Applying Model Driven Engineering Techniques to the Development of Contiki-based IoT Systems

Tansu Zafer Asici
International Computer Institute
Ege University, Izmir, Turkey
tansu.asici@gmail.com

Burak Karaduman
International Computer Institute
Ege University, Izmir, Turkey
bburakkaraduman@gmail.com

Raheleh Eslampanah
Electric and Electronics Engineering
Izmir University of Economics, Izmir, Turkey
raheleh.eslampanah@ieu.edu.tr

Moharram Challenger
Electronics and ICT Department
University of Antwerp, Belgium
and *Flanders Make*, Belgium
moharram.challenger@uantwerpen.be

Joachim Denil
Electronics and ICT Department
University of Antwerp, Belgium
and *Flanders Make*, Belgium
joachim.denil@uantwerpen.be

Hans Vangheluwe
Mathematics and Computer Science Dept.
University of Antwerp, Belgium
and *Flanders Make*, Belgium
hans.vangheluwe@uantwerpen.be

***Abstract*—The huge variety of smart devices and their communication models increases the development complexity of embedded software for the Internet of Things. As a consequence, development of these systems becomes more complex, error-prone, and costly. To tackle this problem, in this study, a model-driven approach is proposed for the development of Contiki-based IoT systems. To this end, the available Contiki metamodel in the literature is extended to include the elements of WiFi connectivity modules (such as ESP8266), IoT Log Manager, and information processing components (such as Raspberry Pi). Based on this new metamodel, a domain-specific modeling environment is developed in which visual symbols are used and static semantics (representing system constraints) are defined. Also, the architectural code for the computing components of the IoT system such as Contiki, ESP8266, and RaspberryPi are generated according to the developer's instance model. Finally, a Smart Fire Detection system is used to evaluate this study. By modeling the Contiki-based IoT system, we support model-driven development of the system, including WSN motes and sink nodes (with ContikiOS), WiFi modules and information processing components.**

***Index Terms*—Model-driven Engineering (MDE), Internet of Things (IoT), Embedded Software, Wireless Sensor Network, ContikiOS, Smart Fire Detection System**

I. Introduction

Internet of Things (IoT) systems are rapidly taking their place in different technologies and markets, such as home appliances, smart buildings, and Industry 4.0 applications. In these systems, devices communicate with each other and work in coordination, to create an intelligent environment. Generally, these devices can communicate in two ways. The first method is used by devices with Internet connection capability (e.g., WiFi connection, using the IEEE 802.11 protocol) where they communicate by connecting directly to the Internet. WiFi connection suffers from various problems such as limited access range to cover large areas (introducing the need for range extenders which results in extra costs and more setup complexity). The second method is used by low power devices using 802.15.4 protocol and point to point communication, such as Wireless Sensor Networks (WSN). This communication protocol can route the transmitting data through neighbor nodes (also called sensor motes). Thus, they are spreadable and can cover wide areas. These two communication protocols can be combined so new IoT systems can have both accessibility via Internet and have wide area coverage.

However, these communication protocols need different underlying technologies and therefore, different programming languages, which make the embedded software development complex, error-prone, and costly. Also, these systems usually require other, complementary parts. For example, a WSN based IoT system may require a gateway to transmit the data to the Internet using the IEEE 802.15.4 protocol. Furthermore, an information management sub-system may be needed. This information management sub-systems should process the information received from the WiFi modules and WSN devices. It can be implemented using a desktop application, a web application with access to a database, or even by a cloud based system. Moreover, this application needs to be written in a yet another programming language. Designing and developing such a system adds even more complexity.

To address this complexity, a Model-Driven Engineering (MDE) approach can be used to increase the level of abstraction and automatically synthesize some of the artifacts. For this purpose, in this study, a modeling environment is proposed for the development of Contiki [8] and ESP8266 (a low cost WiFi transceiver module)[1] based IoT systems which are using both WSN and WiFi communication. As MDE raises the abstraction level of the software development process, this approach can reduce the development cost and time as well as decrease the number of errors which in turn leads to a shorter testing time.

To apply the MDE approach, domain-specific modeling

This study is partially funded by the Scientific Research Project No 17-UBE-002 at EGE University, Izmir-Turkey.

[1]https://www.espressif.com/en/products/hardware/esp8266ex/overview

languages can be used [6]. Domain-specific modeling allows domain-experts to reason in their problem domain rather than in the solution domain. Domain-specific languages (DSL) are defined by specifying the concepts and relations between components within the domain by using a meta-model. The concrete syntax of the language (i.e. used in the language editor) can be textual and/or graphical. Finally, the semantics of the language can be defined by using a set of model to code/text transformation rules.

In our previous study [9], a meta-model was proposed for ContikiOS. The idea was purely about modeling the ContikiOS programs. In this study, this meta-model is extended, a new graphical editor editor is developed and the transformation rules are defined for code generation. The modeling environment is extended to support the other required IoT components and systems such as WiFi based modules/micro-controllers (i.e. ESP8266) and WSN gateway devices (such as a Java program on a Raspberry Pi[2]) as well as the data processing and repository such as a IoT Log Manager System. A fire detection system [1] is used as a case study to evaluate the proposed language.

