. A model of interest can be broken down into atomic models, each responsible for specific behavior. The atomic models can then be coupled together to create the complete model of interest. The modular nature of DEVS enables us to design and test atomic models individually, and then integrate them with the other models.

Some authors have previously applied DEVS formalism to the IoT domain. In a study by [20], the authors proposed methods to simulate moving IoT nodes more efficiently, and used DEVS for modeling the wireless communications aspect of the devices. In [21], a novel approach was introduced, aimed at improving energy efficiency in smart buildings based on user location. Authors in [22] used DEVS for modeling botnets using Markov Chains. Their model behaved similarly to the spread of Mirai and Torii botnets. A DEVS model of smart home networks, was used to determine the optimal working hours for energy consumption [23], as well as a DEVS-based IoT management system that provides users with metrics of power consumption [24]. Research in [25] explored a simulation acceleration method for a DEVS-based hybrid system using multiple CPUs and GPU cores. Their simulation environment was a fire-spreading application comprised of IoT sensor networks. DEVS was also used to model a Fog computing environment and showed that combining fog and cloud computing can enhance the user experience by offloading tasks to the fog nodes [26].

Our work differs from the previous research in that we employed DEVS to develop models for simulation, and ultimately for deployment on the devices. We also modeled the MQTT protocol using DEVS, another key contribution of this work.

3 Methodology

As discussed earlier, our goal was to apply DEVS for the simulation as well as the operation of IoT devices. To achieve this, we developed two sets of DEVS models with identical interfaces, state variables, and state transitions. One set of models was designed to be used for the simulation environment, and the other set for deployment. Moreover, any model that performed a hardware-agnostic task was reused in both sets. For simulating the DEVS models, we adopted Cadmium, a header-only C++ library[1]. Cadmium has the advantage of having a real-time version, which we later used to execute the DEVS model on the IoT device. In this study, we chose the ESP32 microcontroller for devices which is a popular choice in the industry.

3.1 Deployment Models

ADC Model
ADC is a model that periodically reads the ADC channel of the device and transmits its value through an output port, defined as a floating-point number proportional to the analog input signal, depicted in Fig. 1. This model can optionally receive a Boolean input to toggle its state between outputting data or becoming passive.

Fig. 1. The ADC atomic model.

Wrapper Model
The Wrapper model receives raw data and creates a set of values consisting of a value (for instance, from the ADC), an ID, and a type of channel. Then, it sends this set as its output. The input and output ports of this model are depicted in Fig. 2.

Fig. 2. The Wrapper atomic model.

[1] Source code for the Cadmium library can be found at https://github.com/SimulationEverywhere/cadmium.

In the implementation, we used instances of a user-defined C++ class named *Message* to represent this information. For example, an object of this type could be comprised of the following fields:

- data (as Double): **24.0**
- id (as String): "**adc1**"
- type (as String): "**temperature**"

Base Model

The Base model performs the main computation of the IoT device. For instance, for IoT sensor nodes, this model is responsible for collecting different ADC channels and creating a final output to be later transmitted over the network. In other applications, this model can receive commands that have been transmitted from the network and control the connected actuators to the board. Hence, the definition of the internal and external transition functions of this model is dependent on the specific application. However, its interface, i.e., the input and output ports, is the same among its different applications. This model has two sets of input and output ports. One set of ports is defined to provide communication with ADC and/or actuator models of the device, and the other set is for sending and/or receiving objects of type *MQTTMessage*. The fields of this type contain the data (as a JSON string), the MQTT topic, and the desired QoS level for the MQTT message. For instance:

- data (as String): **{data: 25.4, type: "temperature", unit: "C"}**
- topic (as String): **/home/garden/sensor1**
- QoSLevel (as Integer): **2**

In an example of the sensor node, this Base model can receive sensor readings for one or more ADC channels and then perform some form of sensor fusion algorithm. This operation can involve simple averaging or any other complex computations. Eventually, the result of the computation is set as the *data* field of the *MQTTMessage* object out of the model.

The input and output ports of this model are depicted in Fig. 3.

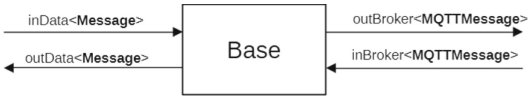

Fig. 3. Base DEVS model

Connection Model

The Connection model provides network connectivity for the board. This model receives *MQTTMessage* objects as input, which are produced by the Base model, and then transmits them to an MQTT broker accessible on the network. Conversely, it can also receive data from an MQTT Broker and dispatch as *MQTTMessage* instances to the Base model. Moreover, on the ESP32 board, the implementation of this model causes the device to initially connect to a Wi-Fi Access Point during initialization.

3.2 Simulation Models

In what follows, the descriptions of the equivalent simulation models are discussed. The ADC model in the simulation environment has the same specification as the deployment model, except it produces pseudo-random numbers in its output function. The Wrapper and Base models are identical for both simulation and deployment. For the Connection model, a coupled model mimics the functionality of an MQTT client in the simulation environment including the state changes in its connections. This model was complemented by the Network Medium and MQTT Broker model. The Network Medium model is a queueing network model that captures the latency of the network transmission, and the MQTT broker provides communication for multiple MQTT clients connected to it. The subsequent sections provide details about these models.

MQTT Client
The MQTT Client establishes communication with the device model on one end and with the MQTT broker on the opposite one. The type of data exchanged with the device is *MQTTMessage*, as discussed in the previous section, while with the broker, it is of type *MQTTPacket*. Objects of type *MQTTPacket* contain fields such as the packet ID and packet type, in accordance with the MQTT protocol specification. The exchange of objects of this type allows us to perform proper state changes similar to how they occur in actual implementations of this protocol. *MQTTPacket* objects are comprised of:

- **type** (as an MQTTPacketType): Specifies the type of the MQTT packet, such as PUBLISH, PUBREL, PUBREC, etc.
- **body** (as a map of String to String): Depending on the type of the packet, maps the required fields, such as “packetID” and “Payload”, to their corresponding values.
- **newPublishPacket** (as a Boolean): A flag that designates whether the packet is a newly created PUBLISH packet for the client, or it is a PUBLISH packet that was received from the broker.

For instance, an MQTTPacket object has the following fields:

- **type**: PUBLISH
- **body**:

```
{
  {"packetID", 100-},
        {"topic", "/home/kitchen/sensor"}
        {"DUPFlag", true}
        {"QoSLevel", 2}
        {"retain", true}
        {"payload", "{ data: 10, sensor_type: \"temperature\"}"}
}
```

- **newPublishPacket**: true

The MQTT client is a coupled model composed of two atomic DEVS models: the *Base* and the *Acknowledgement Buffer*. The Base model transmits and receives MQTT

packets to and from the Broker. In this process, whenever a packet with a QoS level of 1 or 2 is received, the Base model also sends a copy of the packet to the Acknowledgement Buffer model. The Acknowledgment Buffer stores information about packets and their connection states by maintaining a variable that maps each packet ID to its current connection phase. Additionally, this model keeps track of the last timestamp at which the packets arrive. Whenever the timer for a particular packet expires, the Acknowledgment Buffer generates a new acknowledged packet and sends it to the Base model. The Base model will then forward any received packets from the Acknowledgment Buffer to the clients. The MQTT client model is depicted in Fig. 4.