This paper is organized as follows: Section II discusses the related work. The extended meta-model is introduced in Section III. The modeling environment, including the graphical concrete syntax and the static semantics (constraints), is described in Section IV . The Code generation mechanism for both target domain and model checking tools is presented in Section V. Section VI demonstrates the case study and Section VII evaluates the study and discusses the results. The paper is concluded in section VIII.

II. Related Work

In the literature, some MDE studies have been conducted in order to facilitate the design, development and implementation of WSN and IoT systems. Two surveys ([10] [14]) carried out a systematic mapping study of this domain.

According to the above-mentioned surveys, different MDE-based languages developed between the years 2007 and 2015. Furthermore, the modeling motivation of most of these studies is code-generation to increase productivity [10]. For example, the nesC language has a code generation capability for TinyOS. However, none of these studies address ContikiOS.

LwiSSy is a DSL [4] to model Wireless Sensor and Actuators Network (WSAN) systems. It allows the separation of responsibilities between domain experts and network experts. It also considers the separation of structure, behavior, and optimization concerns by using multiple views. In the study of [15], a model-driven architecture (MDA) is proposed which provides platform independent modeling (PIM), platform-specific modeling (PSM), and transformation rules for WSAN application development. Doddapaneni et al. [7] proposed a framework to separately model the software components and their interactions, the low-level and hardware specification of the nodes, and the physical environment where the nodes are deployed. This multi-view architectural approach requires linking the models together for mapping the models.

Tei et al. [17] propose a process that enables step-wise refinement to separately address data processing-related and network-related concerns. Their approach is similar to the modeling purpose of Rodrigues et al. [15]. They focus on the separation of responsibilities between domain experts and network experts for the model-based engineering of systems working on TinyOS. However, Tei et al. [17] have limited support for experts. PSM is not supported in their study and the experts simply create templates over platform independent models.

Our work contributes to the aforementioned noteworthy studies in the way of providing a model-driven engineering method for developing Contiki based IoT systems. To the best of our knowledge, currently no study addresses modeling of WSNs based on ContikiOS and required components to build a IoT system. In Durmaz et al. [9], an MDE approach is applied for Contiki-based devices by providing a meta-model for components of Contiki operating system. This meta-model is adopted and extended in our study. The unique lightweight thread structure of Contiki makes it more useful in the implementation of the WSN systems [12], [13]. This can be considered as a reason for the developers to prefer ContikiOS instead of the other existing platforms such as TinyOS. Hence, providing a modeling language as proposed in this study can facilitate the efficient development of IoT systems based on WSN.

Many systems and solutions have been designed for the fire detection system case study we address in this paper. In [11] the k-coverage algorithm is used for fire detection. In study [2], WSN devices communicate using mesh network and measure smoke, humidity and temperature sensors and these values are interpreted then concluded about fire risk. Unlike creating a mesh network like [2], the study in [5] uses GPS technology to transmit data directly to the base station and the data be accessed using web browsers. In another study [16], ad-hoc and multi-hop networks are used to detect fires in nature and send alarm messages using the shortest path algorithm. In our previous studies, [13] and [12], a cloud based library fire detection system is designed and implemented with single hop and multi hop WSNs using ContikiOS. However, none of these studies have been done using domain-specific modeling in the design and development steps.

In conclusion, this study includes the modeling of Contiki operating system, WiFi connectivity devices and other required components to develop the IoT system. The WiFi module addressed in this study is ESP8266 and the complementary components are IoT Log Manager and information processing device (such as the program on a Raspberry Pi). Furthermore, this study proposes a meta-model including the required elements of the system, graphical editor, static semantic, and code generation for the modeling environment.

[2]https://www.raspberrypi.org/

III. The Extended Meta-model

The meta-model introduced in this Section represents the abstract syntax of the domain-specific language proposed in this study. Generally, the meta-model defines the elements, the relations between the elements, and possibly the cardinally constraints of the relations.

In this study, we first update the meta-model in [9] by adding the missing features of the ContikiOS such as *Process Event* and *Process Post*. Then, the meta-model is extended with the elements and relations of the other components in the IoT such as ESP8266 representing a WiFi connectivity module, RaspberryPi representing a gateway, and IoT Log Manager system. In general, the proposed meta-model in this study is the result of analyzing different WSN programs working in IoT system. This task is done in close collaboration with WSN group.

WSN elements are represented with the ContikiOS elements in the meta-model including *Events*, *UDP/TCP protocols*, *Processes*, *Timers*, etc. For details about the ContikiOS part of the meta-model, the interested readers can refer to our previous study in [9]. Considering the space limitations, the extended meta-model is not shown in this paper. However, it is available on our online materials related to this paper at: Link. However, the key elements of extended parts are briefly described in this section.