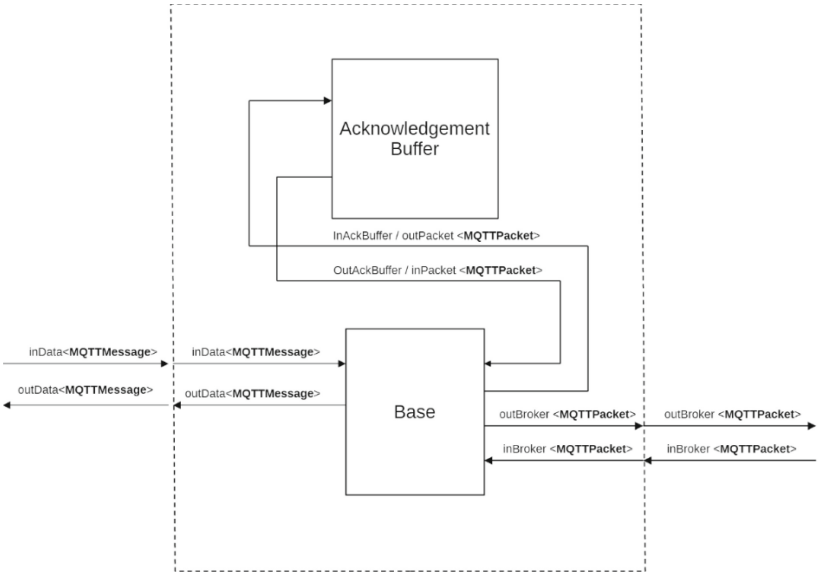

Fig. 4. MQTTClient DEVS model

Network Medium

The network medium is connected by an array of input and output ports to the MQTT clients and by one set of input and output ports to the broker. The model receives *MQTTPacket* objects from clients and then adds them to a queue. The packets in this queue will be dispatched after a certain time interval, which is determined by the exponential distribution function. This model sends objects of type *BrokerMessage* to the MQTT broker, which contains the *MQTTPacket* objects and the index of the input array in which the packets have arrived. Similarly, the Network Medium model can receive objects of type *BrokerMessage* from the MQTT broker model, which, after the expiry of the service time, will be forwarded to the intended client. The MQTT broker distinguishes between different clients based on the index of the input/output array with which they communicate. Therefore, during the simulation, clients should remain connected to specific input and output ports of the model (Fig. 5).

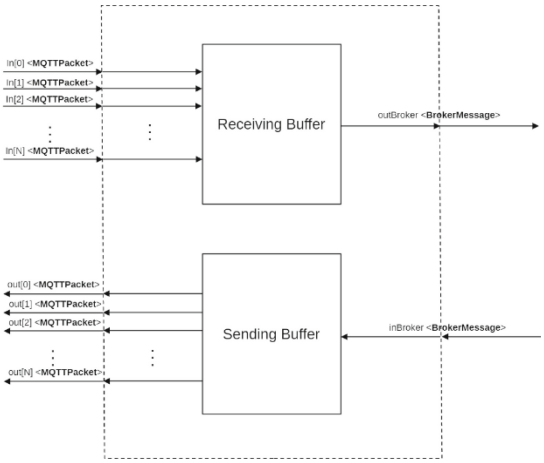

Fig. 5. Network Medium DEVS model.

MQTT Broker

The MQTT Broker receives *BrokerMessage* objects from the Network Medium model. Depending on the packet type, it performs the appropriate connection state change or broadcasts it to different clients. Similar to the MQTT client model, the MQTT Broker is a coupled model comprised of two atomic models: the *Base* and the *Acknowledgment Buffer*. The Base model stores the list of topics to which clients are subscribed and updates this list upon receiving a new SUBSCRIBE packet. Alternatively, when a PUBLISH packet arrives, the Base model inspects the topic of the packet and broadcasts a set of PUBLISH messages to every client that was previously subscribed to that topic. Other types of packets, including the newly generated PUBLISH packets with QoS levels 1 or 2, are then forwarded to the Acknowledgment Buffer. The Acknowledgment Buffer, in turn, updates the connection state for the packet ID, generates an appropriate acknowledgment packet, and sends it to the Base model. The Base model simply forwards any packet it receives from the Acknowledgment Buffer to the clients. In the Acknowledgment Buffer, a state variable is also present to keep track of the last timestamp when the packet was received. Upon its expiry, the model regenerates the acknowledgment packet for the Base model. The input and output ports of this model are depicted in Fig. 6.

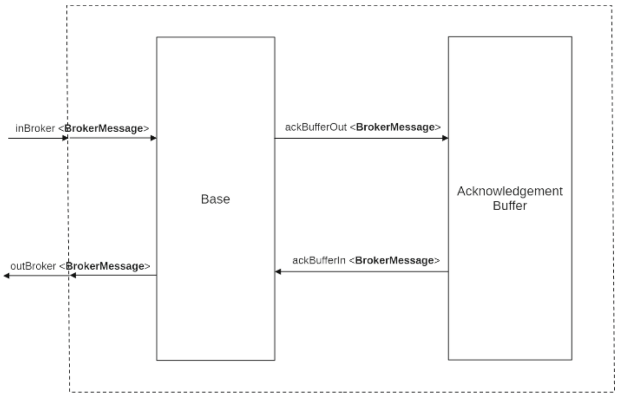

Fig. 6. MQTTBroker DEVS model

4 Implementation

To implement the DEVS model in Cadmium, the first step was defining the model's state using a user-defined C++ type. For instance, the state for the Sensor nodes in our implementation was defined as:

```
struct SensorState {
  double sigma;
  ConnectionPhase phase;
  Message<double> newMessage;
  vector<string> topicsToSubscribe;
  vector<string> topicsToPublish;

  SensorState(vector<string> topicsToPublish,
   vector<string> topicsToSubscribe = {})
    : sigma(std::numeric_limits<double>::max()),
     newMessage(),topicToPublish(topicToPublish),
      listOfTopicsToSubscribe(listOfTopicsToSubscribe) {
       if(listOfTopicsToSubscribe.size() > 0) {
         phase = ConnectionPhase::
               DATA_AVAILABLE_FOR_SUBSCRIBE;
         sigma = 0;
       }
}
```

Where *ConnectionPhase* is an *enum* with the following definition:

```
enum class ConnectionPhase {
   IDLE,
   DATA_AVAILABLE_FOR_PUBLISH,
   DATA_AVAILABLE_FOR_SUBSCRIBE
};
```

An instance of *SensorState* holds information about sigma, a variable that was used in the time advance function, its connection phase, and other data used as state variables in this model. After defining these data types for different states, the DEVS models were created by declaring classes inherited from Cadmium's Atomic template class. In the case of the Sensor model, the class was declared as:

```
class Sensor : public Atomic<SensorState>
{ /* …  */ }
```

To complete the definition of the DEVS model, the input and output ports were then defined, and the virtual member functions of the *Atomic* class, such as the internal and external transition were overridden. The type of Port objects specified using a template argument determines the values they send or receive. These types can be primitive types, such as double and int, or any other user-defined types. In the example of the Sensor model, these ports were defined as:

```
Port<MQTTMessage> outMQTT;
Port<MQTTMessage> inMQTT;
```

In our implementation, a list of ports that send and receive objects of the same type was represented using a 'std::vector' of ports. For example, the ports of the buffer models in the Network Medium model were declared as:

```
std::vector<Port<MQTTPacket>> receivingBufferIn;
std::vector<Port<MQTTPacket>> sendingBufferOut;
```

After defining the ports, the models were coupled together using Cadmium's *addCoupling* function. Furthermore, to facilitate the coupling between the MQTT client and broker, a function was defined that kept track of the coupled ports and assigned free ones to the models. The signature of the function was as follows:

```
void assignFreePort(std::shared_ptr<MQTTClient> mqttClient,
    std::shared_ptr<MQTTBroker> mqttBroker,
          std::shared_ptr<PropagationMedium> networkMedium);
```

The final step was to create a top model consisting of all inner submodels coupled together. Then, to execute the models in the simulation, an object of type *RootCoordinator* was instantiated, and the start() member function was called. For instance:

```
auto model = make_shared<TopLevelModel>("top");
auto rootCoordinator = cadmium::RootCoordinator(model);
auto logger = make_shared<cadmium::CSVLogger>("log.csv", ";");
rootCoordinator.setLogger(logger);
rootCoordinator.start();
rootCoordinator.simulate(std::numeric_limits<double>::
                                                    infinity());
rootCoordinator.stop();
```

In contrast, if we wanted to run the models for their operation on the devices, an object of *RealTimeRootCoordinator* was created. This was done via:

```
auto realTimeRootCoordinator =
  cadmium::
  RealTimeRootCoordinator
  <cadmium::ChronoClock<std::chrono::steady_clock> >
                                                (model, clock);

  realTimeRootCoordinator.start();
  realTimeRootCoordinator
         .simulate(std::numeric_limits<double>::infinity());
  realTimeRootCoordinator.stop();
```

5 Measuring Levels of Soil Moisture

A case study was conducted in which a sensor node periodically captured the moisture levels of the soil using its ADC channel, and an actuator that triggered the irrigation system when the moisture fell below a specified threshold. Communication between the sensor and the actuator was established using the MQTT protocol. Before deployment on the boards, these models were executed in a simulation environment. This section provides information about the simulation setup and presents the results. Furthermore, to complete the simulation environment, an additional DEVS model was developed to simulate the behavior of gardening soil. This model stored the moisture value, as well as transitions to change this value. The sensor node was connected to this model to retrieve moisture readings, while the actuator sent irrigation commands to this model. Specifically, when the actuator initiated a command to start irrigation, the model's internal transition function consistently incremented the moisture level, unless a subsequent signal was received, indicating the termination of irrigation. The input and output ports of the soil model are depicted in Fig. 7.