The required elements and relations of the ESP8266 are added to the meta-model. All the elements of the ESP8266 are connected to a main element called *ESP* in the meta-model. There are five attributes in *ESP* element: *Name* attribute for device name, *SSID* and *Password* attributes are the credentials to connect a access point. *Host* attribute is a key parameter (API Key) to connect to the IoT Log Manager, and *SerialNo* attribute is a defining value (Tag name) to send the data generated by ESP8266 to the IoT Log Manager. The *HttpClient* element has an object *Name* attribute to make requests to the server; and *WebServer* element has a *Name* and *PortNo* attributes to create a web server inside ESP8266 and open a listening port for communication. *Wifi* element has *Mode* attribute that specifies in which WiFi mode ESP8266 operates and *Status* attribute represents the status of the established connection. *WifiMulti* element is used to connect ESP8266 to a access point in WiFi range. Moreover, a *servomotor* element is considered in the meta-model as an actuator, e.g., in the fire detection system to open doors automatically during alarm state. The *Port* attribute is indicating the listening port to interpret incoming *POST* requests. *Angle* attribute specifies rotation angle for servo motor, to open or close the door. A servo motor can be controlled by a ESP8266 using *Network* element, and it is accessible by specifying the parameters of *IP*, *Gateway* and *Sub-net* which are sub-elements of the *Network* element.

Elements of the *application* (e.g., a Java application) on a computing device such as Raspberry Pi provides connection between the WSN Nodes (specifically the Sink Node) which are using Contiki operating system and the parts of the system. *PortName* attribute indicates *Serial-port* name which physically connects RaspberryPi and Sink Node. *URLConnection* element is an element which provides network connection between RaspberryPi and the IoT Log Manager. *BufferedReader* and *OutputStreamWriter* supply RaspberryPi to read, and parse and match the data which comes from Sink node. This data usually contains node id, sensor values, and IPv6 address. Additionally, *JavaTimer* element allows the developer to call desired function in specified periods (in seconds).

The IoT Log Manager receives the sensor data from the sink nodes via gateway and also directly from ESP8266. It is possible to create various events in the IoT Log Manager. The events can be created by the end users. *Channel* is the element where data for a specific IoT system is preserved. The users can create as many *Channel*s as they need and each *Channel* has its own special *read* and *write* API key. Using these API keys, data can be sent to the channel or it can be received from channel by *POST* requests.

Finally, the main element of the system meta-model is *IoTSystem* which contains *Contiki*, *ESP8266*, *RaspberryPi* and *IoT Log Manager* elements, and it has *Name* attribute to specify the system name. The meta-model is implemented using the Eclipse Modeling Framework (EMF) Ecore [3] and is serialized in an XMI format.

IV. The Concrete Syntax and Static Semantics

In this section, the graphical concrete syntax (i.e. the Graphical Editor) for modeling and development of ContikiOS based IoT systems is introduced. This graphical editor is developed using the Eclipse Sirius Framework [4].

Generally, the concrete syntax provides a mapping between some graphical notations and abstract elements. In this study, the graphical notations used to represent the elements in the editor are shown in Figure 1.

The developed domain-specific modeling environment consists of a multi-view and layered structure. The top layer contains the general view in the modeling of the system, and the lower layer includes Contiki view, RaspberryPi view, Log Manager view and ESP8266 model view. This reduces the complexity of the system design as the designer works with a specific view at a time. Thus, the developed graphical concrete syntax has five main views. The first view is for System layer which provides the high level view of the system. This view includes representative elements for the other view in the lower layer. An instance of System view is shown in Figure 2.

In Figure 2, the *Contiki* element is connected to *RaspberryPi* via *Contiki* relation to specify that the nodes with ContikiOS send the data to the RaspberryPi and it forwards to the corresponding channel. To transfer the data to a channel in IoT Log Manager system, *RaspberryPi* uses related *Tags* for each sensor node.

ESP8266 has a direct connection with the *Log Manager* since it transfers the data directly to the *Log Manager* using

[3] https://www.eclipse.org/modeling/emf/

[4] https://www.eclipse.org/sirius/

Icon	Concept	Icon	Concept
Br	BufferedReader		ProcessPost
	Channel		ProcessThread
	Class		Psock
	ClientConnection		PT
	Ctimer		PTThread
	Data		PT ThreadCall
	ESP		Raspberry Pi
	Etimer		Rtimer
	Evet		ServerConnection
	Field		ServoMotor
	HttpClient		Stimer
	IoTSystem		Tag
	JavaTimer		TCPIPEvent
	LogManager		TimeEvent
	Multithreading		TimeoutEvent
	Network		Timer
	Node		URLConnection
OSw	OutputStreamWriter		WebServer
	Pin		WiFi
	Contiki		WiFiClient
	Port		WiFiMulti
	ProcessEvent		

Fig. 1. The graphical notations of the elements in the editor

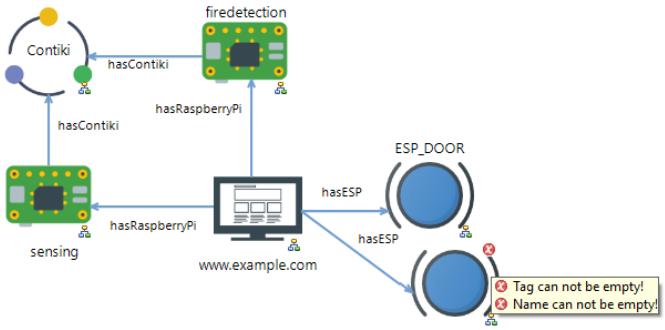

Fig. 2. High-level system view in the modeling environment

POST requests. The *Log Manager* is the central component of the system where all the data transfers end. It store the data and use them to make a decision to trigger another device, such as an actuator, or to do reporting, such as displaying some graphs/diagrams to the user.

Another view is *Contiki* view in which *WSN mote* elements are located. In this view, the designer can use the concepts and relations of the WSN to model the ad-hoc communication part of the IoT system. These elements are shown in Figure 1.

Figure 3 shows the *ESP8266* (on the top) and *RaspberryPi* (at the bottom) views. It includes the palette elements and relations at the right hand side, and simple instance models at the left.

For the *ESP* view, some of the main graphical notations

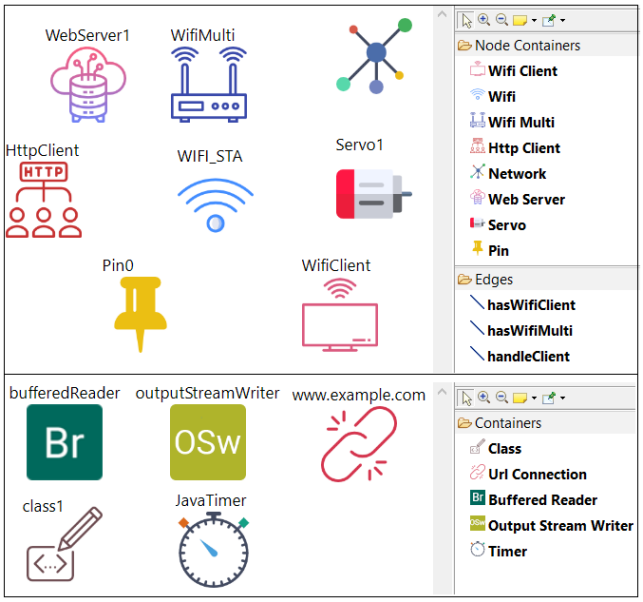

Fig. 3. Elements in ESP and RaspberryPi Views

(icons) are the *Web Server* representing the server and cloud, *WiFi Multi* representing multiple WiFi connections, *Servo* representing a Servo motor, and *Pin* representing the pin which *ESP8266* actively uses.

For the *RaspberryPi*, as it is programmed in Java, the elements in this view belong to Java programming language. For example, *JavaTimer* element (see Figure 3) specifies the Timer in Java and *UrlConnection* element is used to connect to a specific URL, in our use case RaspberryPi is connected to the IoT Log Manager web address. *BufferedReader* is used to read the data from *SinkNode* and *OutputStreamWriter* is used to write the data to *IoT Log Manager*.

The last view is the model of *IoT Log Manager*. This view can be accessed via the *IoT Log Manager* element in the *System* view. The *IoT Log Manager* elements such as *Channel* and *Tag* are shown in Figure 4 for a fire detection channel and sensing channel with some tags for specific sensors. These elements are part of the case study discussed in Section VI.

Basically, *ESP8266* and *RaspberryPi* transfer the data to a specific predefined channel using tags. For example, in a system with two channels, each with three sensors (see Figure 4), the data can be transferred to *firedection* channel using tags *sensor1*, *sensor2* and *sensor3*. The tags should be labeled in a unique way which is controlled by the *IoT Log Manager*.

To apply the domain rules in the modeling environment and help the user to design more accurate models, some semantic constraints are implemented in the framework using Acceleo Query Language (AQL)[5] which is built-in the Eclipse Sirius Framework. Some of these semantic constraints are presented bellow for the system view of our modeling environment.

Example rule 1: $aql : self.Name.size() > 0$

Example rule 2: $aql : self.esp\_tag- > size() > 0$

[5]https://www.eclipse.org/sirius/doc/specifier/general/Writing_Queries.html

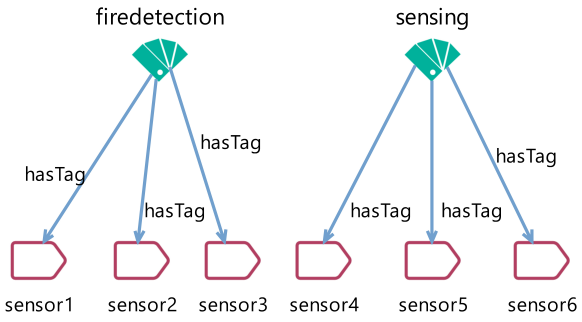

Fig. 4. IoT Log Manager View

The example rule 1 checks whether the *Name* attribute is blank or not, and the example rule 2 checks whether the tags which are necessary to connect the elements to the *IoT Log Manager* are available or not. As the result of these example rules, the framework forces the designer to fill the name spaces and provide the related tags. If the conditions are not satisfied then error messages appear in the user model as shown in Figure 2. This leads to improve the model before going to the next phase, the code and specification generation.