At the beginning of the simulation, the actuator first subscribes to the topic */garden/moisture_sensor*, and the sensor model starts reading the moisture value from the soil model and publishes them to the same topic. Whenever the moisture level falls below 15, the actuator sends a signal to the soil model to initiate the irrigation system.

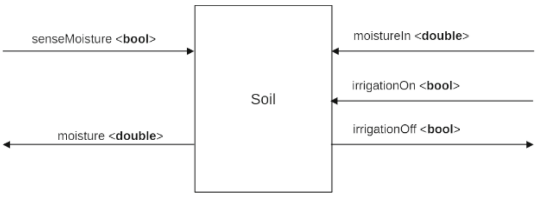

Fig. 7. Soil DEVS model

5.1 Simulation Result

The simulation ran for 10 sensor readings. To showcase the packet retransmission mechanism, an additional simulation was run with a condition added to a buffer of the Network Medium, causing intentional packet drops based on a certain probability. Namely, packets with a packet ID divisible by 3 had a 50% chance of being dropped. Table 1 depicts the timestamp of the PUBCOMP packets being received by the Broker when data was being published to the actuator model in both simulation runs.

Table 1. Timestamp of the received PUBCOMP message from the sensor node

Packet ID	Timestamp of the PUBLISH	Timestamp of PUBCOMP without retransmission	Timestamp of PUBCOMP with retransmission
1	1	1.55052	1.48447
2	2	2.47999	2.91148
3	3	3.60933	7.56755
4	4	4.52406	4.38234
5	5	5.74917	5.60835
6	6	6.59165	11.7505
7	7	7.65689	8.83374
8	8	8.38723	8.97219
9	9	9.59758	11.5696
10	10	10.2355	11.7876

As Table 1 shows, the PUBCOMP packets that required retransmission were received by the broker with a significant delay. The total number of packets exchanged between models is shown in Fig. 8.

By applying realistic conditions for packet drops, we can estimate the arrival of the packets at the destination. Furthermore, the total number of packets that need to be transmitted to and from the device can be used to estimate the power consumption of the devices.

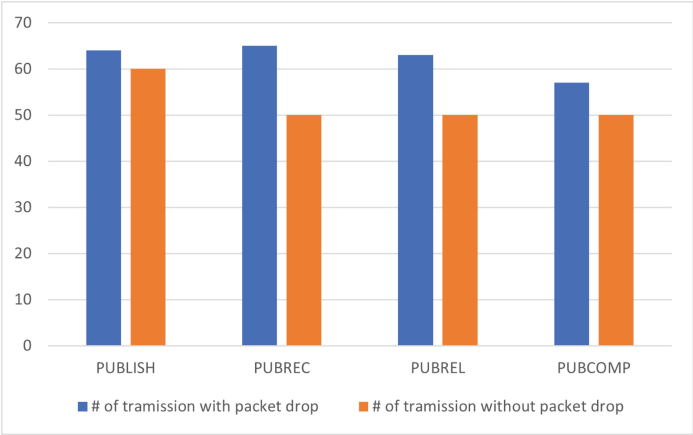

Fig. 8. Number of packets exchanged in the simulation with and without retransmission.

6 Conclusion

In this work, we applied Discrete Event System Specification (DEVS), a modular and hierarchical formalism for the model-driven development of IoT devices. We developed two sets of models to be used in both simulation and deployment on the device. Each model was responsible for a specific functional aspect of the device, and their composition created the complete application. In our case study, we applied the models in a simulation environment that included a sensor node and an actuator for an irrigation system. The results of the simulation showed timestamps for different events occurring in the system and provided metrics related to packet transmission in the MQTT protocol.

For our future work, we aim to develop models that implement more functionalities of IoT devices and for a wider range of hardware. The conditions that cause packet retransmission can be further investigated to be based on real scenarios. Hence, the metrics obtained from the simulations would more closely resemble the deployment environment.