It is worth to note that, although the *Contiki* view is rather comprehensive and represents the main features of this WSN operating system, the other views including *ESP* and *RaspberryPi* are representative as they can be extended based on the required components (sensors and actuators) in the application domain (and related product family).

V. Translational Semantics

The semantics of the proposed modeling language is defined by translation of the user models to both a specification for design verification and to the target domain artifacts. These transformations are realized using the Acceleo [6] template-based model transformation engine.

A. Generation of a Design Verification Model

It is important to verify the system design model before it is implemented. To this end, the design model needs to be transformed to a formal specification/model for which required properties can be checked. This helps the designer to inspect the model and to check the structure or behavior of the system to conform to domain rules. In this way, unpredictable problems and errors can be detected during the design phase.

Petri-nets is a formalism used for verification of pertinent properties of automation systems, embedded systems, and network systems. In this study, we use Petri-nets to check, a.o., for the occurrence of bottlenecks as well as for reachability or (un)desirable states.

To this end, the user instance models (the whole IoT system model) are automatically transformed to Petri-net stochastic models in the Pipe tool to check the required properties. For this purpose, the instance model elements are transformed to places; relations are transformed to arcs; sensors' data is transformed to tokens; and transitions are added to combine two or more arcs.

[6]https://www.eclipse.org/acceleo/

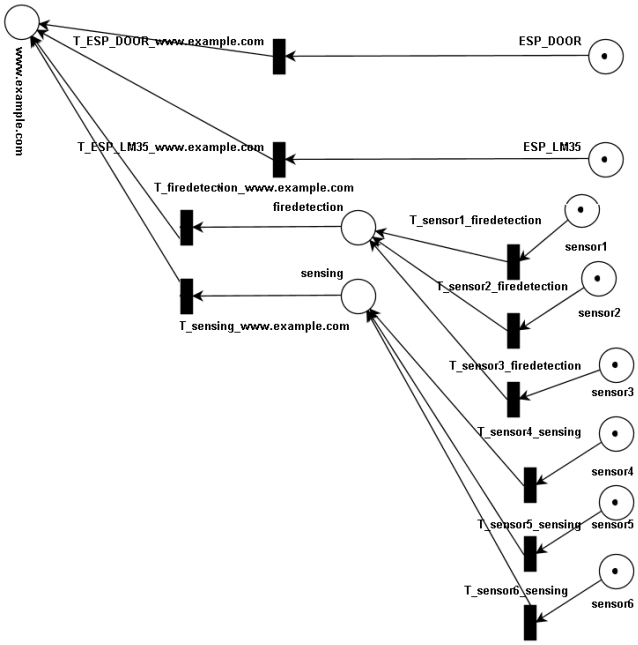

Fig. 5. A Petri-net Model generated from an Instance Model

In Figure 5, the elements which are used in Figures 2 and 4 are represented by a Petri-net. As seen in Figure 5, ESP8266 modules directly send messages and sensor nodes send messages via RaspberryPi to the IoT Log Manager (named www.example.com).

In the transformation rules, the places represent RasperryPi(s), ESP8266(s), sensor nodes(s) and IoT Log Manager. The routes of the messages are encoded by arcs, and message transitions are added to the Petri-net model. The messages are represented by tokens. In Listing 1, an excerpt of the Acceleo transformation rule is given to generate the Petri-net model as an XML file for the Pipe Petri-net analysis tool.

Listing 1. Code excerpt from Petri-net .xml file in Acceleo

```
1[file(Name.concat('.xml'),false,'UTF-8')]
2<?xml version="1.0"encoding="ISO-8859-1"?><pnml>
3<net id="Net-One"type="P/T_net">
4[for(l:LogMan|aIoTSystem.logman)]
5  [for(e:ESP|aIoTSystem.esp)]
6    <place id="[e.Name/]">
7      <graphics><position x="720"y="[90*i/]"/></graphics>
8      <name>
9        <value>[e.Name/]</value>
10       <graphics><offset x="0" y="0"/></graphics>
11     </name>
12     <initialMarking>
13       <value>1</value>
14       <graphics><offset x="0" y="0"/></graphics>
15     </initialMarking>
16     <capacity><value></value></capacity>
17   </place>
18   <transition id="T_[e.Name/]_[l.URL/]">
19   <graphics><position x="435" y="[(i*90)/]"/></graphics>
```

B. Target Domain Artifact Generation

In this study, the translation to the target domain, so called code generation, provides the architectural code for Mote (SourceNode), SinkNode, RaspberryPi (Java Code), ESP8266 (Arduino code), and configuration for IoT Log Manager.

The code generation starts with creating the files of the elements in the IoT system. This is done using a for loop in Acceleo. For example, in Listing 2 Lines 2-4, the Java files for the RaspberryPi elements used in the model are generated. Part of the code is not directly modelled in the design model but it can be concluded indirectly, e.g., importing libraries. In Listing 2 Line 6-9, if the RaspberryPi the model includes a JavaTimer element then the rule will be executed and the import statements will be generated.

Similarly, if the model has ESP8266 element(s), then the related file(s) is/are created with "ESP8266.Name.ino" file name(s). Depending to the ESP8266 elements, the importing rule for that element, such as the one in Lines 11-13, is run and the related library is imported.

For each Contiki model/view, a C file is created. However, two types of nodes, Mote and Sink-Node, can be implemented using ContikiOS for each of which separated code file must be generated. To specify this separation, SinkNode has Server-Connection and Mote has ClientConnection element. In an instance model, if a node has a relation with a ServerConnection element, it is a SinkNode. If it does not have it, then it must be determined as Mote. This rule is shown in Listing 2 Lines 15-19.

Listing 2. Excerpt of the Acceleo rules for generating some of the files and importing their libraries

```
1[template public generateElement(aIoTSystem:IoTSystem)]
2[for(c: Class|aIoTSystem.raspberrypi.class)]
3  [file(c.Name.concat('.java'),false,'UTF-8']
4  package[aIoTSystem.raspberrypi.raspberry_channel.Name/];
5...
6[if(aIoTSystem.raspberrypi.javatimer->size()>0)]
7  import java.util.Timer;
8  import java.util.TimerTask;
9[/if]
10...
11[if(e.servo.Name.toString()<>'invalid')]
12  #include <Servo.h>
13[/if]
14...
15[for(n:Node|aIoTSystem.platform.node)]
16  [for(p:Process_Thread|n.processThread)]
17    [if(p.hasServerConnection.Name=
18      aIoTSystem.platform.serverConnection.Name)]
19      [file(n.Name.concat('.c'), false, 'UTF-8')]
```

To use UDP or TCP communication in the Contiki between the WSN motes, the required code is generated using the rule listed in Listing 3.

To create reply message a TCPIP_Event element must be added to the model. Since Contiki is an event based operating system, if any UDP or TCP packet is received, an event named tcipip_event is generated by the operating system. Therefore sending the reply message is bounded to this event.

Listing 3. Send reply according to UDP/TCP connection

```
1[let a:String=n.processThread.handlesEvent->
2  select(event|event.eClass().name='TCPIP_Event')->
3  at(i).eGet('replyMessage')]
4[if(n.UDP)] //NOTE:Add Server Connection to the model
5  PRINTF("DATA sending reply\n");
6  uip_ipaddr_copy(&[p.hasServerConnection.Name/]->
7    ripaddr, &UIP_IP_BUF->srcipaddr);
8  uip_udp_packet_send([p.hasServerConnection.Name/],
9    "[a/]", sizeof("[a/]"));
10 uip_create_unspecified(
11   &[p.hasServerConnection.Name/]->ripaddr);
12[/if]
13[if(n.TCP)] //TCPIP Reply
14  sprintf(buf, "[a/]"(%d)",1);
15  PRINTF(buf);
16  PRINTF("\n\r");
17  uip_send(buf,strlen(buf));
18[/if]
19[/let]
```

Using these rules the required files and their architectural code are generated automatically for different components of the IoT system. These artifacts should be completed using the user's delta code defining the domain-specific business logic such as the controlling conditions to have a fully functional system.

VI. Case Study: Smart Fire Detection System

The proposed modeling framework is evaluated using a use case called fire detection system. In this domain, the IoT components are used to recognize the signs of a fire in a large area and to react instantly.

To achieve this, sensor nodes sample the temperature in each 4 seconds. Thus, the fire can be prevented at the earliest possible time. In the system view of this case study, see Figure 6, the IoT Log Manager is located in the center of the system. SinkNode, ESP8266 Module (with its LM35 sensor), and RaspberryPi are connected to the IoT Log Manager. Motes can send packets to a sink node using multi-hop communication. The SinkNode takes the data from the closest source node (Mote) and transfers the data to RaspberryPi. Since RaspberryPi has direct access to IoT Log Manager, it transmits the data which is collected from sensors to a relevant channel by matching sensor's Tag. ESP8266 with a LM35 sensor, measures the temperature of its surrounding, and sends the temperature data to the IoT Log Manager.

The IoT Log Manager keeps the received data in PostgreSQL database, and visualizes them using the Blazer query tool (see Figure 7) based on the user query script. This helps the user to provide the desired reports.

In IoT Log Manager, a channel is defined by giving a Channel name and Channel description and the events are created in this Channel to control the conditions and trigger proper actuators. The end users can use his/her logical comparisons for events or write their own complex scripts to create various events. The IoT Log Manager compares the values which comes from RaspberryPi and ESP8266 and if the temperature is above a set-point, then it sends an e-mail to alert the people who are in charge. Moreover, Notifications are sent to smartphones via Pushbullet[7] application to alert the user about the danger.