References

1. Da Xu, L., He, W., Li, S.: Internet of Things in industries: a survey. IEEE Trans. Ind. Inf. **10**(4), 2233–2243 (2014)
2. Noura, M., Atiquzzaman, M., Gaedke, M.: Interoperability in Internet of Things: taxonomies and open challenges. Mob. Netw. Appl. **24**, 796–809 (2019)
3. Udoh, I.S., Kotonya, G.: Developing IoT applications: challenges and frameworks. IET Cyber-Phys. Syst. Theory Appl. **3**(2), 65–72 (2018)
4. Patel, P., Cassou, D.: Enabling high-level application development for the Internet of Things. J. Syst. Softw. **103**, 62–84 (2015)
5. Nguyen, X.T., Tran, H.T., Baraki, H., Geihs, K.: FRASAD: a framework for model-driven IoT application development. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), pp. 387–392. IEEE (2015)

6. Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., Muccini, H.: A model-driven engineering framework for architecting and analysing wireless sensor networks. In: 2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA), pp. 1–7. IEEE (2012)
7. Saadawi, H., Wainer, G.: Verification of real-time DEVS models. In: Proceedings of the 2009 Spring Simulation Multiconference, pp. 1–8. Citeseer (2009)
8. Labiche, Y., Wainer, G.: Towards the verification and validation of DEVS models. In: Proceedings of 1st Open International Conference on Modeling & Simulation, pp. 295–305. Citeseer (2005)
9. Olsen, M.M., Raunak, M.S.: A method for quantified confidence of DEVS validation. In: SpringSim (TMS-DEVS), pp. 135–142 (2015)
10. Manrique, J.A., Rueda-Rueda, J.S., Portocarrero, J.M.: Contrasting Internet of Things and wireless sensor network from a conceptual overview. In: 2016 IEEE International Conference on Internet of Things (iThings), pp. 252–257. IEEE (2016)
11. Gupta, H., Vahid Dastjerdi, A., Ghosh, S.K., Buyya, R.: IFogSim: a toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. Softw. Pract. Experience **47**(9), 1275–1296 (2017)
12. Sotiriadis, S., Bessis, N., Asimakopoulou, E., Mustafee, N.: Towards simulating the Internet of Things. In: 28th International Conference on Advanced Information Networking and Applications Workshops, pp. 444–448. IEEE (2014)
13. Lin, Y.-W., Lin, Y.-B., Yen, T.-H.: Simtalk: simulation of IoT applications. Sensors **20**(9), 2563 (2020)
14. Nayyar, A., Singh, R.: A comprehensive review of simulation tools for wireless sensor networks (WSNs). J. Wirel. Netw. Commun. **5**(1), 19–47 (2015)
15. Arslan, S., Ozkaya, M., Kardas, G.: Modeling languages for Internet of Things (IoT) applications: a comparative analysis study. Mathematics **11**(5), 1263 (2023)
16. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: a language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 125–135 (2016)
17. Thramboulidis, K., Christoulakis, F.: UML4IoT—a UML-based approach to exploit IoT in cyber-physical manufacturing systems. Comput. Ind. **82**, 259–272 (2016)
18. Costa, B., Pires, P.F., Delicato, F.C., Li, W., Zomaya, A.Y.: Design and analysis of IoT applications: a model-driven approach. In: 14th International Conference on Dependable, Autonomic and Secure Computing, pp. 392–399. IEEE (2016)
19. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. Academic Press (2000)
20. Im, J.H., Oh, H.-R., Seong, Y.R.: Simulation of a mobile IoT system using the DEVS formalism. J. Inf. Process. Syst. **17**(1), 28–36 (2021)
21. Maatoug, A., Belalem, G., Mahmoudi, S.: A location-based fog computing optimization of energy management in smart buildings: DEVS modeling and design of connected objects. Front. Comp. Sci. **17**(2), 172501 (2023)
22. Barakat, G., Al-Duwairi, B., Jarrah, M., Jaradat, M.: Modeling and simulation of IoT botnet behaviors using DEVS. In: 2022 13th International Conference on Information and Communication Systems (ICICS), pp. 42–47 (2022)
23. Albataineh, M., Jarrah, M.: DEVS-IoT: performance evaluation of smart home devices network. Multimed. Tools Appl. **80**, 16857–16885 (2021)
24. Albataineh, M., Jarrah, M.: DEVS-based IoT management system for modeling and exploring smart home devices. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 73–78. IEEE (2019)

25. Kim, S., Cho, J., Park, D.: Accelerated DEVS simulation using collaborative computation on multi-cores and GPUs for fire-spreading IoT sensing applications. Appl. Sci. **8**(9), 1466 (2018)
26. Etemad, M., Aazam, M., St-Hilaire, M.: Using DEVS for modeling and simulating a Fog Computing environment. In: 2017 International Conference on Computing, Networking and Communications (ICNC), pp. 849–854. IEEE (2017)