There are also two ESP8266 modules with servo motors in the system that control the door locks. In the case of

[7]https://www.pushbullet.com/

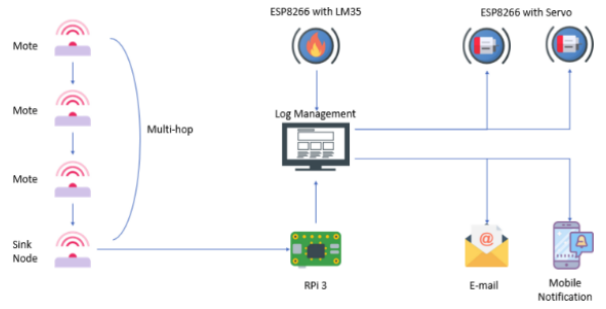

Fig. 6. Design of smart fire detection system

Fig. 7. Report and graph generation screen with query tool

emergency, the IoT Log Manager sends POST messages to the ESP8266(s) to release the door locks (to open the doors). Of course, there can be any number of doors or any other devices which can be controlled by an event. Finally, the people in the building can send their status, such as pushing the Safe button in the app, as the feedback to the system.

Due to page limitations, some of the other views and models of this case study are shown as examples in the previous sections. For example, the IoT Log Manager view of this case study is shown in Figure 4 modeling the related channels and sensor tags. Also, in Figure 5 the generated Petri-net model for this case study is demonstrated. Using this model, the bottleneck analysis is done for this system. Also, the complete model of Contiki view is not shown in this paper due to its size. However, it is available in the online bundle of the paper discussed in Section III.

Finally, for testing the use case, the motes were placed at a distant where they can communicate with each other and create a mesh network. While RaspberryPi was receiving sensor data which was collected by Sink Nodes and ESP8266 modules from the environment, one of the nodes was heated by a lighter. When the IoT Log Manager has detected that the temperature is rising, a callback has been sent to the ESP8266 which controls the servo motor to open the door lock. At the same time, a notification has been sent to the smart phone and an e-mail has been sent to the user to warn about danger.

VII. Evaluation

The general evaluation of the modeling platform is done via assessing its generation performance. To this end, we

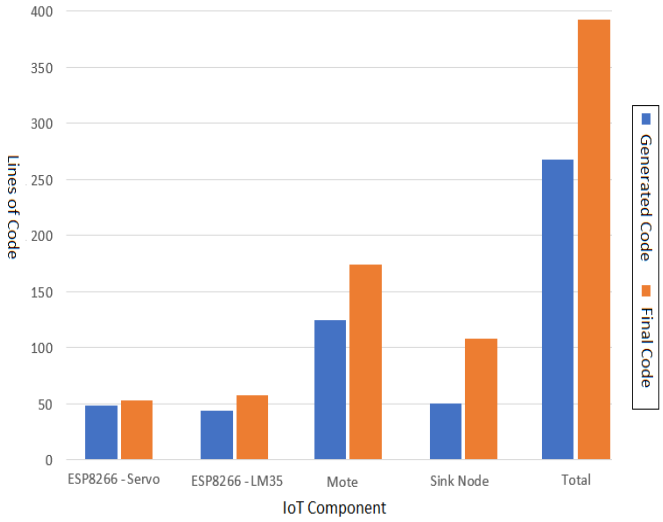

Fig. 8. Generation performance of the proposed modeling framework

have analyzed the generation capability of the framework by comparing amount of the generated architectural code with the final code (the code after adding the end-user's delta code).

We are aware that the evaluation with single use case has the external threat to the validity of the results which can risk the generalization of the results. However, a systematic and extensive evaluation of the approach with multi case study will be done in our next study which will focus both the generation performance and development time performance [3].

Figure 8 shows the comparison of generated lines of code and lines of the final code for different components of the IoT system. According to these figures, the generation performance of the IoT component in this study are about 96% for ESP8266-Servo, 78% for ESP8266-LM35, 71% for Motes, 47% for Sink-Node, and 70% in the total. These numbers show the rate of generated code considering the final code.

As you can see, the Sink Node has the lowest generation rate of almost half of the code, and ESP8266-Servo has the highest with almost all of the code generated automatically. But the Sink Node is not the biggest part of the total code. The biggest part of the code belongs to the Mote with 71% of generation. As the result, more that two third of the whole code in the smart fire detection system is generated.

It is worth to discuss that, generally, some parts of the code cannot be generated in this modeling environment, as they are very specific for the scenarios and the model does not cover that much detail. In fact, the level of abstraction in the modeling language is a design choice. If you involve too much details in the model, the higher level of abstraction can be violated, as the model can be as complex as the target code itself. This is a trade-off which should be assessed by evaluations to reach the optimum level of abstraction which considers both ease of design (less design complexity and less design time) and higher generation performance.

VIII. Conclusion and Future Work

In this study, a domain specific modeling language and its supporting tools are developed for design and implementation of IoT systems to address the increasing complexity and difficulty of these systems. The code generation mechanism provides the architectural codes for ESP8266, ContikiOS and RaspberryPi as well as the specification for design verification in Petri-net.

To this end, the Contiki meta-model available in the literature (implemented in EMF) is extended by the other components required for an IoT system such as Sensors connected to a transceiver module (ESP8266), a computation device as the gateway, and a Log Manager. This meta-model is used as an abstract syntax of the DSML and a set of graphical notations as well as some domain constraints are used to develop the graphical editor for the proposed modeling environment.

ContikiOS provides a light-weight multi-threading mechanism for bi-directional and multi-hop communication of sensor nodes which enables them to cover a large area. RaspberryPi has the important role of bridging the different communication approaches which are used in Contiki based WSNs (IEEE 802.15.4 protocol) and WiFi Modules such as ESP8266 transceiver module (IEEE 802.11 protocol). Finally, Log Manager is the event based central data repository of the system which also handles the triggering the actuators and reporting the data.

Furthermore, the system automatically generated Petri-net model helps the end user or IoT developer to analyze and verify the design model at the early phases of the development, before any implementation, to check the critical properties for the IoT system under development.

Finally, based on the evaluation results, the proposed approach for the development of IoT systems, using the provided generative modeling environment, can reduce the development time and cost.

As a future work, the evaluation of the performance of the proposed approach will be done in a systematic way, using multi case studies and two groups of developers. In this way, we will be able to evaluate the development-time efficiency of the approach as well as generation-performance of the modeling framework.

As another future work, the level of abstraction for the modeling environment is going to be increased to also support Platform Independent Level. The current study, focuses on a set of specific target platforms (e.g., ContikiOS) and provides a Platform-specific Modeling environment for the development of IoT systems using WSNs and WiFi modules. This future work will extend the current work to support modeling of IoT systems independent of their target platforms. This will be done using commonality and variability analysis of WSN Operating Systems such as ContikiOS, TinyOs, RIOT and so on, and also using model to model transformations.

Acknowledgment

Some of the ideas in this work were developed during Short Term Scientific Missions (STSMs Nos. 41940 and 41967) by two of the authors, within the IC1404 Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS) COST Action.

References

[1] S. Arslan, M. Challenger, and O. Dagdeviren. Wireless sensor network based fire detection system for libraries. In *Computer Science and Engineering (UBMK), 2017 International Conference on*, pages 271–276. IEEE, 2017.

[2] Z. Chaczko and F. Ahmad. Wireless sensor network based system for fire endangered areas. In *Third International Conference on Information Technology and Applications (ICITA'05)*, volume 2, pages 203–207, July 2005.

[3] M. Challenger, G. Kardas, and B. Tekinerdogan. A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, 24(3):755–795, 2016.

[4] P. Dantas, T. Rodrigues, T. Batista, F. C. Delicato, P. F. Pires, W. Li, and A. Y. Zomaya. Lwissy: A domain specific language to model wireless sensor and actuators network systems. In *2013 4th International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pages 7–12, May 2013.

[5] N. S. David M. Doolin. Wireless sensors for wildfire monitoring. In *Proc.SPIE*, volume 5765, pages 5765 – 5765 – 8, 2005.

[6] S. Demirkol, M. Challenger, S. Getir, T. Kosar, G. Kardas, and M. Mernik. Sea_l: a domain-specific language for semantic web enabled multi-agent systems. In *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*, pages 1373–1380. IEEE, 2012.

[7] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta, L. Mostarda, and H. Muccini. A model-driven engineering framework for architecting and analysing wireless sensor networks. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, SESENA '12, pages 1–7, Piscataway, NJ, USA, 2012. IEEE Press.

[8] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov 2004.

[9] C. Durmaz, M. Challenger, O. Dagdeviren, and G. Kardas. Modelling Contiki-Based IoT Systems. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, volume 56 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:13, Dagstuhl, Germany, 2017.

[10] F. Essaadi, Y. Ben Maissa, and M. Dahchour. Mde-based languages for wireless sensor networks modeling: A systematic mapping study. In R. El-Azouzi, D. S. Menasche, E. Sabir, F. De Pellegrini, and M. Benjillali, editors, *Advances in Ubiquitous Networking 2*, pages 331–346, Singapore, 2017. Springer Singapore.

[11] M. Hefeeda and M. Bagheri. Wireless sensor networks for early detection of forest fires. In *2007 IEEE International Conference on Mobile Adhoc and Sensor Systems*, pages 1–6, Oct 2007.

[12] B. Karaduman, T. Aşıcı, M. Challenger, and R. Eslampanah. A cloud and contiki based fire detection system using multi-hop wireless sensor networks. In *Proceedings of the Fourth International Conference on Engineering & MIS 2018*, page 66. ACM, 2018.

[13] B. Karaduman, M. Challenger, and R. Eslampanah. Contikios based library fire detection system. In *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pages 247–251. IEEE, 2018.

[14] I. Malavolta and H. Muccini. A study on mde approaches for engineering wireless sensor networks. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 149–157, Aug 2014.

[15] T. Rodrigues, F. C. Delicato, T. Batista, P. F. Pires, and L. Pirmez. An approach based on the domain perspective to develop wsan applications. *Software & Systems Modeling*, 16(4):949–977, 2017.

[16] B. Son, Y.-s. Her, and J.-G. Kim. A design and implementation of forest-fires surveillance system based on wireless sensor networks for south korea mountains. *International Journal of Computer Science and Network Security (IJCSNS)*, 6(9):124–130, 2006.

[17] K. Tei, R. Shimizu, Y. Fukazawa, and S. Honiden. Model-driven-development-based stepwise software development process for wireless sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(4):675–687, April 2015.